Remote Batch Invocation for Web Services
Document-Oriented Web Services with Object-Oriented Interfaces

Ali Ibrahim, William R. Cook
Dept. of Computer Science
University of Texas at Austin
{*aibrahim,wcook*}*@cs.utexas.edu*

Marc Fisher II, Eli Tilevich
Dept. of Computer Science
Virginia Tech
{*fisherii,tilevich*}*@cs.vt.edu*

***Abstract*—The Web Service Description Language defines a service as a procedure whose inputs and outputs are structured XML data values, sometimes called *documents*. In this paper we argue that document-oriented interfaces can be viewed as batches of calls to finer-grained procedural interfaces. Turning this correspondence around, we show that flexible documents can be specified by converting a block of fine-grained invocations into a batch document. 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 by providing a language extension to support remote batches, our approach features a fully integrated client environment. 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. 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.**

I. 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* (RPC) [1], *document-oriented services* [2], [3] and *representation state transfer* (REST) [4] 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 structured XML documents [5].

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 these forms. Despite this complexity, there is no standard methodology for detailed design of service documents or specification of their semantics.

Remote Batch Invocation for Web Services (RBI-WS) is a new approach to document-oriented web services supporting object-oriented interfaces. We show how fine-grained object interfaces can be translated into a web service interface (WSDL) whose instances describe batches of fine-grained calls. The translation is bi-directional, the web service interface can be translated back into fine-grained object interfaces for use by the client.

RBI-WS is supported by several tools. At design time, tools translate between WSDL interfaces and collections of object interfaces. A runtime server library supports automatic interpretation of batch web service documents, by executing each of the primitive object calls specified in the batch. A batch can contain conditionals and iterations, in addition to sequences of basic calls. A client library can be used to create batches. Finally, an extension of java with a batch statement is compiled by a source-to-source translator that replaces batch statements with appropriate calls to the client library. Standard SOAP/WSDL tools, for example Apache Axis [6], can also be used to access batch services.

We evaluate RBI-WS by redefining an existing web service, in this case part of the Amazon Wed Service (AWS) suite. 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.

RBI-WS is an extension of our previous work [7] on batches for Java Remote Method Invocation (RMI). The contributions of this paper are the design of a mapping between batches and web service documents (WSDL), and a demonstration that batches are a natural generalization of an existing document-oriented service. This work is a first step toward a more systematic approach to design of web service documents.

II. Documents and Interfaces

It is well known that code can be represented as a document; code in most languages can be expressed as abstract syntax trees. Conversely, one might wonder whether it is 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 sentence in this language is a document. A document can be viewed as having a mix of data and operations to perform on the data. If the operations can be composed in different ways such as batching, sequencing, and branching, then it makes sense to think of the document as representing code. We argue this is often the case in large services. Because performance and flexibility are important, the web service allows clients to compose multiple operations in a single document.

We will think of each individual operation in a document as a method in the server API. How to divide the document into data and operations is subjective; it depends on how the programmer would like to think about the document.

A. 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

```
<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

```
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: Calls to represent Figure 1

sample input document a client could send to the Amazon Associates Web Service (AWS). The document represents a request to selects two Amazon items by their ASIN ids and retrieve 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. A possible set of interfaces is presented in Figure 11 and described in Section V.

The document in Figure 1 can be viewed as a script that specifies the sequence of calls 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 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

```
<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

```
aws.login("ABC");
Cart cart = aws.getCart("0","XYZ");
cart.moveToCart(cart.getCartItem(0), 1);
cart.saveForLater(cart.getCartItem(1));
```

Figure 4: Calls to represent Figure 3

Amazon shopping cart. This document can be viewed as representing the sequence of calls in Figure 4.

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.

B. Blocks with Control Flow as Documents

If we can think of documents as representing programs using specialized APIs, then it is natural to think about the commonalities between these programs. The specific operations in an API vary for each web service, but the composition operators are common: batching, sequence, branching, and looping. Given a set of object-oriented interfaces, can we produce a XML schema which describes documents that are specific to that API and incorporates common composition operators? We would like the encoding to have the following properties:

1) The XML schema should allow the clients to specify batches of API calls with support for conditionals, loops, and exceptions. These constructs allow flexible composition of operations.
2) The XML schema should specify as much information as possible about the set of interfaces including type information. We should be able to recover the original interfaces from the XML schema.
3) The XML schema corresponding to the language should produce a reasonable set of data transfer classes when given to standard code generator tools such as Axis [6] and Microsoft Visual Studio [8]. This property is a practical consideration given we would like programmers to use our XML schema directly for code generation.

The latter point 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 also tells the server to send this variable binding to the client. The keyword `root` represents the root service object. We decided to have booleans, integers, doubles, and strings serve as the language's primitive types. The primitive types can be manipulated using common unary and binary operators such as $+$ and $\vee$. The method calls $m_1 \ldots m_n$ represent a finite set of method calls available on the server.

What is interesting about this family of DSL is that they are limited compared to general purpose languages. Since we see these languages as glue languages, we decided

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

Figure 5: Domain Specific Language for Web Services

```
let a = root.getItem("1111");
let b = root.getCart("222", "xxx");
b.add(a, 1);
```

Figure 6: Example web DSL code.

```
<sequence>
 <let var="a">
  <getItem><string>1111</string></getItem>
 </let>
 <let var="a">
  <getCart>
   <string>1111</string><string>xxx</string>
  </getCart>
 </let>
 <add>
  <cart_ref ref="b"/><item_ref ref="a"/><num>1</num>
 </add>
</sequence>
```

Figure 7: First try at representing code in Figure 6 as XML.

to omit support for procedural abstraction or modules. There are a couple of other interesting design choices. The language does not provide a natural way to perform aggregation, although the programmer can workaround this restriction with the help of the server (Section V-C discusses this in more detail). Another interesting omission is the lack of constructors. Instead the language relies on the web service providing factory methods for constructing objects.

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 a specific set of methods on any object that is *reachable* from the root service object. An object is reachable if it is the return value of a method defined in the DSL on a reachable object. There are many ideas for limiting accessibility further such as 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.

C. Encoding Web DSL programs as XML

There are many possible encodings of a web service DSL program into XML. Figure 6 shows an example of a web DSL program. Our first attempt at encoding this program in XML is in Figure 7. Each element represents an AST object where the name of the element is the object's type. Fields of the AST objects are specified as an ordered list of child elements. For example, the add element represents a call to the Cart#add method and the first child element represents the cart to invoke the add method on.

Our stated goals for the schema encoding of web DSL programs is to preserve type information about the interface and to produce a schema that can be consumed by WSDL code generation tools. The XML schema which describes documents such as in Figure 7 requires substitution groups to enforce typing constraints. Unfortunately, neither of our WSDL code generation tools (Axis 2 and Microsoft Visual Studio) could recognize sub-typing. expressed in this fashion. Moreover, the differentiation of sub-elements using position resulted in the code generators not correctly naming fields in the generated classes.

Instead, our XML schema encoding uses explicit type polymorphism as shown in Figure 8. In this encoding, each element represents an AST object and the type attribute specifies the object's class and the element name specifies the field name in the enclosing object. This encoding is more verbose because of the additional type attribute definition, but our WSDL code generation tools were able to interpret this form of polymorphism and were able to generate meaningful field names in the generated classes.

D. Encoding Interfaces as XML Schema

We will now describe our algorithm for encoding a set of interfaces into a XML schema which describes XML documents such as the one in Figure 8. The schema enforces a basic level of type safety, e.g. the condition

```
<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">
  <this type="Cart__Ref" ref="b" />
  <p1 type="Item__Ref" ref="a" />
  <p2 type="Number" value="1" />
 </step>
</batch>
```

Figure 8: XML enconding of program in Figure 6.

child element of an conditional element should produce a boolean value. However, the language is not strongly typed, the programmer asserts the type of references which creates a hole in the type system. We were not able to plug this hole without too much additional complexity.

The schema can be divided into a generic schema which is the same for all sets of interfaces and an interface schema which is unique for a set of interfaces. The generic schema defines schema types for the composition operations and primitive types. The *Operation* type is the base type for all types (We will use XSCS [9] syntax to express XML schema).

```
abstract complexType Operation {
  attribute binding { string }
  attribute neededLocally { boolean } }
```

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 operation element represents the name of the handle for the result of that operation. The XML attribute *neededLocally* in the batch operation tag allows 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*. The value types include numbers, booleans, and strings. Control flow structures provided are sequences, conditionals, and loops. For example, the schema definition for **if** conditional statements is below.

```
complexType IfStmt extends Void { {(
    cond { Boolean },
    then { Operation },
    else { Operation }) } }
```

The interface schema defines schema types for each interface. Each interface I is translated into four types in the schema.

```
abstract complexType I extends super(I)

complexType I__Ref extends I
  { attribute val { boolean } }

abstract complexType Coll__I {}

complexType C__I__Ref extends C__I
  { attribute val { boolean }   }

complexType C__I__Val extends C__I
  { (item { I } [0,] )   }
```

The *super* function finds the first ancestor of I that is defined in the schema, because there is no restriction that the set of interfaces contain their super-types as well. The schema types ending in `__Ref` represent references to a type and are used to reference the variables defined in let statements. The schema types beginning with `C__` represent the collections parameterised by that type. The schema type beginning with `C__` and ending with `__Val` represents a literal collection of values of a type (such as a literal array of integers). There is no support for sub-typing relations between these collection types. Each method m in I is translated into a schema type.

```
complexType I_m extends returnTyp(m) { (
    this { name(I) } if isRoot(I) [0,1],
    p1 { paramType(m, 1) }
     , ...,  pk { paramType(m, k) }
  )}
```

The method schema type is a sub-type of the return type of the method, because that is the value produced by the method invocation. The `this` sub-element represents the receiver of the call. It is optional if the interface is the root interface. If omitted, the method call is invoked on the root interface object. Subsequent sub-elements represent the parameters of the method.

For example, the *AmazonService__getItem* tag in Figure 8 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.

The return document schema consists of a return map element. The return map element contains a list of pairs of handle names and value elements. Value elements in the return message can either be any value element defined in the input document schema plus exception elements. Exception elements contain a name string and a message string.

III. 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 which creates the XML schema for the server interfaces. 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.

```
...
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 9: Code listing for batch client using Axis 2 WSDL2Java generated interface.

IV. 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 advantage 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 a different approach for streamlining RBI-WS client programming. This approach extends Java with a new keyword, **batch**. In the following discussion, we first outline the default client bindings, and then we explain how we have adapted the batch extension, detailed in a prior publication [7], for the needs of RBI-WS.

A. 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™.

The client code fragment in Figure 9 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 9 obscures the programmer's intentions. The reason for poor readability is that this code creates abstract syntax objects, 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

```
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 10: Code listing for batch client using batch language extension.

to construct the parameters and set their values, before passing the parameters to service methods. To hide some of this complexity, we next describe an approach that can smooth away the rough edges of the default RBI-WS client bindings.

B. The Batch Extension

We have extended the Java language with new syntax that supports defining a batch with a mix of local and remote operations [7]. The programmer generates the interfaces to a web service by running a custom tool which takes a WSDL and reverses the translation described in Section II-D. 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 10 shows how to rewrite the example in Figure 9 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 [10]. 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.

V. 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 12 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 22-23). A new shopping cart is then created containing one copy of the selected item (lines 27-39), and the cart contents, total price, and a link to complete the purchase are displayed to the user (lines 42-50).

[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);
  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 getSalesRank();
  Image getSmallImage();
  Image getLargeImage();
  Offer[] getOffers();
  String getTitle();
}
interface Offer {
  Item getItem();
  String getOfferListingId();
  Price getPrice();
  String getAvailability();
}
```

Figure 11: Amazon Fine-Grained Interfaces

A. 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.

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 11 gives partial interfaces for five of the classes central to the item search

Java using JAX-WS and standard Amazon Associates API

```
1  void shoppingSequence() {
2    AWSECommerceService service
3        = new AWSECommerceService();
4    AWSECommerceServicePortType port
5        = service.getAWSECommerceServicePort();
6
7    ItemSearchRequest search
8        = new ItemSearchRequest();
9    search.setSearchIndex("Books");
10   search.setKeywords("Dogs");
11   Holder<Items> items = new Holder<List<Items>>();
12
13   port.itemSearch(awsAccessKey, request, items);
14   for(Item item : items.value.getItem()) {
15     out.print(item.getASIN());
16     out.print(item.getTitle());
17     for(Offer offer : item.getOffers().getOffer()) {
18       out.print(offer.getOfferListingId());
19       out.pring(offer.getPrice().getAmount());
20     }
21   }
22   String ASIN = // user selected product
23   String offerListingId = // user selected offer
24   String cartId = null;
25   String HMAC = null;
26
27   CartCreateRequest cartRequest
28       = new CartCreateRequest();
29   CartCreateRequest.Items.Item cartItem
30       = new CartCreateRequest.Items.Item();
31   cartItem.setOfferListingId(offerListingId);
32   cartItem.setQuantity(1);
33   CartCreateRequest.Items cartItems
34       = new CartCreateRequest.Items();
35   cartItems.getItem().add(cartItem);
36   cartRequest.setItems(cartItems);
37   Holder<Cart> cart = new Holder<Cart>();
38
39   port.cartCreate(awsAccessKey, cartRequest, cart);
40   cartId = cart.getCartId();
41   HMAC = cart.getHMAC();
42   for(CartItem item :
43       cart.getCartItems().getCartItem()) {
44     out.print(item.getItem().getASIN());
45     out.print(item.getItem().getTitle());
46     out.print(item.getPrice().getAmount());
47     out.print(item.getQuantity());
48   }
49   out.print(cart.getSubtotal().getAmount());
50   out.print(cart.getPurchaseURL());
51 }
```

Java using batched API and batch keyword

```
1  void shoppingSequence() {
2    BatchClient amazonService
3        = new BatchClient(SERVER);
4
5    batch(AmazonService service : amazonService) {
6      service.login(awsAccessKey);
7      final SearchCriteria crit =
8          service.createSearchCriteria();
9      crit.setSearchIndex("Books");
10     crit.setKeywords("Dogs");
11
12     for(final Item item : service.search(crit)) {
13       out.print(item.getASIN());
14       out.print(item.getTitle());
15       for(final Offer offer : item.getOffers()) {
16         out.print(offer.getOfferListingId());
17         out.pring(offer.getPrice().getAmount());
18       }
19     }
20   }
21   String ASIN = // user selected product
22   String offerListingId = // user selected offer
23   String cartId = null;
24   String HMAC = null;
25
26   batch(AmazonService service : amazonService) {
27     service.login(awsAccessKey);
28     final Offer offer =
29         service.getOffer(ASIN, offerListingId);
30
31     final Cart cart = service.createCart(offer, 1);
32     cartId = cart.getCartId();
33     HMAC = cart.getHMAC();
34     for(final CartItem item : cart.getCartItems()) {
35       out.print(item.getItem().getASIN());
36       out.print(item.getItem().getTitle());
37       out.print(item.getPrice().getAmount());
38       out.print(item.getQuantity());
39     }
40     out.print(cart.getSubtotal().getAmount());
41     out.print(cart.getPurchaseURL());
42   }
43 }
```

Figure 12: Example clients in Java

and shopping cart functionality provided by our service.

The right side of Figure 12 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 27-31). 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.

B. 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.

C. 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);
  final 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.

VI. Related Work

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) [11] has been one of the most prevalent communication abstractions for constructing distributed systems, its shortcoming and limitations have been continuously highlighted by different researchers [12], [13], [14], [15]. 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 [3], [16], [17], [18]. A primary concern is the *granularity* of service requests. Sun's Java Blueprints [16] 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 [19] 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 [4]. 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 SOAP, 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 [20] for *Remote Façade* and *Data Transfer Object* (also called Value Objects [21]) 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 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 data transfer object on the fly, automatically, exactly as needed in a particular situation.

Cook and Barfield [22] 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 [23]. Web services choreography [24] defines how Web services interact with each other at the message level. Researchers [25] have looked at batching in BPEL, a web services choreography langauge. Their work batches web service invocations according to a static analysis of a BPEL program. RBI-WS can be seen as a programming abstraction for choreographing efficient access to remote object-oriented services.

ActiveXML [26] is a framework that uses web services for distributed data management. ActiveXML allows web service invocations to be placed within XML documents. The focus of ActiveXML is different from RBI-WS. ActiveXML is focused on data management; there is no native support for branching or loops. On the other hand, RBI-WS is focused of process management similar to BPEL. Another difference is that RBI-WS produces XML schema that are compatible with existing WSDL to Java translators while ActiveXML describes documents using an extension of XML schema.

VII. 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.

Since not all client environments lend themselves for extending their host language, we plan to experiment with integrating a library-based approach, similar to our previous work on Batch Remote Method Invocation (BRMI) [27]. This extension would have the potential to extend the range of applicability of RBI-WS.

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 V-C, 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.

VIII. 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. Although the WSDL is accessible by standard web service clients, we also provide an approach that streamlines such access in the form of a batch Java language extension. 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

[1] D. Winer, *XML-RPC Specification*, 1999.

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

[3] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[4] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, 2000, pp. 407–416.

[5] 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.

[6] *Apache Axis*, 2001. [Online]. Available: http://xml.apache.org/axis

[7] 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)*, July 2009. [Online]. Available: http://www.cs.utexas.edu/~aibrahim/publications/batches.pdf

[8] A. Ferrara and M. MacDonald, *Programming .Net Web Services*. O'Reilly & Associates, 2002.

[9] E. Wilde and K. Stillhard, "Making XML Schema easier to read and write," in *World Wide Web Conference (WWW)*, 2003.

[10] R. Gabriel, "Is worse really better?" *Journal of Object-Oriented Programming (JOOP)*, vol. 5, no. 4, pp. 501–538, 1992.

[11] B. Tay and A. Ananda, "A survey of remote procedure calls," *Operating Systems Review*, vol. 24, no. 3, pp. 68–79, 1990.

[12] A. S. Tanenbaum and R. v. Renesse, "A critique of the remote procedure call paradigm," in *European Teleinformatics Conference (EUTECO)*, 1988, pp. 775–783.

[13] J. Waldo, A. Wollrath, G. Wyant, and S. Kendall, "A note on distributed computing," Sun Microsystems, Tech. Rep., 1994.

[14] U. Saif and D. Greaves, "Communication primitives for ubiquitous systems or RPC considered harmful," in *Distributed Computing Systems Workshop, 2001 International Conference on*, 2001, pp. 240–245.

[15] S. Vinoski, "RPC under fire," *IEEE Internet Computing*, pp. 93–95, 2005.

[16] 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.

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

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, "UML profile for enterprise distributed object computing," Object Management Group, Tech. Rep. OMG Document ptc/2002-02-05, 2002.

[19] 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 (EDOC)*, 2007, p. 63.

[20] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, 2002.

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

[22] W. Cook and J. Barfield, "Web services versus distributed objects: A case study of performance and interface design," in *IEEE International Conference on Web Services (ICWS)*, 2006, pp. 419–426.

[23] 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*, vol. 4, no. 1, pp. 7–24, 2005.

[24] C. Peltz, "Web services orchestration and choreography," *IEEE Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[25] L. Bao, P. Chen, X. Zhang, S. Chen, S. Hu, and Y. Yang, "Batch invocation of web services in BPEL process," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC)*, 2008, pp. 511–516.

[26] S. Abiteboul, O. Benjelloun, and T. Milo, "The Active XML project: an overview," *The VLDB Journal*, vol. 17, no. 5, pp. 1019–1040, 2008.

[27] E. Tilevich, W. Cook, and Y. Jiao, "Explicit batching for distributed objects," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, June 2009. [Online]. Available: http://www.cs.utexas.edu/~wcook/Drafts/2008/brmi.pdf