Stream-Based Web Service Invocation

Steffen Preißler, Hannes Voigt, Dirk Habich, Wolfgang Lehner
Technische Universität Dresden
Lehrstuhl für Datenbanken
dbgroup@mail.inf.tu-dresden.de

Abstract:
Service-oriented architectures (SOA) based on Web service technology play an increasingly important role in many different application areas. The current service invocation methodology suffers from performance problems and heavy resource consumption when services are used to process large amounts of data. A number of solutions to this problem have been proposed. Unfortunately, these modified invocation methodologies are applicable only to a limited number of business scenarios, because they do not provide all features of the traditional methodology. In this paper, we introduce a new service invocation methodology that allows large volume data processing and does not limit applicability. Our approach macerates the existing request–response paradigm and incorporates stream semantics into the Web service invocation methodology without reducing its features. We describe our stream-based service invocation and present evaluation results that show the advantages of our approach.

1 Introduction

The paradigm of service-oriented architectures (SOA) becomes more and more attractive to a variety of application areas. The base concept of SOA is a service. A service represents a mechanism to enable access to one or more capabilities using a prescribed interface [OAS06]. Web services [W3C02] are the de-facto standard for implementing such a mechanism and thus service-oriented infrastructures. To use a Web service, a client sends a request to the service interface. The service processes this request and afterwards, according to the offered functionality, sends a response back to client. This so called *request–response paradigm* is a foundation of service-orientation and guarantees loosely-coupled and autonomous services.

The communication between a client and the services it invokes is message-based. The size of these messages is performance critical. The well investigated memory consumption and performance problem in the XML-based message communication arises when serializing or de-serializing XML messages to or from other programming language paradigms [CGB02, MF05, CYZ$^+$06, HPL$^+$07]. This inefficiency is owed to the impedance mismatch between the hierarchical XML model and the (mostly object-oriented) programming language model that processes the XML content [vE03]. Thus a conversion needs to be done, which devours up to 90% of the processing time [DA03] and up to ten times the in-memory size of its correspondent text-based XML representation [vE03] when creating the whole Document Object Model (DOM) tree. If large data sets have to be exchanged between client and service, these issues become more evident.

Streaming is a much more efficient way to process large sets of equally structured data items – well studied in the field of data stream system. However, the stream processing paradigm is hard to realize with the current Web service concept, mainly because the request–response paradigm of Web service communication does not support service invocations with continuous data. The message serialization and de-serialization is processed in a single step before the whole message is send and after the whole message arrived, respectively. The straight forward approach of mapping every data items to a single service invocations proved to be very inefficient [GYSD08]. Additionally it strongly restricts the business logic of the service, since it does not allow to process data items in a common context.

In this paper we introduce a concept to add stream semantics to the Web service invocation and its execution. Our innovative Web service invocation approach macerates the traditional *request–response paradigm* to incorporate stream-based semantics and establishes, thereby, an efficient way to transmit and process continuous and large-scale data sets by a specific service. Our concept is (i) more efficient in memory consumption and performance, (ii) provides a common context for all data item in a large volume data set and (iii) allow for more functionality like intermediate results.

The reminder of the paper is organized as follows. First, we present our approach for stream-based Web service invocation and execution in detail (Section 2). Second, we evaluate our approach against current techniques for data transmission on Web service level, as they can be found in current infrastructures, in terms of response time (Section 3). Finally, we conclude and give a short outlook on future work (Section 4).

2 Stream-based Web Services

In this section we introduce our approach to incorporate stream semantics into the Web Service invocation and its execution. It enables clients to transfer large scale and continuous data to a service as a stream. Furthermore, it enables the service to process this data in a stream-based fashion. This leads to lesser memory consumption within the service instance, a higher throughput and instant response items of already finished data parts. These statements are evaluated against other approaches in Section 3.

To provide a formal foundation for the upcoming sections, let D be a data set with n data items d_i so that $D = (d_1, d_2, \ldots, d_n)$. Each data item d_i represents some kind of equally structured XML data. A given data set D also can be seen as an amount of single-item messages with an equally structured content. So the content of a single-item message corresponds to one data item d_i and data set D corresponds to the sum of all single-item messages sent to the service. This mapping provides a unified view of all data that emerges between a client and a service and allows for the usage of our stream-based approach in a consistent and a more general way.

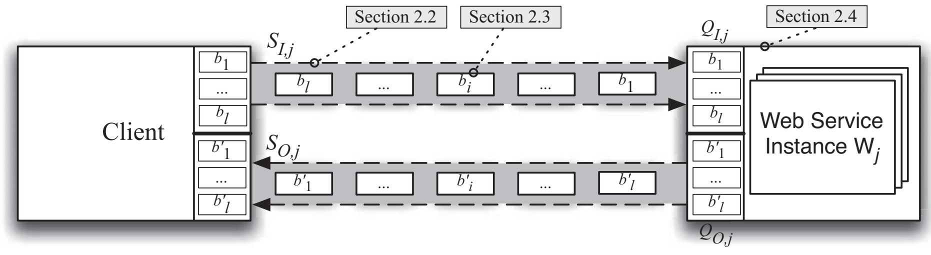


Figure 1: Invocation model with stream semantics.

2.1 Overview

The foundation of SOA is the *request–response paradigm*. Therefore, our stream-based Web Service invocation and its execution in this architecture should retain this fundamental building block. So to incorporate stream semantics into the Web Services invocation model and to preserve the fundamental paradigm, three adjustments have to be made. An overview of the adjustments described below is depicted in Figure 1 and includes the reference to the specific sections.

First we define a request message R containing data set D of size n as an input stream S_I for the service. Furthermore a response message R' returned by the service containing data set D' is defined as an output stream S_O. Since the input stream is limited to the size of D a stream is only established for the time D is transmitted. Hence it represents a request in a traditional point of view. Every request R_j with one separate D_j creates a new input stream $S_{I,j}$ and, depending on the service's functionality, an output stream $S_{O,j}$ (see Figure 1). This implies that every $\{S_{I,j}, S_{O,j}\}$ pair belongs to exactly one client request R_j and one service instance W_j processing R_j. Now we have two streams which preserve a common context for all data items in D.

Second, data set D has to be modified somehow to be processed by the service in a stream-based fashion. So data set D is divided into l processing buckets b_i with $B = (b_1, b_2, \ldots, b_l)$ that can be executed by the service in one step. The size and structure of all processing buckets B are predetermined by the service and have to be described in the service description (Section 2.3).

Third, in the traditional service invocation model the way a service processes the XML data has to be adjusted. Traditionally, one service instance W_j, which is instantiated by one service request R_j, can randomly access the whole data set D_j of its request. W_j builds D'_j as the content for R'_j until D_j is processed completely. To incorporate a stream-based data processing into the service execution, the service instance must be aware of an input queue where items $d_{i,j}$ arrive and an output queue where response items $d'_{i,j}$ have to be delivered to. Both queues work concurrently that means the instance can already deliver response items while still receiving request items. In the following subsections we will describe the three modifications in more detail.

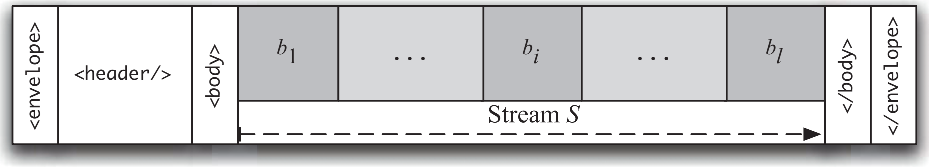


Figure 2: Stream-based SOAP message.

2.2 Stream Definition

The streams are realized by taking the SOAP specification. Remember a SOAP message consists of a header containing meta data and a body containing the user data that means the data set D. We structure the stream in the same way. The header comprises meta data about the stream itself and is used for negotiation. The body contains the data set distributed over l processing buckets b_i. Figure 2 depicts the structure of this stream. Actually, the whole stream of a request message R can be seen as stretched SOAP message.

To transfer R with stream semantics, the SOAP header is sent with all information that would normally be specified within the traditional SOAP header including all WS-* extensions. This opens the lower level HTTP connection. Since the whole message is already streamed using TCP/IP and HTTP, these lower level protocols are used as a channel to establish the stream and to stream structured XML content.

After establishing the input stream S_I, the starting SOAP `body` tag is transmitted. Now the client can send data set D with the help of processing buckets b_i, which actually are XML snippets of the data set D with a predefined structure and size (see Section 2.3). Note that the buckets are transmitted in a deeper XML hierarchy level than the already transfered `body` tag. When the client has finished sending D, it transmits the closing SOAP `body` tag, which resides on the same hierarchical level like the starting `body` tag, and the closing SOAP `envelope` tag. This triggers the disconnection of S_I.

The output stream S_O from the service to the client is established when the first response bucket b'_1 has to be sent. This behavior conforms to the traditional approach of sending the response message. The establishment of output stream S_O works in the same methodology as the input stream S_I. Since one processing bucket b_i can be processed by the service in one step, the response bucket b'_i can be streamed back immediately after successfully processing b_i and while still receiving further buckets. S_O is closed when the last bucket b'_l, the closing `body` tag and the closing `envelope` tag is sent. The advantage of sending the corresponding `body` and `envelope` tags as stream delimiter is that the whole stream itself forms one large SOAP message. Therefore it can be read by a traditional service, which buffers the whole XML stream and process the full DOM tree at once.

2.3 Bucket Definition and Description

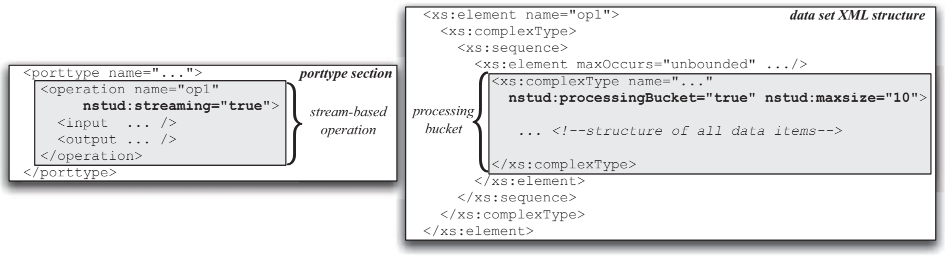
In this section, we define the structure of processing buckets in detail and how these processing buckets are described within the service description. As already mentioned in 2.1 we divide D into l processing buckets. This enables D to be processed in a stream-based fashion since one bucket defines the granularity of D, that is transmitted over the stream and to the service in one step. Furthermore the concept of processing bucket acts as an intermediate layer between D and its data items d_i and adds conceptional flexibility for further extensions shortly described in Section 4.

Remember, data set D represents an array of equally structured data items d_i. We define one bucket b_i containing m data items with $b_i = (d_1, \dots, d_m)$. Each bucket is simply a collection of single data items in pure XML notation without further information. So one bucket has the simple *textual* form of $b_i = (e_1, \dots, e_m)$ with e_i denotes the root element of data item d_i enclosing all its children elements.

The number of data items in each processing bucket can differ between a request bucket b_i and its response bucket b'_i. The most common assumption is, that every data item d_i generates exactly one response item d'_i. This describes a $N : N$ relationship where N data items generate N response items. If one data item generates a number of response items, a $1 : N$ relationship is reflected. An example is a service that returns all invoices for a given customer id as single response items. With both relationships, the response stream allows to stream back response items although not all request items have been processed completely.

A different case is the $N : 1$ relationship as it can be found in services implementing several aggregate functions that, e.g., computes the average value of all data items. Since the exact average value is computed with the arrival of the last data item d_n, the client has to wait for D to be processed completely. Thus no response items can be streamed back in between. But in certain scenarios, preliminary values may be already used for further processing or for a definition of stop criterions. To avoid this drawback, the application logic on service side can be extended to put preliminary values to the output stream. These preliminary values can be flagged with the help of an additional attribute, which can be easily added to the response item's XML structure within the service description.

The definition of processing buckets is predetermined by the service. Since the service provides the implementation to process the buckets, its definition must reside within the WSDL document of the service. The client uses this bucket definition to transmit the given data set accordingly. In addition, the WSDL must indicate the capability of the stream-based service. To describe these two parts, we augment the WSDL document in two sections. First, we add an attribute `streaming` in its own namespace to the `operation` element in the porttype section to indicate the stream-based processing capability of this operation. Second, we augment the XML Schema definition for the request message and the response message of this operation. Since data set D is an array of equally structured data items d_i, the XML Schema definition describes an unbounded set of complex elements containing the structure of every data item. To define processing buckets, we add an attribute `processingBuckets` to the element that forms the envelope for one data



```
<porttype name="...">
  <operation name="op1"
      nstud:streaming="true">
    <input  ... />
    <output ... />
  </operation>
</porttype>
```

```
<xs:element name="op1">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" .../>
        <xs:complexType name="..."
          nstud:processingBucket="true" nstud:maxsize="10">

          ... <!--structure of all data items-->

        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figure 3: WSDL extensions within porttype section and the data set XML structure.

item. Additionally, an attribute `maxsize` prescribes the maximum number of data items that can be bundled as one processing bucket. This attribute allows optimizing the data transfer for small data items. Figure 3 depicts the WSDL extensions for the `op1` operation to flag it for its stream-based capability and to define its processing buckets. The processing bucket definition for response items is not shown but is labeled in the same fashion as the request items.

2.4 XML Processing Model

For a stream-based XML processing of D on service side, the traditional XML processing model and hence the programming model have to be adapted. To access the XML document transmitted to the service, current approaches and implementations stores the whole XML message in a buffer and the Document Object Model (DOM) tree is built. A reference of this tree, that is, the reference of its root element is passed to the functional implementation which can randomly access all nodes of the document. The document D is processed and a response document D' is built. Only after processing D successfully, D' is returned to the framework. This execution model serves well in the context of the strict request–response and hence stepwise execution model. In the context of stream-based document processing three modifications have to be done to the application logic and to the framework hosting it.

As a first modification we introduce an input queue Q_I. Since we are not able to guarantee that all data items d_i can be processed in the speed the processing buckets arrive, the input queue buffers all processing buckets b_i and provides an interface for the application logic to access the application data within these buckets. This input queue Q_I is instantiated for every service instance W_j and is already depicted in Figure 1.

Second, we incorporate a cursor-like, forward-only XML processing model within the application logic. We assume that not all data items d_i fit into the service instances main memory at once, thus the incoming data items have to be accessible in a cursor-like, forward-only fashion and the application logic has to apply some window semantics to operate on all d_i. The application logic of W_j pulls every processing bucket out of the input queue $Q_{I,j}$ and processes it accordingly. Since a processing bucket consists of a list of data items in pure XML text, these data items can be parsed and a partial DOM-tree can be built representing all data items in one processing bucket.

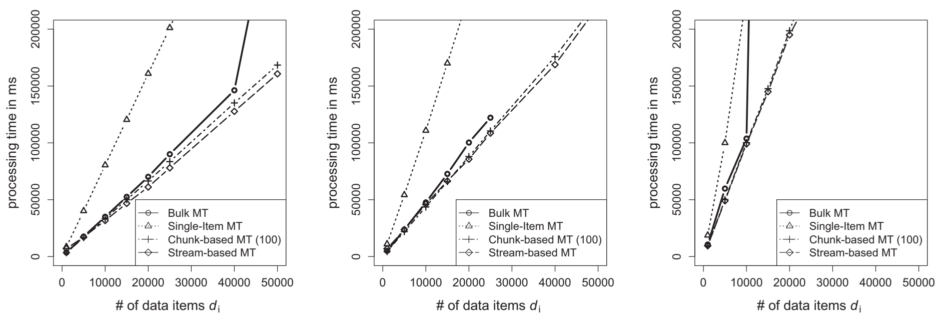
The third modification for our approach is the supply of an interface to place already processed data items $d'_i \in D'$ in output queue Q_O which is used by the hosting framework to send intermediate results over the output stream. This allows the framework for sending response buckets b' while still receiving the request. The framework bundles the items in Q_O into processing buckets according the service description. This output queue Q_O is instantiated for every service instance W_j and is already depicted in Figure 1.

3 Evaluation

In this section, we evaluate our stream-based Web service invocation and execution approach. We investigate how efficiently a large data set can be processed by a Web service. Assume a large data set $D = (d_1, d_2, \ldots, d_n)$ with thousands of equally structured business items d_i. Furthermore, assume a client, which wants this data set D to be processed by a Web service W. The client requires a certain number k of service invocations (requests) r_i to process the whole data set. We denote the set of all required invocations as R with $R = (r_1, r_2, \ldots, r_k)$.

We investigate four invocation models, that means four ways to process D with the Web service W. One of those four is our *stream-based message transfer* approach. We compare our approach with the three traditional invocation models *bulk message transfer*, *single-item message transfer* and *chunk-based message transfer*. *Bulk message transfer* is the straightforward way to process data set D with a Web service W. Utilizing *bulk message transfer*, we pack the whole data set D with size $s_D = \sum_1^n s_{d_i}$ into one request and send it to the service with one service call r so that $|R| = 1$. After receiving the whole request message r and thereby the whole data set D, the service W processes the complete data set. As result W creates a response message r' and sends it back to the caller. *Single-item message transfer* is the opposite way to process a data set D with a Web service W. In *single-item message transfer* we put every item $d_i \in D$ in one separate service request r_i so that $|R| = |D|$. After one item d_i has been processed by the service W and we have received the response, we send a new request containing the next item d_{i+1}. *Chunk-based message transfer* is the generalization of the two aforementioned models and was first formally considered in [SMWM06]. The main idea is to distribute the items d_i of the data set D to k different service requests r_j so that $|R| = k$ with $1 \leq k \leq |D|$. If the items are uniformly distributed, r_j contains $\frac{|D|}{k}$ data items of D. Obviously, *bulk message transfer* corresponds to *chunk-based message transfer* with $k = 1$ and *single-item message transfer* corresponds to *chunk-based message transfer* with $k = |D|$. The chunk size significantly influences the response time and can be tuned accordingly. For example, [GYSD08] proposes an approach to automatically optimize the chunk size.

To evaluate the performance behavior of the four invocation models, we conducted a series of experiments. In detail, we measured the response time of a Web service invoked with a



(a) One Client. (b) Two Clients. (c) Four Clients.

Figure 4: Total response time of different invocation models.

data set D for different numbers of concurrent invocations and different sizes of D. The service and the clients were hosted on a single machine with 2GB RAM. 512MB RAM have been assigned to the server as heap size for hosting this single service. Response times were measured on client side. All results are an average of 30 runs. For *chunk-based message transfer* we chose a chunk size of 100 data items. Other chunk sizes showed worse results.

Figure 4 illustrates the server's total response time, that means how long the client had to wait for all data items being processed. Figure 4(a), Figure 4(b) and Figure 4(c) show the results for one client, two concurrent clients and four concurrent clients, respectively. As we see, the number of concurrent clients influences the response time of the server by a constant factor independently of the used invocation model. Accordingly, we see similar results for each number of concurrent clients. *Single-item message transfer* has the worst total response time. Putting every single item in a single message leads to a significant overhead in message size and message parsing, which results in longer response times. The overhead arises because the message header is sent and parsed in total n times. *Bulk message transfer* and *stream-based message transfer* do not involve such overhead and, consequently, show better response times. Although *chunk-based message transfer* also leads to overhead, its overhead is significantly smaller and is compensated by advantages in message processing. *Bulk message transfer* shows the best performance of all invocation models for small data sets. For larger data sets, it is outperformed by *chunk-based message transfer* and our *stream-based message transfer* approach. *Stream-based message transfer* shows for large data sets the lowest total runtime of all invocation models.

Figure 4 also illustrates the scalability of the four invocation models. Of course, for all invocation models the response time increases linearly with the size of the data set D. But the increment differs from model to model. *Single-item message transfer* scales worst, since its overhead increase with the size of the data set D. *Bulk message transfer* scale better as long as size of D remains within the memory limit (40.000 data items in our experiment); close to the memory limit, the performance breaks down; beyond the memory limit, *bulk message transfer* is not applicable anymore. *Chunk-based message transfer* and *stream-based message transfer* scale similarly good as *bulk message transfer* does below the memory limit. But since both have a constant memory usage they scale also for data sets larger than the memory.

Aside from their performance behavior, the four invocation models differ in their functional properties. For many business scenarios, the application logic essentially requires to process data items within a common context. *Single-item message transfer* does not provide a common context since every item is processed separately. *Chunk-based message transfer* processes data items chunk-wise; therefore a common context exists for all items of one chunk. Nevertheless, many scenarios need a common context for all data items. *Bulk message transfer* and *stream-based message transfer* provide such a common context inherently and are in this respect not restricted to certain business scenarios.

To conclude, the evaluation shows that large data sets cannot be processed efficiently with common approaches. *Bulk message transfer* as the traditional invocation model maps directly to the request–response paradigm. It preserves one common context for all items of the data set it and enables the service to be implemented stateless. But it suffers from long latency, i.e., the time until a client receives the first response message, and bad scalability. In contrast, the invocation model *single-item message transfer* packs every data item into a single request message. This results in low latency, but it shows a worse performance and does not provide one common context. *Chunk-based message transfer*, as the generalized approach, shows the best performance if the optimal chunk size is chosen. But it has mediocre latency and lacks a common context for all data items. Our new *stream-based message transfer* approach proves to be superior to the traditional invocation models. With shortest total response time and the best scalability it shows the best performance [PVHL08]. It offers low latency, so the client receives first responses after a short time. Additionally, it is applicable to a wide range of use cases, because it inherently provides a common processing context for all data items.

4 Conclusion and Outlook

In this paper, we have extended the Web service environment to enable stream-based invocations and XML processing. Thereby, we utilize the existing protocols and specifications to lift the already existent streaming in low-level protocols up to the application layer. We also implemented our approach within an existing SOAP engine and evaluated the performance benefits in comparison to other common techniques. In general, our stream-based approach consists of (i) an extended service interface, (ii) an enhanced SOAP message interpretation with a stream bucket concept, and (iii) an appropriate XML processing model on application layer. From the evaluation, we can conclude that our stream-based approach decreases memory consumption and CPU usage for large scale data, while preserving full applicability.

However, the proposed data bucket concept is considered as a first step, because one bucket consists only of pure application data and no further information is annotated. This results in a compatible but not very robust stream management. A more sophisticated bucket design incorporating metadata about a bucket allow for more flexibility in bucket processing, more robust data transmission and fine-grained exception management. To annotate a bucket with metadata, the application data in one bucket can be enclosed with a bucket envelope in a separate namespace. This envelope comprises metadata like a unique bucket identifier for a request item and response item correlation or other information like performance hints or service status that the client framework can use to optimize its transfer. A fine-grained exception management can be achieved by replacing certain response items with warnings or exceptions that apply only to these items. All of this was not scope of this paper and is considered as future work.

5 Acknowledgements

The project was funded by means of the German Federal Ministry of Economy and Technology under the promotional reference ”'01MQ07012”'. The authors take the responsibility for the contents.

References

[CGB02] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *HPDC*, 2002.

[CYZ$^+$06] Shiping Chen, Bo Yan, John Zic, Ren Liu, and Alex Ng. Evaluation and Modeling of Web Services Performance. In *ICWS*, pages 437–444, 2006.

[DA03] K. Devaram and D. Andresen. SOAP Optimization via Parameterized Client-side Caching. In *Parallel and Distributed Computing and Systems*. ACTA Press, 2003.

[GYSD08] Anastasios Gounaris, Christos Yfoulis, Rizos Sakellariou, and Marios Dikaiakos. Robust runtime optimization of data transfer in queries over WS. In *ICDE*, 2008.

[HPL$^+$07] Dirk Habich, Steffen Preissler, Wolfgang Lehner, Sebastian Richly, Uwe Aßmann, Mike Grasselt, and Albert Maier. Data-Grey-Box Web Services in Data-Centric Environments. In *ICWS*, pages 976–983, 2007.

[MF05] Ana C. C. Machado and Carlos A. G. Ferraz. Guidelines for performance evaluation of web services. In *WebMedia*, pages 1–10. ACM Press, 2005.

[OAS06] OASIS. OASIS Reference Model for Service Oriented Architecture 1.0, 2006. http://www.oasis-open.org/specs/index.php#soa-rmv1.0.

[PVHL08] Steffen Preissler, Hannes Voigt, Dirk Habich, and Wolfgang Lehner. Data Stream Semantics in Service-Oriented Environments. Technical report, Technische Universität Dresden, 2008. http://wwwdb.inf.tu-dresden.de/files/publications/PVHL08.pdf.

[SMWM06] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query Optimization over Web Services. In *VLDB*, pages 355–366, 2006.

[vE03] Robert van Engelen. Pushing the SOAP Envelope with Web Services for Scientific Computing. In *ICWS*, pages 346–352, 2003.

[W3C02] World Wide Web Consortium W3C. Web Service specifications, 2002. http://www.w3.org/2002/ws/.

Dissertationspreise