Remote Batch Invocation for Web Services

Document-Oriented Web Services with Object-Oriented Interfaces

Ali Ibrahim
Department of Computer Science
University of Texas at Austin
aibrahim@cs.utexas.edu

Yang Jiao
Computer Science Department
Virginia Tech
jiaoyang@cs.vt.edu

Marc Fisher II
Computer Science Department
Virginia Tech
fisherii@cs.vt.edu

William R. Cook
Department of Computer Science
University of Texas at Austin
wcook@cs.utexas.edu

Eli Tilevich
Computer Science Department
Virginia Tech
tilevich@cs.vt.edu

Abstract

The Web Service Description Language defines a service as a procedure whose inputs and outputs are arbitrarily structured data values, sometimes called *documents*. In this paper we argue that document-oriented interfaces can be viewed as batches of calls to finer-grained object-oriented interfaces. Turning this correspondence around, we show that flexible documents can be specified by converting a block of fine-grained object-oriented invocations into a batch document. The statements in the block operate directly on virtual service objects, freeing the programmer from the need to explicitly construct invocation objects and then manually correlate them to results in the response. Batch blocks can also include conditionals and loops. Our system, Remote Batch Invocation for Web Services, translates object-oriented interfaces into a WSDL describing batches of calls. The WSDL can be used by standard web service clients, but a wrapper library simplifies client invocation in existing languages. Extending a language with support for remote batches provides a fully integrated client. The result is a powerful infrastructure for web services that directly connects to standard object-oriented interfaces, with tool support to automatically create and decode documents representing sequences of method invocations. We have used our infrastructure to create a Web server wrapper for the Amazon Associates Web service, which shows that remote batches can support a clean object-oriented programming model over a stateless web service.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; D.1.3 [*Programming Techniques*]: Distributed Programming

General Terms Languages, Design

Keywords distributed computing, Web services, document-oriented computing, middleware, service-oriented architecture, language design

1. Introduction

A *web service* is a remote invocation in which the world wide web is used as the transport protocol. Although some web services resemble traditional Remote Procedure Calls (Winer 1999), document-oriented services (Christensen et al. 2001; Papazoglou et al. 2007) and representation state transfer (REST) (Fielding and Taylor 2000) are becoming more prevalent. In this paper, we focus on document-oriented web services that use the Simple Object Access Protocol (SOAP) to send and receive arbitrarily structured (XML) documents (Box et al. 2002).

The document-oriented approach is flexible; documents can represent complex objects (e.g., purchase orders, medical information), complex actions (e.g., creation, multiple updates, bulk removal, or specialized operations), queries, or combinations of the preceding. Despite this complexity, there is no standard methodology for detailed design of service documents. Some support is found in Fowler's distribution patterns, *Remote Façade* and *Data Transfer Object* (Fowler 2002), which improve performance at the expense of clean design.

> "[With Remote Façade] you give up the clear intention and fine-grained control you get with small objects and small methods. Programming becomes more difficult and your productivity slows." (Fowler 2002)

"In many ways, a Data Transfer Object is one of those objects our mothers told us never to write." (Fowler 2002)

Some web services allow multiple actions to be grouped together into a single input document to improve performance. However they usually do not support even simple dependency between actions, for example, when the result of the first action is a parameter to the second action. The client must also decode compound results and associate them to the original actions.

The desire to improve performance, results in the system designer specifying more complex input and output documents. While a simple document may naturally correspond to making a remote procedure call, programming with such complex documents is more indirect. Rather than performing an action on an object, the client must instead create an object describing an action and send it to the server for interpretation. This indirection complicates client programming and also introduces a new category of design issues that are not addressed by standard object-oriented design methodologies.

At the performance level, remote procedure calls are not *latency compositional*. By this we mean that performing two remote calls `f(g(x), y)` incurs the cost of two round trips, while a specialized procedure `fg(x, y)` could perform the same function with only one round trip. Asynchrony is another way to avoid latency costs, but it does not help in this case because the second call cannot be made until the first call returns its value (Liskov and Shrira 1988). The Remote Façade pattern requires anticipating all possible client compositions and building them as separate entry points, thereby imposing an unreasonable burden on the programmer. A cleaner approach to composing web service operations would enable greater flexibility and maintainability, without requiring service designers to anticipate all possible client interactions.

Remote Batch Invocation for Web Services (RBI-WS) is a new approach to document-oriented web services supporting fine-grained object-oriented interfaces. RBI-WS allows complex interfaces to server objects to be converted into a web service interface whose instances describe batches of calls to the server objects.

We demonstrate how RBI-WS can enable the creation of more flexible and composable web service interfaces with two automated programming tools. A server tool translates a set of interfaces to a web service definition language (WSDL) specification and provides a generic web service capable of interpreting messages that conform to that specification. A client tool, a source-to-source translator, converts Java code with the `batch` statement to plain Java which uses Batch Execution Service and Translation (BEST), our middleware library for batched execution using SOAP. In addition to the client tool we provide, users may connect to the batched web service using their favorite SOAP library and the WSDL generated by the server tool.

We demonstrate our language extension by looking at an existing web service and using remote batch invocation to implement various client use cases. Our demonstration highlights the expressiveness and flexibility of using RBI-WS to build web services, while indicating that it could also provide performance and code size reduction advantages.

We have previously designed a language extension, Remote Batch Invocation (RBI), for use with remote method invocation (RMI) for Java (Ibrahim et al. 2009). Thus the main contribution of this paper is not the idea of batching for remote services, but the exploration of three new ideas:

1. A discussion of the relationship between document-oriented web services and traditional object-oriented interfaces.
2. An algorithm for translating a set of object-oriented interfaces into input and output documents that support different forms of compositionality.
3. An application of RBI-WS to a real-world web service and a study of its usefulness.

2. Documents and Interfaces

One might wonder whether it is reasonable to consider collections of operations as documents? Conversely, is it reasonable for documents to be understood as batches of operations? To address these questions, we first consider what a "document" is and how it is used. The interface of a web service defines a *language* of legal inputs. A document is a sentence in this language. The key point is that the server must *interpret* the input document to determine which actions it should perform. If the document represents a purchase order, then the various parts of the document are interpreted to specify parts of the order and perform the requisite actions on the server. Thus a document can be viewed as a sentence of a domain-specific language specialized to represent calls to a particular API. The language defines how the calls may be composed and what output the client expects. The separation of the language into API calls and composition operators is subjective; one can view the composition operators as part of the API. We will consider control flow operators and naming services as external to the API. Unlike a general programming language, most domain-specific languages associated with web services allow only limited composition such as, perhaps, sequencing of unrelated API calls.

2.1 Documents as Collections of Calls

To illustrate the correspondence between documents and calls on an API, we will examine a couple of real-world input documents. The code listing in Figure 1 shows a sample input document a client could send to the Amazon Associates Web Service (AWS). We will look at this web service in more detail in Section 5. This request selects two Amazon items by their ASIN ids and requests their Amazon sales ranks and images. Although not shown here, there may be multiple lookup requests as well as multiple operations in a single input document.

One can imagine how this interface would look if it were designed as a set of fine-grained local object-oriented interfaces. There possible set of interfaces is given in Figure 2.

```
<ItemLookup>
<AWSAccessKeyId>XYZ</AWSAccessKeyId>
<Request>
  <ItemIds>
    <ItemId>1</ItemId>
    <ItemId>2</ItemId>
  </ItemIds>
  <IdType>ASIN</ItemIdType>
  <ResponseGroup>SalesRank</ResponseGroup>
  <ResponseGroup>Images</ResponseGroup>
</Request>
</ItemLookup>
```

Figure 1: Example request document for AWS

```
interface Amazon {
  void login(String awsAccessKey) ;
  Item getItem(String ASIN);
  ...
}
interface Item {
  int getSalesRank();
  Image getSmallImage();
  ...
}
// calls specified in document
aws.login("XYZ");
Item a = aws.getItem("1");
Item b = aws.getItem("2");
return new Object[] {
  a.getSalesRank(), a.getSmallImage(),
  b.getSalesRank(), b.getSmallImage() }
```

Figure 2: Interfaces and calls to represent Figure 1

The document in Figure 1 can be viewed as a script that specifies a sequence of calls to the corresponding methods in the interfaces in Figure 2. The `ItemId` tags represent calls to `getItem` and the `ResponseGroup` tags specify which accessors to invoke on each item that is located. The interpretation of the document is fairly complex, and the result is also a complex structure whose form depends upon the input document. The input document is in effect a kind of query.

Documents which specify updates can also be naturally supported by a set of fine-grained interfaces. The XML document in Figure 3 shows a request to modify an Amazon shopping cart. The interfaces in Figure 4 are one way to capture the fine-grained server methods that this document might invoke.

We believe that many service documents can be understood as specifying a pattern of calls to fine-grained server objects. As web service interfaces become more sophisticated (e.g. Amazon Associates Web Service), the documents begin to resemble scripts in a small specialized programming language.

```
<CartModify>
  <AWSAccessKeyId>ABC</AWSAccessKeyId>
  <Request><CartId>0</CartId>
    <HMAC>XYZ</HMAC>
    <Items>
      <Item>
        <Action>MoveToCart</Action>
        <CartItemId>0</CartItemId>
        <Quantity>1</Quantity>
      </Item>
      <Item>
        <Action>SaveForLater</Action>
        <CartItemId>1</CartItemId>
      </Item>
    </Items>
  </Request>
</CartModify>
```

Figure 3: Sample AWS update request document

```
interface Amazon {
  void login(String awsAccessKeyId) ;
  Cart getCart(String cartId, String HMAC);
  ...
}
interface Cart {
  void moveToCart(CartItem item, int quantity);
  void saveForLater(CartItem item);
  CartItem getCartItem(CartItemId itemId) ;
  ...
}
// calls specified in document
aws.login("ABC");
Cart cart = aws.getCart("0","XYZ");
cart.moveToCart(cart.getCartItem(0), 1);
cart.saveForLater(cart.getCartItem(1));
```

Figure 4: Interfaces and calls to represent Figure 3

2.2 Blocks with Control Flow as Documents

If we can think of documents as representing a set of calls on object-oriented interfaces, it is natural to think of the reverse correspondence. Given a set of object-oriented interfaces, can we produce a XML schema which describes documents used to encode blocks of operations on those interfaces? We would like this domain specific language to have the following properties:

1. The language should allow the clients to specify batches of method invocations with support for let assignment, conditionals, loops, and exceptions. These constructs allow flexible composition of operations on the remote interfaces.
2. The language should specify as much information as possible about the set of interfaces including type information. This design allows the web service definition lan-

$$
\begin{aligned}
n &\in Name \\
l &\in Variable \\
c &\in C : \texttt{Boolean} + \texttt{Number} + \texttt{String} \\
&\quad + \texttt{Collection[C]} \\
binop &\in \{+, -, *, /, \vee, \wedge, >, =\} \\
unop &\in \{-, not\} \\
E &= \texttt{root} \mid l \mid E\ binop\ E \mid unop\ E \mid c \\
&\quad \mid E.m_1(E_1, \ldots, E_{j1}) \mid \ \ldots \ \mid E.m_n(E_1, \ldots, E_{jn}) \\
S &= E \\
&\quad \mid S_1 ; S_2 \\
&\quad \mid \texttt{let}\ l = E\ \texttt{in}\ S \\
&\quad \mid \texttt{let*}\ l = E\ \texttt{in}\ S \\
&\quad \mid \texttt{if}\ S_1\ S_2\ S_3 \\
&\quad \mid \texttt{for}\ (v \in E)\ S
\end{aligned}
$$

Figure 5: Domain Specific Language for Web Services

guage description to provide as much information as possible about the service to clients.

3. The XML schema corresponding to the language should produce a reasonable set of DTO's when given to standard code generator tools such as Axis and Microsoft Visual Studio. This property is a practical consideration given we would like programmers to use our domain specific language without the need for special tools.
4. The XML schema corresponding to the language should be human readable.

The last two points will be addressed in the next subsection. Figure 5 shows the structure of the family of languages we chose to represent document-oriented web services.

The language contains basic control structures for sequencing, naming, branching, and looping. The `let*` construct behaves similar to the normal binding `let` construct, but additionally marks that binding as required at the client. The keyword `root` represents the root service object. The language is statically typed. Although not shown in the figure, the primitive types of this language are booleans, integers, doubles, and strings. The language includes some simple arithmetic and logical operators to operate over the primitive types. The method calls $m_1 \ldots m_n$ represent the interface methods.

What is interesting about this family of DSL is that they are limited compared to general purpose languages. There are no abstraction mechanisms and no module system. Since we see these languages as glue languages, this seems reasonable. There are a couple of other interesting design choices. The language does not provide a natural way to perform aggregation, although it can be done if the service API helps out. There is only one numeric type instead of the standard numeric types defined in SOAP web services. Both these design choices are ones we plan on revisiting after more case study evaluations. Another interesting omission is the lack of constructors. Instead the language relies on the web service providing factory methods for constructing objects of interest.

One issue we are avoiding in this paper is security, since we feel it is mostly orthogonal to the idea of web service documents as batching. Our current implementation allows a client to invoke any method on any object that is *reachable* from the root service object. An object is reachable if it is the return value of a method on another reachable object. There are many ideas for limiting accessibility further such as limiting the methods which can be invoked or explicitly defining which set of objects can be manipulated by clients. Our language may also make it easier to execute denial of service attacks on the web service because of the ability to use loops. One simple approach may be to limit the number of *steps* that a batch executes. A single step may be defined as one reduction in the operational semantics of the language.

2.3 Encoding Interfaces as XML Schema

There are many possible encodings of a web service DSL into an XML schema. The specification of the language is constrained by the abilities of XML schemas. For example, XML schemas have support for single inheritance, but no support for multiple inheritance or parameterized types. We are also restricted by the capabilities of existing WSDL code generation tools. For example, an early iteration of the encoding used substitution groups which allow transparent polymorphism, i.e. documents do not need to specify a type attribute indicating the actual type of the element. Unfortunately, neither of our WSDL code generation tools (Axis 2 and Microsoft Visual Studio) could recognize polymorphism expressed in this fashion.

Our final encoding can be divided into a generic part which is the same for all sets of interfaces and a specific part which is unique for a set of interfaces. The general structure of our XML schema encoding is to represent each interface and basic type with a schema type. Every expression encoding enforces the type of its sub-expressions and itself extends the schema type which corresponds to its own type. This encoding enforces type safety and provides the client with type information about the web service.

The generic schema is shared among all web services and describes naming, control flow, basic data types, exceptions, and output format. Appendix A contains part of the generic schema; some repetitive sections have been removed for brevity. The *Operation* type is the base type for all types. Objects are identified by *handles*, which represent the identity of an object or value that exists on the server during execution of the batch. The XML attribute *binding* in the batch operation tag represents the name of the handle for the result of that operation. The XML attribute *neededLocally* in the batch operation tag allow the programmer to specify whether the value is needed by the client. The *Any* type is the base type for all value types, i.e. all types except for *Void*. Control flow structures provided are sequences, conditionals, and loops. The basic data types include numbers, booleans, and strings. Exceptions contain name and message strings. The output document is an untyped map from strings to values of the *Any* type.

The batch contains a series of calls to various objects. The sub-elements of a call represent the arguments to the call, which can be other calls or handles of previously defined objects. The target of an object-oriented method invocation is the first sub-element and is named *this*. An exception to this rule, is that method invocations on the root object do not specify the *this* sub-element.

```
<batch type="Sequence">
 <step type="AmazonService__getItem" binding="a">
  <p1 type="String" value="1111" />
 </step>
 <step type="AmazonService__getCart" binding="b">
  <p1 type="String" value="222" />
  <p2 type="String" value="xxx" />
 </step>
 <step type="Cart_add">
  <p1 type="Item__Ref" ref="a" />
  <p2 type="Number" value="1" />
 </step>
</batch>
```

For example, the *AmazonService__getItem* tag represents a call to the getItem method. Its target is the root object, so the *this* sub-element is omitted. It has one parameter, an item id that specifies which item to retrieve The binding attribute defines the handle for the method return value, in this case ”a”. Return handles are optional, but they are useful even if the method returns void; the handle is later used to identify any exceptions that might return from the call.

Figure 6 shows a formal description of how a set of interfaces are translated into the web service specific schema. The syntax used for XML schema is the compact representation XSCS (Wilde and Stillhard 2003). To simplify the presentation, we assume that all interfaces extend an imaginary type TOP and that the parent type of all interfaces is explicitly given.

First, we define a pair of helper functions: $lookupType$ and $lookupParent$. The $lookupType$ function takes a type and a set of interfaces and returns the corresponding type schema type. If the type is a primitive type, then $lookupType$ returns the name of a standard XML schema type defined independently. If the type is one of the interfaces being translated, $lookupType$ returns the name of the interface. Otherwise, the return value is undefined. The $lookupParent$ function takes a type and returns the first ancestor type that is present in the set of interfaces being translated. If no such ancestor exists, then $lookupParent$ returns *batch:Any* as the parent type.

The translation for a set of interfaces is simply the concatenation of translation for each interface. The translation for each interface produces five schema types related to the type of the interface.

- A schema type which represents the interface type.
- A schema type which represents references to objects with interface type.
- A schema type which represents the type of the collection whose elements have the interface type.
- A schema type which represents references to objects that are collections whose elements have the interface type.
- A schema type which represents collections of instances of the interface type.

In addition, a schema type is produced for each method in the interface. Each method schema type extends the schema type corresponding to the method return type. This schema type has child elements for the object to invoke the method on and for each parameter. If the interface is a root type, then the object to invoke the method on is optional and if omitted denotes that the method should be invoked on the root service object.

3. Batch Service Servers

So far we have presented an algorithm for translating a set of object-oriented interfaces to a XML schema that describes a custom DSL for operating over those interfaces. However, we would like for programmers not to have to write a custom interpreter for each web service. To that end, we have implemented a generic interpreter which can be deployed as an Axis 2 web service. Axis 2 is a popular open source web service engine that supports SOAP web services. The programmer supplies our generic interpreter with two pieces of information: a web service definition language (WSDL) XML document describing the web service and the name of a class implementing the root interface for the web service.

The programmer constructs the WSDL by running a custom Java to WSDL tool. This tool implements the translation algorithm in Figure 6. The translation algorithm along with the generic batch XML schema provide the definitions of the batch input and output documents. The rest of the WSDL defines an executeBatch operation which interprets batch input documents and produces batch output documents.

The programmer also specifies the class that implements the root web service interface. This class must have a default constructor which will be used to create the root service object. The root service object persists for the lifetime of a batch execution.

The generic interpreter web service can then be deployed as a normal Axis 2 service that appropriately interprets batch requests and delegates method calls to the appropriate objects.

4. Batch Service Clients

In order for the programmer to gain practical benefits from using RBI-WS, convenient client bindings must be provided. What makes the creation of such bindings nontrivial is that Web services are a language-independent communication infrastructure, and the same service may need to be invoked by multiple clients written in different languages. One ad-

$$\begin{aligned}
n, p, a &\in String \\
x \in \text{Java Base Types} &: \texttt{int/Integer, short/Short, long/Long, float/Float} \\
&\quad \texttt{double/Double, boolean/Boolean, String, void/Void} \\
&\quad \texttt{TOP} \\
I &: \texttt{interface}\ name\ \texttt{extends}\ parent\ \{m_1 \ldots m_j\} \\
T &: I + X \\
m \in M &: return\ name(param_1\ formal_1, \ldots, param_k\ formal_k) \\
s &\in P(I)
\end{aligned}$$

$$lookupType(T, s) = \begin{cases} name(I) & I = T \land I \in s \\ \texttt{batch:String} & T = \texttt{String} \\ \texttt{batch:Number} & boxed(T) <: \texttt{Number} \\ \texttt{batch:Boolean} & boxed(T) = \texttt{Boolean} \\ \texttt{batch:Void} & boxed(T) = \texttt{Void} \end{cases}$$

$$lookupParent(I, s) = \begin{cases} \texttt{batch:Any} & parent(I) = \texttt{TOP} \\ parent(I) & name(J) = parent(I) \land J \in s \\ lookupParent(J, s) & name(J) = parent(I) \land \notin s \land I \neq \texttt{TOP} \end{cases}$$

$Translate(I, isRoot, s) \rightarrow$

```
abstract complexType $name(I)$ extends $lookupParent(I,s)$ {}

complexType $name(I)$_Ref extends $name(I)$ {
  attribute val { boolean }
}

$\forall m \in \{m_1(I), \ldots, m_j(I)\}$
abstract complexType $name(i)$_$name(m)$ extends
$lookupType(return(m), s)$ {
  (
    this { $name(I)$ } $if\ isRoot$ [0,1],
    p1 { $lookupType(param_1(m), s)$ }
     , ..., pk { $lookupType(param_j(m), s)$ }
  )
}
abstract complexType Collection_$name(I)$ {}

complexType Collection_$name(I)$_Ref extends Collection_$name(I)$
  { attribute val { boolean }  }

complexType Collection_$name(I)$_Value extends Collection_$name(I)$
  { (item { $name(I)$ } [0,] )  }
```

Figure 6: Interface to XML Schema Translation for Batches

```
...
BatchExecutorStub best =
  new BatchExecutorStub(ENDPOINT);
Batch batch = new Batch();
StringValue id = new StringValue();
id.setVal("1");
StringValue name = new StringValue();
name.setVal("John Smith");
AWSE_searchItem search = new AWSE_searchItem();
search.setNeededLocally(true);
search.setBinding("x");
search.setP0(id);
search.setP1(name);
Item__Ref ref = new Item__Ref();
ref.setRef("x");
Item_getName getName = new Item_getName();
getName.set_this(ref);
getName.setNeededLocally(true);
getName.setBinding("y");
Sequence seq = new Sequence();
seq.setStep(new Operation[] {search,getName});
batch.setOp(seq);

Output out = best.executeBatch(batch);
for (OutputBinding binding : out.getBinding()) {
  // output the result
}
```

Figure 7: Code listing for batch client using Axis 2 WSDL2Java generated interface.

vantage of RBI-WS is that its WSDL can be processed by any standard WSDL client binding generator, making our approach applicable for any SOAP-enabled client. Although such default RBI-WS client bindings can be used out-of-the-box, we have also experimented with two different approaches for streamlining RBI-WS client programming. The first approach extends Java with a new keyword, `batch`. Since not all client environments lend themselves for extending their host language, our second approach uses a middleware platform called Batch Remote Method Invocation (BRMI). In the following discussion, we first outline the default client bindings. Then we explain how we have adapted the batch extension and the BRMI middleware, detailed in prior publications (Tilevich et al. 2009; Ibrahim et al. 2009), for the needs of RBI-WS.

4.1 Default Clients

The WSDL created by the interface translation can be imported by standard web service clients, including Apache Axis 2 and Microsoft Visual Studio^{TM}.

The client code fragment in Figure 4.1 connects to the Amazon web service, looks up a merchandise item by its name and id, and returns the results back to the client.

Unfortunately, the code in Figure 4.1 obscures the programmer's intentions. The reason for poor readability is that this code creates abstract syntax objects, which represent the sequence of calls given in Section 2.1, and as such it is indirect and reflective. In fact, this code is even somewhat more complex than the code for invoking the standard Amazon Web service. One reason for the extra complexity is that the code uses separate steps to construct the parameters and set their values, before passing the parameters to service methods. To hide some of this complexity, we next describe two approaches that can smooth away the rough edges of the default RBI-WS client bindings.

```
BatchClient amazonService
  = new BatchClient(SERVER);
batch(AmazonService service : amazonService) {
  final Item x = service.searchItem("1", "John Smith");
  final String y = x.getName();
    // output result
}
```

Figure 8: Code listing for batch client using batch language extension.

4.2 The Batch Extension

One approach we took is to extend the Java language with new syntax that supports defining a batch with a mix of local and remote operations (Ibrahim et al. 2009). The programmer generates the interfaces to a web service by running a custom tool which takes a WSDL and reverses the translation in Figure 6. Using those interfaces, the programmer can invoke remote operations on remote objects inside the batch block. An object is remote if it is the root remote service object or if it is obtained from a remote operation. The compiler separates the remote and local operations producing a partitioned program that may have been difficult for the programmer to write by hand because of interactions between the remote control flow and the remote-to-local dataflow. At runtime the remote operations are executed as a single batch and the results are threaded into the local operations as needed. Figure 8 shows how to rewrite the example in Figure 4.1 using the batch syntax. Using the batch syntax, the programmer intent is clearer and the code is more succint.

Remote Batch Invocation uses the following syntax:

```
batch (Type Identifier : Expression) Block
```

The *Identifier* specifies the name of the root remote object. The *Expression* specifies the service which will provide the root remote object. The *Block* specifies both remote and local operations. A remote operation is an expression or statement executed on the server. All remote operations inside the batch block are executed in sequence followed by the local operations in sequence. A single remote call is made which contains all of the remote operations. This is the key property, as it provides a strong performance model to the programmer albeit lexically scoped (Gabriel 1992). Exceptions in a remote operation are re-thrown in the local operation sequence at the original location of the remote operation. If the remote operations fail due to a network error, then an exception is thrown before any of the local operations execute. Operations inside the batch block are reordered and it is possible that the block executes differently as a batch than it normally would. The compiler does try to identify some of these cases and warn the programmer; however, it is up to the programmer to be aware of the different Java semantics inside the batch block.

The compiler partitions operations inside the batch block by marking them as *local* or *remote*. Remote expressions execute on the server, possibly with input from static local expressions. Local expressions execute on the client, possibly with output from remote expressions.

Remote Batch Invocation does not support remoting of many Java constructs, including casts, `while` loops, `for` loops, remote assignments, constructor calls, etc. Although some of these constructs can be used inside the `batch` block, they will be executed locally. If using these constructs would interfere with the remote batch execution, the batch translator will raise an error. Future work may relax some of these restrictions. If remote assignments were allowed, then it would be possible to aggregate (e.g. sum or average) over collections remotely (we currently have an alternative solution to this problem). General loops could also be supported without significant changes to the model.

Exceptions are a special case. The remote batch cannot catch exceptions remotely, but it does propagate them to the client in the original location of the remote operation that produced the exception. In this way, the client can catch exceptions raised remotely and handle them locally.

In our previous work, the `batch` keyword was implemented for Java Remote Method Invocation (RMI) which is more powerful in many ways than SOAP web services. For example, our implementation of batching for RMI supported sending any Java object which implements the `Serializable` interface to and from the server. On the other hand, SOAP web services only allow defined record-like types and certain primitive types to be transferred. We did think of allowing the transfer of Java types which are just data-holders (sometimes called beans, DTOs, or value objects), but we decided against it, since the web service can easily provide methods to construct the data-holder on the server directly. For RBI-WS, we modified the compiler to restrict input and output to the batch to primitive types and strings.

4.3 The BRMI Middleware Library

Batch Remote Method Invocation (Tilevich et al. 2009) was originally created as a middleware mechanism for optimizing Java RMI. We are in the process of implementing our BRMI library for RBI-WS in C#.

BRMI provides a library for expressing a collection of remote methods that are recorded on the client, sent to the server in bulk, executed by the server, and return results to the client in bulk. BRMI also provides facilities for expressing conditionals and loops. A key concept of BRMI is a batch interface, in which all methods return either a `Future` object or another batch interface. A `Future` is a placeholder for a result of a remote method. By returning futures, batch interface methods can be used for recording a sequence of remote invocations. An explicit call to the `flush` method launches the execution of a batch. After this method is called, the future objects can be queried for the actual values they hold.

As a specific example, a batch interface for class `Item` of the Amazon service can be specified in C# as follows:

```
namespace BatchWS
{
    public interface Item : Batch
    {
        ...
        Future<boolean> isAvailable();
        Future<string> getTitle();
    }
}
```

A client can use this batch interface to express that a collection of remote methods be invoked in bulk on the server as follows:

```
...
Item item = amazon.getItem(asin);
...
Future<bool> res = amazon.rIf(item.isAvailable());
Future<string> title = item.getTitle();
amazon.rEnd();
amazon.flush();
if (res.get())
  System.Console.WriteLine(title.get(),
                        " is available");
...
```

The code snippet above uses the root batch interface to obtain an `Item` batch interface for a particular item. Then it checks whether the item is available, and if this is the case, obtains the item's title. All the remote invocations are performed in one batch, including the remote conditional. Because C# is not an extensible language, BRMI provides conditional constructs as library calls–`rIf` and `rEnd`.

To integrate BRMI with RBI-WS, our infrastructure will provide a generator taking a WSDL interface as input and creating a corresponding version of C# batch interfaces. A call made through a batch interface will be forwarded to a module encoding all batch calls into a RMI-WS SOAP request. When the SOAP request returns, the same module decodes the results and updates the future objects involved.

The translation module will be general and will work with any RBI-WS WSDL/batch interfaces. Because BRMI provides a natural object-oriented interface for batching remote invocations, it exemplifies how a library-based approach can be leveraged for streamlining client-side RMI-WS programming. Even though our prototype implementation is C# specific, we envision that a BRMI interface can be easily provided in most object-oriented languages.

5. Case Study: Amazon Associates Web Service

To gain insight into how real-world web services are actually implemented and used, we examined the Amazon Associates Web Service (AWS).[1] AWS is primarily intended for individuals who want to earn product referral fees by providing links to Amazon products on their web sites. AWS includes operations for browsing the Amazon catalog, searching for products sold by Amazon and Amazon marketplace sellers, looking up product and seller information, and managing a shopping cart.

The left side of Figure 11 shows a typical sequence of calls to AWS. This sequence corresponds to two calls to AWS. The first call (lines 7-13) performs a search for books about dogs. Lines 14-21 output all of the offers for the found books. The user is then assumed to select one of these offers to purchase (lines 24-25). A new shopping cart is then created containing one copy of the selected item (lines 29-41), and the cart contents, total price, and a link to complete the purchase are displayed to the user (lines 44-52).

5.1 A Batched Amazon Web Service

The first goal of our case study is to determine if we can build an efficient batched web service that provides similar functionality to AWS. Therefore we have prototyped a batch web service based on AWS. To prototype the service rapidly, we created a set of server object classes that access the existing Amazon web service. Figure 10 shows the architecture of our service.

Our *Batched Amazon Server* consists of three main components. The *JAX-WS Amazon Client Library* is a set of classes generated using Sun's wsimport tool. This tool takes as input a WSDL file describing a web service and produces a set of classes for accessing the web service. We then build a set of *Server Object Classes* on top of the library. This set of 12 Java classes provides an object-oriented interface to the product search, browse node hierarchy, seller information, and shopping cart functionality of the Amazon Associates web service. This functionality corresponds to 8 of the 22 operations defined by the Amazon Associates web service. Figure 9 gives interfaces for five of the classes central to the item search and shopping cart functionality provided by our service.

The right side of Figure 11 shows the same shopping sequence as the left side, but implemented for the Batched Amazon Server using the batch keyword. Of particular interest is the portion of the code responsible for adding the item to the cart (lines 29-41). For the batched version of the API, only three statements are required as compared to the Amazon version of the API which requires nine statements. These savings come from removing the need to create a document describing the cart operation, and shows the type of advantage that can be gained from using our batched API even for simple cases.

[1] `http://aws.amazon.com/associates/`

```
interface AmazonService {
  void login(String awsAccessKey) ;
  Cart createCart(Offer offer, int quantity) ;
  Cart getCart(String cartId, String HMAC) ;
  SearchCriteria createSearchCriteria() ;
  Item[] search(SearchCriteria criteria) ;
  Item getItem(String ASIN);
  BrowseNode getBrowseNode(String browseNodeId) ;
  Offer getOffer(String ASIN, String offerListingId) ;
}

interface Cart {
  String getCartId() ;
  String getHMAC() ;
  CartItem[] getCartItems() ;
  CartItem[] getSavedForLaterItems() ;
  Price getSubTotal() ;
  String getPurchaseURL();
  void add(Offer offer, int quantity) ;
  void clear() ;
  void remove(CartItem item) ;
  void moveToCart(CartItem item, int quantity) ;
  void saveForLater(CartItem item) ;
}

interface CartItem {
  String getCartItemId() ;
  Item getItem() ;
  int getQuantity() ;
  Price getItemTotal() ;
  void setQuantity(int quantity) ;
}

interface Item {
  String getASIN();
  String getDetailPageURL();
  String getSalesRank();
  Image getSmallImage();
  Image getLargeImage();
  Offer[] getOffers();
  String getTitle();
  BrowseNode[] getBrowseNodes();
}

interface Offer {
  Seller getSeller();
  Item getItem() ;
  String getOfferListingId();
  Price getPrice();
  String getAvailability();
  String getQuantity();
  boolean isEligibleForSuperSaverShipping();
  boolean isEligibleForPrime();
}
```

Figure 9: Amazon Fine-Grained Interfaces

Java using JAX-WS and standard Amazon Associates API

```java
void shoppingSequence() {
  AWSECommerceService service
      = new AWSECommerceService() ;
  AWSECommerceServicePortType port
      = service.getAWSECommerceServicePort() ;

  ItemSearchRequest search
      = new ItemSearchRequest() ;
  search.setSearchIndex("Books") ;
  search.setKeywords("Dogs") ;
  Holder<Items> items = new Holder<List<Items>>() ;

  port.itemSearch(awsAccessKey, request, items) ;
  for(Item item : items.value.getItem()) {
    out.print(item.getASIN()) ;
    out.print(item.getTitle()) ;
    for(Offer offer : item.getOffers().getOffer()) {
      out.print(offer.getOfferListingId()) ;
      out.pring(offer.getPrice().getAmount()) ;
    }
  }


  String ASIN = // user selected product
  String offerListingId = // user selected offer
  String cartId = null ;
  String HMAC = null ;

  CartCreateRequest cartRequest
      = new CartCreateRequest() ;
  CartCreateRequest.Items.Item cartItem
      = new CartCreateRequest.Items.Item() ;
  cartItem.setOfferListingId(offerListingId) ;
  cartItem.setQuantity(1) ;
  CartCreateRequest.Items cartItems
      = new CartCreateRequest.Items() ;
  cartItems.getItem().add(cartItem) ;
  cartRequest.setItems(cartItems) ;
  Holder<Cart> cart = new Holder<Cart>() ;

  port.cartCreate(awsAccessKey, cartRequest, cart) ;
  cartId = cart.getCartId() ;
  HMAC = cart.getHMAC() ;
  for(CartItem item :
      cart.getCartItems().getCartItem()) {
    out.print(item.getItem().getASIN()) ;
    out.print(item.getItem().getTitle()) ;
    out.print(item.getPrice().getAmount()) ;
    out.print(item.getQuantity()) ;
  }
  out.print(cart.getSubtotal().getAmount()) ;
  out.print(cart.getPurchaseURL()) ;
}
```

Java using batched API and batch keyword

```java
void shoppingSequence() {
  BatchClient amazonService
      = new BatchClient(SERVER) ;



  batch(AmazonService service : amazonService) {
    service.login(awsAccessKey) ;
    final SearchCriteria crit =
        service.createSearchCriteria() ;
    crit.setSearchIndex("Books") ;
    crit.setKeywords("Dogs") ;

    for(final Item item : service.search(crit)) {
      out.print(item.getASIN()) ;
      out.print(item.getTitle()) ;
      for(final Offer offer : item.getOffers()) {
        out.print(offer.getOfferListingId()) ;
        out.pring(offer.getPrice().getAmount()) ;
      }
    }
  }

  String ASIN = // user selected product
  String offerListingId = // user selected offer
  String cartId = null ;
  String HMAC = null ;

  batch(AmazonService service : amazonService) {
    service.login(awsAccessKey) ;
    final Offer offer =
        service.getOffer(ASIN, offerListingId) ;









    final Cart cart = service.createCart(offer, 1) ;
    cartId = cart.getCartId() ;
    HMAC = cart.getHMAC() ;
    for(final CartItem item : cart.getCartItems()) {
      out.print(item.getItem().getASIN()) ;
      out.print(item.getItem().getTitle()) ;
      out.print(item.getPrice().getAmount()) ;
      out.print(item.getQuantity()) ;
    }
    out.print(cart.getSubtotal().getAmount()) ;
    out.print(cart.getPurchaseURL()) ;
  }
}
```

Figure 11: Example clients in Java

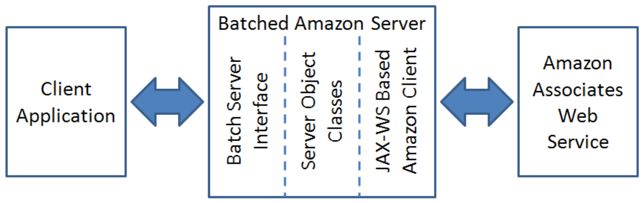

Figure 10: Architecture of Batched Amazon Web Service

5.2 Batching Mechanisms

Examining the AWS API and its accompanying documentation indicates that Amazon is concerned about the ability of their service to handle large numbers of small requests. Specifically, Amazon limits access to AWS to at most one request per second per IP address. To prevent this constraint from impeding the use of AWS, the API includes three different methods for submitting multiple requests in a single transaction. The first method is *batch requests*, which allow up to two separate requests involving the same operation to be combined into a single transaction. The second method is the `MultiOperation` operation, allowing up to two different basic operations to be included in one transaction, with up to two different requests for one of those operations. This method effectively allows up to three requests, one for one operation and two for another operation, to be combined into a single transaction to the web service. The final mechanism for batching is unique to the `ItemLookup` operation; up to ten item ids can be included in a single `ItemLookup` request.

While allowing some performance gain, the ad-hoc batching mechanisms provided by Amazon severely constrain the types of interactions that can be represented in a single transaction. A batching mechanism such as RBI-WS allows for a much wider range of interactions to be represented within a transaction. However, there are potential disadvantages to adopting a more general batching mechanism. As mentioned earlier, the unconstrained nature of our batches could allow for denial of service attacks in which a batch uses excessive resources. Since AWS's batching mechanisms all have specific, small upper-bounds on the number of batched operations, combined with various other bounds in the system, one can infer that Amazon is concerned about the resource utilization of individual transactions. Therefore, as suggested earlier, some mechanism for limiting the resource usage of a batch may be necessary.

5.3 Server-side Aggregation

In the current batch model, a remote variable can only be assigned once and only at its declaration point. This model limits the ability to express aggregation over collections of objects on the server. The absence of aggregation makes it difficult to express certain types of remote operations naturally. For example, consider an application for the Amazon service that attempts to add the cheapest offer for a particular product to a shopping cart. Such an application would naturally be expressed as follows:

```
batch(AmazonService service : amazonService) {
  final Item item = service.getItem(ASIN) ;

  Offer minOffer = null;

  for(final Offer offer : item.getOffers()) {
    if(minOffer == null ||
        offer.getPrice() < minOffer.getPrice()) {
      minOffer = offer ;
    }
  }

  service.getCart(cartId, HMAC).add(minOffer, 1) ;
}
```

For this batch to execute successfully, the variable `minOffer` must be a remote variable. However, this requires `minOffer` to be declared as final, and therefore the assignment inside of the `if`-statement would raise a compiler error. One approach to address this limitation is to add dedicated aggregator classes. In this case, we can define a class with the following interface and an appropriate factory method on the `AmazonService` class:

```
interface OfferHolder {
  void setOffer(Offer offer) ;
  Offer getOffer() ;
  boolean hasOffer() ;
}
```

Now we can change the batch above to the following:

```
batch(AmazonService service : amazonService) {
  final Item item = service.getItem(ASIN) ;

  fina OfferHolder minOffer = service
    .createOfferHolder() ;

  for(final Offer offer : item.getOffers()) {
    if(!minOffer.hasOffer() || offer.getPrice()
        < minOffer.getOffer().getPrice()) {
      minOffer.setOffer(offer) ;
    }
  }
  service.getCart(cartId, HMAC)
    .add(minOffer.getOffer(), 1) ;
}
```

Similar classes can be defined for any of the service interfaces for which aggregation might be desired as well as for the basic data-types. Unfortunately, this requires that the web service programmer have the foresight to provide these aggregator classes.

6. Related Work

Remote Batch Invocation for Web Services (RBI-WS) exposes to the programmer a new programming abstraction that aims at providing greater expressiveness without jeopardizing performance. Addressing the challenges of distributed computing through intuitive programming abstractions has been the target of numerous prior research efforts. Since the research literature on the topic covers a wide and diverse spectrum of ideas and approaches, we only compare RBI-WS with closely related state of the art.

Although Remote Procedure Call (RPC) (Tay and Ananda 1990) has been one of the most prevalent communication abstractions for constructing distributed systems, its shortcoming and limitations have been continuously highlighted by different researchers (Tanenbaum and Renesse 1988; Waldo et al. 1994; Saif and Greaves 2001), and some of them suggest that RPC has had harmful influence on the development of distributed systems (Vinoski 2005). The document-oriented interfaces of Web services have been promoted as an alternative to RPC. Despite the criticisms of RPC and its object-oriented counterparts, accessing distributed functionality through a familiar method call paradigm provides unquestionable convenience advantages. RBI-WS enables the programmer to leverage the performance advantages of document-oriented interfaces by using easy-to-compose object-oriented interfaces.

The design of document-oriented web services is a complex area that involves many factors, including technology, interoperability, and transactions (Papazoglou et al. 2007; Singh et al. 2004; Dijkman et al. 2003; Fielding et al. 2002). A primary concern is the *granularity* of service requests. Sun's Java Blueprints (Singh et al. 2004) advises to "consolidate related fine-grained operations into more coarse-grained ones to minimize expensive remote method calls", warns that "too much consolidation leads to inefficiencies", and concludes that designers should "ensure that the Web service operations are sufficiently coarse grained". The contradiction between these recommendations stems from the impossibility of creating a single level of granularity that will work for all clients. Explicit batches solve this problem by allowing clients to perform operations at the required level of granularity.

The SOAP Bundling Framework (Takase and Tajima 2007) is a web service proxy that allows sequential batches of multiple calls to an underlying web service. The calls are independent and do not support loops or conditionals.

Representational State Transfer (REST) is an architectural model that is an alternative to SOAP web services (Fielding and Taylor 2000). REST web services rely as much as possible on web protocols for handling caching, security, naming, etc. A REST request is an URL with a path and form parameters, which are frequently interpreted as an object address and method parameters. As such REST resembles a very abstract fine-grained RPC, or a shell command. The output can be any valid hypertext media such as HTML or images. REST has a simpler request model than XML, and this ease of use contributes to its popularity. Although contracts are often promoted as one of the benefits of service-orientation, REST does not currently support formal interface specifications, analogous to WSDL. The main problem is that REST, like RPC, is not latency compositional. URLs do not naturally combine to form compound requests. While defining composite URLs is certainly possible, we find it easier to define composition in SOAP services, since XML is naturally compositional.

Software design patterns (Fowler 2002) for *Remote Façade* and *Data Transfer Object* (also called Value Objects (Alur et al. 2003)) can be used to optimize remote communication. A *Remote Façade* allows a service to support specific client call patterns using a single remote invocation. Different Remote Façades may be needed for different clients. RBI-WS enables the creation of a custom Remote Façade for each client as long as the client call pattern is supported as a single batch. A *Data Transfer Object* is a `Serializable` class that provides block transfer of data between client and server. As with the Remote Façade, different kinds of Data Transfer Objects may be needed by different clients. RBI-WS constructs a value object on the fly, automatically, exactly as needed in a particular situation. RBI-WS also generalizes the concept of a data transfer object to support transfer of data from arbitrary collections of objects.

Cook and Barfield (Cook and Barfield 2006) first pointed out that documents can be viewed as batches of primitive operations. They showed how a set of hand-written wrappers can provide a mapping between object interfaces and batched calls expressed as a web service document. RBI-WS automates the process of creating the wrappers and generalizes the technique to support remote conditionals and operations on collections. In essence, RBI-WS program can scale as well as hand-optimized web services (Demarey et al. 2005). Web services choreography (Peltz 2003) defines how Web services interact with each other at the message level. RBI-WS can be seen as a programming abstraction for choreographing efficient access to remote object-oriented services.

7. Future Work

In the future, we plan to continue this work in the following directions. First, while designing and developing RBI-WS, we have made several choices that may have impacted the usability and expressiveness of our methodology. For instance, we chose to use a single type to represent all numbers in a given interface. While this decision has simplified the interfaces, it can also complicate their use, as the type no longer provides any hint about the range or precision of the expected value. Similarly, the current model allows the construction of relatively unconstrained batches of operations, which can lead to security problems. Therefore we intend to investigate how these decisions have impacted various properties of RBI-WS and improve on them if necessary.

Additionally, our exploration of the Amazon Associates Web Service revealed various shortcomings of our design and suggested new features that could be beneficial. As mentioned in Section 5.3, for many types of use-cases, the ability to perform aggregation on the server over a set of objects is necessary. Currently, we have provided an ad-hoc solution that requires the service developer to create aggregation objects. A more automatic approach is possible and should be explored.

Also, AWS includes several different search operations, many of which include a large number of searching, sorting, and paging parameters. Currently, we use search criteria objects to specify these search parameters. The use of these objects incurs the disadvantage of separating the act of specifying search criteria from the act of searching. To address this issue, we are considering several approaches to specifying search criteria in a more natural manner.

8. Conclusion

This paper has argued that document-oriented interfaces can be effectively represented as batches of method calls to fine-grained object-oriented interfaces. An input document can contain information expressing object instantiation, selection, access, and update. An output document can encapsulate multiple results. In the opposite direction, a document can be specified by combining a block of fine-grained object-oriented invocations into a batch. Our approach enables the programmer to express how the statements in a block operate directly on virtual service objects, without the need to explicitly construct invocation objects and correlate them to the response. In addition, batch blocks can include conditional expressions, loops, and exception handling. Our reference implementation, Remote Batch Invocation for Web Services, represents object-oriented interfaces as a WSDL that describes a batch of invocations. The WSDL is accessible by standard web service clients. In addition, we provide two approaches that streamline such access: a batch Java language extension and a prototype BRMI C# middleware platform. Our powerful web services infrastructure directly connects to object-oriented interfaces, providing tool support for automatically creating and processing documents that embody sequences of invocations.

As experimental validation, we have created a Web service wrapper for the Amazon Associates Web service, showing how remote batches enable a clean object-oriented style for programming a stateless web service, without needing remote object proxies.

All in all, this work explores the following novel ideas. It discusses the relationship between document-oriented and object-oriented programming interfaces. It shows how a set of object-oriented interfaces can be effectively translated into a web service DSL defined by a XML schema. Finally, this work demonstrates the utility of RBI-WS by applying it to a real-world web service.

References

D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003.

D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) version 1.1, 2002.

E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl, 2001. URL `http://www.w3.org/TR/wsdl`.

W. Cook and J. Barfield. Web Services versus Distributed Objects: A Case Study of Performance and Interface Design. In *the IEEE International Conference on Web Services (ICWS'06)*, pages 419–426, 2006.

C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. *Studia Informatica Universalis Regular Issue*, 4(1): 7–24, 2005.

R. Dijkman, D. Quartel, L. F. Pires, and M. van Sinderen. The state-of-the-art in service-oriented computing and design. Technical Report ArCo Project Deliverable ArCo/WP1/T1/D1/V1.00, University of Twente, 2003.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. UML profile for enterprise distributed object computing. Technical Report OMG Document ptc/2002-02-05, 2002.

R. T. Fielding and R. N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM. ISBN 1-58113-206-9. doi: http://doi.acm.org/10.1145/337180.337228.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321127420.

R. Gabriel. Is worse really better? *Journal of Object-Oriented Programming (JOOP)*, 5(4):501–538, 1992.

A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In *The 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, July 2009. URL `http://www.cs.utexas.edu/~aibrahim/publications/batches.pdf`.

B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI 1988: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54016.

M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007. doi: http://dx.doi.org/10.1109/MC.2007.400. URL `http://dx.doi.org/10.1109/MC.2007.400`.

C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

U. Saif and D. Greaves. Communication primitives for ubiquitous systems or RPC considered harmful. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 240–245, 2001.

I. Singh, S. Brydon, G. Murray, V. Ramachandran, T. Violleau, and B. Stearns. *Designing Web Services with the J2EE 1.4 Platform: JAX-RPC, XML Services, and Clients*. Pearson Education, 2004. ISBN 0321205219.

T. Takase and K. Tajima. Efficient web services message exchange by SOAP bundling framework. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 63, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2891-0.

A. S. Tanenbaum and R. v. Renesse. A critique of the remote procedure call paradigm. In *EUTECO 88*, pages 775–783. North-Holland, 1988.

B. Tay and A. Ananda. A survey of remote procedure calls. *Operating Systems Review*, 24(3):68–79, 1990.

E. Tilevich, W. Cook, and Y. Jiao. Explicit batching for distributed objects. In *The 29th International Conference on Distributed Computing Systems*, June 2009. URL `http://www.cs.utexas.edu/~wcook/Drafts/2008/brmi.pdf`.

S. Vinoski. RPC Under Fire. *IEEE INTERNET COMPUTING*, pages 93–95, 2005.

J. Waldo, A. Wollrath, G. Wyant, and S. Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1994.

E. Wilde and K. Stillhard. Making xml schema easier to read and write. In *WWW 2003*, 2003.

D. Winer. *XML-RPC Specification*, 1999.

A. Appendix A

```
<schema targetNamespace=
  "http://www.cs.utexas.edu/aibrahim/batch"
        xmlns:s=
  "http://www.w3.org/2001/XMLSchema"
        xmlns:batch=
  "http://www.cs.utexas.edu/aibrahim/batch">

 <s:complexType name="OutputBinding">
   <s:sequence>
     <s:element name="value" type="batch:Any"
                minOccurs="0"
                maxOccurs="unbounded"/>
   </s:sequence>
   <s:attribute name="key" type="s:string"
                use="required"/>
 </s:complexType>

 <s:element name="output">
   <s:complexType>
     <s:sequence>
       <s:element name="binding"
                  type="batch:OutputBinding"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
     </s:sequence>
   </s:complexType>
 </s:element>

 <s:element name="batch">
   <s:complexType>
     <s:sequence>
       <s:element name="op"
                  type="batch:Operation" />
     </s:sequence>
   </s:complexType>
 </s:element>

 <s:complexType abstract="true"
                name="Operation">
   <s:attribute name="binding"
                type="s:string"/>
   <s:attribute name="neededLocally"
               type="s:boolean"
               default="false"/>
</s:complexType>

<s:element name="seq"
           type="batch:Sequence"/>
<s:complexType abstract="false"
               name="Sequence">
  <s:complexContent>
    <s:extension base="batch:Void">
      <s:sequence>
        <s:element name="step"
                   type="batch:Operation"
                   maxOccurs="unbounded"/>
      </s:sequence>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="if" type="batch:IfStmt"/>
<s:complexType abstract="false"
               name="IfStmt">
  <s:complexContent>
    <s:extension base="batch:Void">
      <s:sequence>
        <s:element name="cond"
                   type="batch:Boolean"/>
        <s:element name="then"
                   type="batch:Operation"/>
        <s:element name="else"
                   type="batch:Operation"/>
      </s:sequence>
    </s:extension>
       </s:complexContent>
</s:complexType>

<s:element name="cursor"
           type="batch:Cursor"/>
<s:complexType abstract="false"
               name="Cursor">
  <s:complexContent>
    <s:extension base="batch:Void">
      <s:sequence>
        <s:element name="cursorName"
                   type="s:string"/>
        <s:element name="collection"
                   type="batch:Collection"/>
        <s:element name="body"
                   type="batch:Operation"/>
      </s:sequence>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="negation"
           type="batch:Negation"/>
<s:complexType abstract="false"
```

```
               name="Negation">
   <s:complexContent>
     <s:extension base="batch:Number">
       <s:sequence>
         <s:element name="operand"
                    type="batch:Number"/>
       </s:sequence>
     </s:extension>
   </s:complexContent>
 </s:complexType>

 <s:element name="not" type="batch:Not"/>
 <s:complexType abstract="false" name="Not">
   <s:complexContent>
     <s:extension base="batch:Boolean">
       <s:sequence>
         <s:element name="operand"
                    type="batch:Boolean"/>
       </s:sequence>
     </s:extension>
   </s:complexContent>
 </s:complexType>

 <s:element name="and" type="batch:And"/>
 <s:complexType abstract="false" name="And">
   <s:complexContent>
     <s:extension base="batch:Boolean">
       <s:sequence>
         <s:element name="leftOperand"
                    type="batch:Boolean"/>
         <s:element name="rightOperand"
                    type="batch:Boolean"/>
       </s:sequence>
     </s:extension>
   </s:complexContent>
 </s:complexType>

...

 <s:element name="greaterThan"
            type="batch:GreaterThan"/>
 <s:complexType abstract="false"
                name="GreaterThan">
   <s:complexContent>
     <s:extension base="batch:Boolean">
       <s:sequence>
         <s:element name="leftOperand"
                    type="batch:Number"/>
         <s:element name="rightOperand"
                    type="batch:Number"/>
       </s:sequence>
     </s:extension>
   </s:complexContent>
 </s:complexType>

...

 <s:element name="Plus" type="batch:Plus"/>
 <s:complexType abstract="false" name="Plus">
   <s:complexContent>
     <s:extension base="batch:Number">
       <s:sequence>
         <s:element name="leftOperand"
                    type="batch:Number"/>
         <s:element name="rightOperand"
                    type="batch:Number"/>
       </s:sequence>
     </s:extension>
   </s:complexContent>
 </s:complexType>

...

 <s:element name="void" type="batch:Void"/>
 <s:complexType abstract="true" name="Void">
   <s:complexContent>
     <s:extension base="batch:Operation">
      <s:sequence/>
     </s:extension>
   </s:complexContent>
 </s:complexType>

 <s:element name="any" type="batch:Any"/>
 <s:complexType abstract="true" name="Any">
   <s:complexContent>
     <s:extension base="batch:Operation">
      <s:sequence/>
     </s:extension>
   </s:complexContent>
 </s:complexType>

 <s:element name="null" type="batch:Null"/>
 <s:complexType abstract="true" name="Null">
   <s:complexContent>
     <s:extension base="batch:Any">
      <s:sequence/>
     </s:extension>
   </s:complexContent>
 </s:complexType>

 <s:element name="exception"
            type="batch:Exception"/>
 <s:complexType abstract="false"
                name="Exception">
   <s:complexContent>
     <s:extension base="batch:Any">
      <s:sequence/>
      <s:attribute name="type" type="s:string"/>
      <s:attribute name="msg" type="s:string"/>
     </s:extension>
   </s:complexContent>
 </s:complexType>

 <s:element name="cursorValue"
```

```
           type="batch:CursorValue"/>
<s:complexType abstract="false"
               name="CursorValue">
  <s:complexContent>
    <s:extension base="batch:Any">
     <s:sequence>
      <s:element name="var" type="s:string"
          minOccurs="0" maxOccurs="unbounded"/>
     </s:sequence>
     <s:attribute name="size" type="s:integer"/>
    </s:extension>
  </s:complexContent>
</s:complexType>

...

<s:element name="number" type="batch:Number"/>
<s:complexType abstract="true" name="Number">
  <s:complexContent>
    <s:extension base="batch:Any">
      <s:sequence/>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="numberRef"
           type="batch:Number__Ref"/>
<s:complexType name="Number__Ref">
  <s:complexContent>
    <s:extension base="batch:Number">
      <s:sequence/>
      <s:attribute name="ref"
                   type="s:string"/>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="numberValue"
           type="batch:Number__Value"/>
<s:complexType name="Number__Value">
  <s:complexContent>
    <s:extension base="batch:Number">
      <s:sequence/>
      <s:attribute name="val"
                   type="s:string"/>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="coll"
           type="batch:Collection"/>
<s:complexType abstract="true"
               name="Collection">
  <s:complexContent>
    <s:extension base="batch:Any">
      <s:sequence/>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="collNumber"
           type="batch:Collection__Number"/>
<s:complexType abstract="true"
               name="Collection__Number">
  <s:complexContent>
    <s:extension base="batch:Collection">
      <s:sequence/>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="collNumberRef"
  type="batch:Collection__Number__Ref"/>
<s:complexType
  name="Collection__Number__Ref">
  <s:complexContent>
    <s:extension
      base="batch:Collection__Number">
      <s:sequence/>
      <s:attribute name="ref"
                   type="s:string"/>
    </s:extension>
  </s:complexContent>
</s:complexType>

<s:element name="collNumberValue"
  type="batch:Collection__Number__Value"/>
<s:complexType
  name="Collection__Number__Value">
  <s:complexContent>
    <s:extension
      base="batch:Collection__Number">
      <s:sequence>
        <s:element name="item"
          type="batch:Number"
          maxOccurs="unbounded"/>
      </s:sequence>
    </s:extension>
  </s:complexContent>
</s:complexType>

...

</schema>
```