Web Browsers as Service-Oriented Clients Integrated with Web Services

Hisashi Miyashita and Tatsuya Ishihara

IBM Research, Tokyo Research Laboratory, Japan
{himi, tisihara}@jp.ibm.com

Abstract. Web browsers are becoming important application clients in SOAs (Service-Oriented Architectures) because more and more Web applications are built from multiple Web Services. Therefore incorporating Web Services into Web browsers is of great interest. However, the existing Web Service frameworks bring significant complexities to traditional Web applications based on DHTML since such Web Service frameworks use RPC (Remote Procedure Call) or a message-passing model while DHTML is based on a document-centric model. Therefore Web application developers have to bridge the gaps between these two models such as an Object/XML impedance mismatch.

In our novel approach, in order to request Web Services, the application programs manipulate documents with uniform document APIs without invoking service-specific APIs and without mapping between objects and XML documents. The Web Service framework automatically updates the document by exchanging SOAP messages with the servers.

We show that in our new framework, WebDrasil, we can request a service with only one XPath expression, and then get the response using DOM (Document Object Model) APIs, an approach which is efficient and easily understood by typical Web developers.

1 Introduction

Service-Oriented Architecture (SOA) is an important technology to coordinate services across over multiple divisions in enterprise systems. These days, SOA on the client side is receiving attention for delivering services to end users [1], and many client frameworks are now supporting SOAs. For example, the Flex Framework by Adobe, the Eclipse based Rich Client Platform contributed by IBM, and the Mozilla Web browser [2] now support Web Services for SOA. Such a client having a close affinity to SOA is called an SOC (Service-Oriented Client) [1].

Of these clients, the Web browser is the most important platform for SOC. Recently, many websites or Web applications are combined using *mash-up* technology [3]. For example, HousingMaps (`http://www.housingmaps.com`) combines craigslist and Google maps and provides a totally new service to search for properties. This shift was triggered by the breakthrough technology, Ajax [4], which supports asynchronous access to distributed Web Services. Ajax allows a Web browser to be an intelligent client for Web Services by greatly improving the user experience. For example, Google provides Map and Calendar, and is now preparing a spreadsheet service by using Ajax.

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 289–301, 2006.

In line with the importance of Web browsers in service computing, supporting Web Services on Web browsers is becoming vital for SOA.

However, the existing Web Service frameworks such as JAX-WS (formerly JAX-RPC) [5] and Mozilla Web Service [2] do not fit with the programming model of Web browsers. These frameworks are designed not for Web browsers to present data for Web Services, but for native languages (JavaScript and Java) to easily access SOAP messages. Therefore, such frameworks are built on top of RPC (Remote Procedure Call) or a message-passing model. In contrast, Web browsers use a document-centric model. For example, in DHTML (Dynamic HTML) applications on Web browsers, scripts in the documents manipulate the in-memory tree structure, the *DOM (Document Object Model)*, to change the presentation. Since SOAP messages are now widely used as *document-literal* instead of *RPC/encoded* [1], they are viewed as XML documents, not as objects in Object-Oriented (OO) Programming. Therefore, introducing existing Web Services frameworks (from the OO world) into Web browsers (the XML world) doubles the gaps between these two models (as shown in Fig.1).

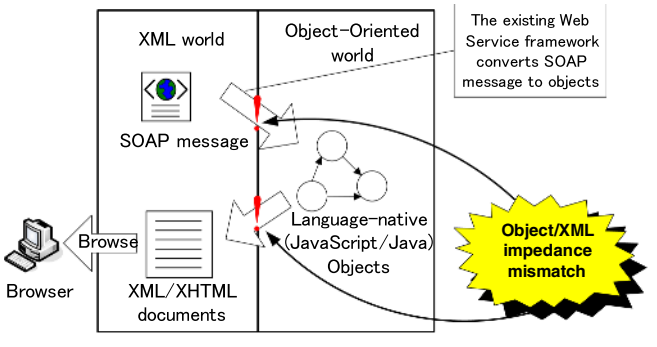

Fig. 1. The gaps by Web Services frameworks on Web browsers

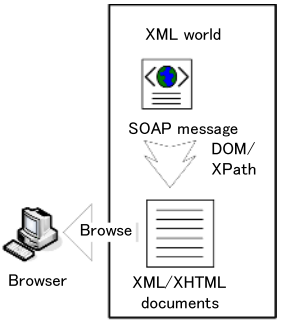

Fig. 2. Our approach

1.1 The Gaps in the Web Services Frameworks

Let us consider examples to clarify the gaps in a framework. In Fig. 3, we show a JavaScript code fragment that sends a request to Amazon.com Search Web Service on Mozilla using an RPC model. In this example, we construct an `searchRequest` object and then call the `KeywordSearchRequest` API to request the service. In contrast, to show the search results using DHTML, we write something like Fig. 4 using a document model. This code converts the `result` object to an HTML document, and inserts it into the DOM.

The gaps between the RPC and document models are twofold: 1) Non-uniform vs. uniform APIs and 2) Object/XML impedance mismatch [7].

For 1), in the RPC model, we invoke a service-specific API, `KeywordSearchRequest`. The names and arguments of such APIs differ from service to service. By comparison, in the document model, we use the uniform APIs, namely DOM APIs, to show the results. These same APIs can be used for services of any type.

[1] WS-I profile [6] does not support RPC/encoded for interoperability.

```
var searchRequest = new Object();
searchRequest.keyword=value;
searchRequest.page="1";
searchRequest.mode="books";
...
// proxy is Web Service Proxy object
proxy.KeywordSearchRequest(
                    searchRequest);
```

Fig. 3. Issue an Amazon search request (an RPC model)

```
var e = document.getElementById('resultid');
for (i = 0; i < result.Details.length; i++){
  e.innerHTML +="<p>"+
    result.Details[i].ProductName+"</p>";
}
```

Fig. 4. Show the search results in DHTML (a document model). The search results are stored in the `result` variable.

For 2), when we create an object to invoke a service, we have to manually convert the resulting object into a document written as HTML. To deal with this conversion, the developers have to understand how such language-native objects are mapped from SOAP messages. In other words, they have to know how the `result.Details[i].ProductName` object is translated from the `ProductName` element in the SOAP response that actually looks like:

```
... <ProductInfo>
        <Details url="...">
          <Asin>A00001O1XK</Asin>
          <ProductName> SOA Handbook </ProductName> ...
        </Details>
... </ProductInfo> ...
```

It is quite difficult for developers to understand how XML documents are mapped to language-native objects and how such objects should be converted to XML documents for presentation. In this example, it is not clear why `result.Details[0].ProductName` is correct, but `result.ProductInfo.Details.ProductName` is not correct. That depends on the specification of the Web Service framework.

As is shown by this example, in bridging the gaps between these two models, SOC application developers have to comprehend both programming and data models, which substantially increases development and maintenance costs for Web applications [7].

1.2 Document-Based Web Service Framework

We address the problems by introducing a new Web Service framework to integrate the programming models. In our approach, we can request services by manipulating documents (Fig. 2). For example, we can issue the same request to the Amazon.com Web Service with an XPath expression:

```
var request = webService.selectSingleNode(
 "./ws:Query[1]/ws:Request/aws:KeyWordSearchRequest\\
 [keyword='keyword' and page='1' and mode='books' ...]",
 namespaces);
```

Unlike existing Web Service stacks, our framework does not impose an object-XML mapping on the client applications. Rather, the client applications concentrate on manipulating documents by using uniform interfaces such as DOM and XPath. Our client framework translates such document manipulations into SOAP message exchanges, and

then caches the requests and responses in the DOM tree appropriately. Thus, we store the results in XML documents, not in objects. This model is completely aligned with DHTML, and free from the Object/XML impedance mismatch. Actually, we can even directly present the responses from the XML by setting the styles with CSS without any XML transformation. Of course, we can directly extract data from the response with XPath. For example, we can access all the `ProductName` elements in the previous example by specifying something like:

```
var productNames = response.selectNodeList(
                    ".//Details/ProductName", namespaces);
```

By unifying the programming models of Web Services and Web browsers, we can achieve seamless integration between them. Developers can seamlessly deal with the usual Web applications and Web Service clients rather than fighting with various APIs introduced by individual Web Services.

In addition, in our approach we can efficiently integrate Web Service technology with powerful and successful Web standards such as HTML, XSLT, CSS, and XForms. Web Service solution providers can rely on the power of these technologies when they design service-specific XML messages. With reduced effort, they can build stylish and attractive clients by transforming the DOM tree with XSLT, defining styles with CSS, and using XForms to create forms.

The rest of this paper is organized as follows. In Section 2, we introduce the architecture design principles of our novel Web Service client framework. In Section 3, we present our Web Service client implementation named *WebDrasil*, and use some examples to show the efficiency of our client Web Service programming model. Finally, Sections 4 and 5 provide related work and our conclusions, respectively.

2 Web Service Architecture Based on Web Browsers

We propose a novel framework for Web Services suitable for Web Browsers that removes the gaps between DHTML applications and the existing SOAP stacks. Rather than explicitly sending SOAP messages or invoking new interfaces (typically generated from WSDL), we simply access a tree with a uniform API such as DOM. We show the mechanism in Fig. 5:(1) First, we insert a request message into a tree; (2) Try to get the result by accessing the location where the response message is to be inserted into the tree; (3)–(5) The client framework automatically exchanges the required messages with the server(s) and inserts the response into the tree; (6) Finally, we can access the response.

Let us explain these steps by using an example. Suppose we want to retrieve the cached page of `http://www.ibm.com` via the Google Web Service. In our framework, as Step (1), we place the request shown in Fig. 6 into the DOM tree by using the DOM or XPath APIs (the details are discussed in Section 2.1). In this example, we insert the request under the `ws:Query` element, as shown in Fig. 7. In Step (2), we access the DOM tree where the response message will be stored, that is, under the `ws:Response` element, as shown in the left panel of Fig. 8. In Step (3), when the client framework detects the changes in the DOM tree, the framework translates the message under the `ws:Request` element into a SOAP message and then sends it to

the appropriate endpoint by using the WSDL definition of the service. In Step (4), the framework receives the SOAP response. In Step (5), the framework places the response from the SOAP message into the DOM tree, as shown in the right panel of Fig. 8. In Step (6), we find the required information in the `return` element, which should be the cached page of `http://www.ibm.com`. Notice that Steps (3)–(5) are automatically processed by the framework. Users can retrieve the result as though it already existed in the tree with this mechanism.

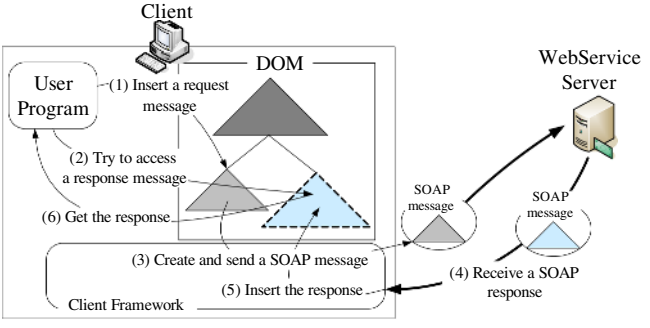

Fig. 5. The Mechanism of our Web Service Client Framework

```
<gws:doGetCachedPage xmlns:gws="urn:GoogleSearch">
  <key xsi:type="xsd:string">0000</key>
  <url xsi:type="xsd:string">http://www.google.com/</url>
</gws:doGetCachedPage>
```

Fig. 6. A request message of doGetCachedPage

In the rest of this section, we explain the details of our architecture. In the next subsection, we explain how we specify requests from a Web Service. We continue by discussing synchronous/lazy/asynchronous update modes and our cache mechanism. Finally, we discuss response transformations that help user programs be independent of the details of the Web Service interfaces.

2.1 Querying Services

In order to request a Web Service, we have to prepare a SOAP message from the request data, which we call a *query* in our framework (Fig. 5 (1)). That is, instead of invoking a method, we do a query to request a service.

We use the following steps to request a service: (a) create a SOAP body part for the query; (b) construct a SOAP envelope from the service specification and make a SOAP message by putting the body in the envelope; (c) select an endpoint for the service and send the SOAP message to it.

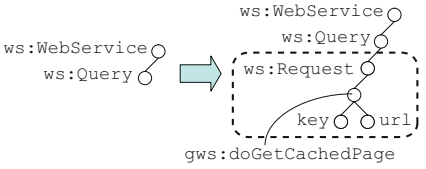

Fig. 7. The request message in the DOM. The `ws:WebService`, `ws:Query`, and `ws:Request` elements are introduced just for bundling messages in the DOM, where `ws` is a prefix of the reserved namespace for Web Service.

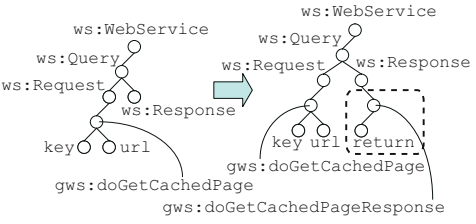

Fig. 8. From the `ws:Response` element, we obtain the response message from the Web Service

Another approach is to use a string to specify a query. Considering the WWW architecture, all resources are specified by a URI, which is just one string, which greatly contributes to simplifying the interface of the WWW. If we can describe a request in one string, our query model is also simplified as well.

For this purpose, we introduce an XPath query model. Instead of directly constructing an XML message, we specify an XPath expression that can be interpreted as a request message. Let us consider again the example shown in Fig. 6. In this example, the essential information to request the service is that the key is `0000` and the URL is `http://www.google.com/`. Therefore, we can specify these two in the following XPath expression.

```
doGetCachedPage[key='0000' and url='http://www.google.com/']
```

By extracting the type information from the WSDL definition, we can construct the request message in Fig. 6 from the above expression. That is, "`xsi:type`" attributes are automatically added by the WSDL definition.

More formally, this process requires translations from an XPath expression to an XML tree, which involves some challenging issues. Later, we explain our sample implementation in Subsection 3.2, and discuss the advanced topics in Section 5. Here, suffice it to say that we can convert some XPath expressions into XML trees, as long as they conform to the following restrictions:

- Every name test has a concrete QName (e.g., `*` or `//` is not allowed).
- All predicates are of the form:

$$[\mathit{PathExpr} = \mathit{Literal} \text{ and } \ldots \text{and } \mathit{PathExpr} = \mathit{Literal}],$$

  where the notations *PathExpr* and *Literal* are as defined in the XPath specification [8].

For Steps (b) and (c), since the service endpoints are described in WSDL, we can automatically generate a SOAP envelope to transfer the data, and then send the SOAP message to the specified endpoint.

2.2 Update Mode

Since SOAP communication takes considerable time, it is critical for applications to determine when we should send messages and when we should wait for the response. In our framework, we have the following three update modes for the timing of requests, which determines when we process requests for services (see Fig. 9).

Synchronous. Start the request when the request part is prepared in the DOM tree and wait until it finishes.

Lazy. Defer the request until the response part is accessed.

Asynchronous. Start the request when the request part is prepared in the DOM tree but do not wait for the completion at that time. When the response part is accessed, block the execution until the response is available.

Let us explain the differences of these three update modes by using examples. We suppose that we have just put a request message in the DOM tree (Step (1) in Fig. 5). In synchronous mode, we call the `send(element)` API, where `element` points at the `ws:Request` element in Fig. 7. Then the execution is blocked until the `ws:Response` part in the DOM (Fig. 8) is updated with the response message. In lazy mode, we do not have to explicitly call `send(element)`. Instead, we can directly access the `ws:Response` element. At that time, if we have not received the response message, the access is blocked until the `ws:Response` part is updated. In asynchronous mode, we explicitly call `send(element)`. But the execution is not blocked at that time. Then when we access the `ws:Response` element and if we have not received the response message, the access will be blocked as in the lazy mode.

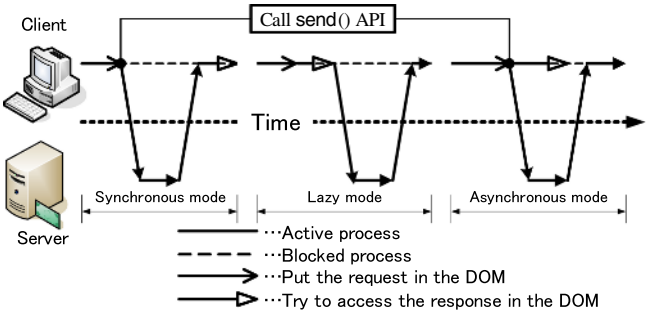

Fig. 9. The interaction patterns of the three update modes

Of these approaches, the lazy update has a clear merit for the simplicity of the programming model, because we do not have to use any extra API call such as `send()`. Therefore, the lazy update is the default mode in our framework.

For the other options, we have to explicitly specify when the request part in the DOM has been prepared. However, the asynchronous update is the preferred option considering the users' experience and is aligned with the Ajax style.

2.3 Cache Mechanism

Since the numbers of request and response messages are unlimited, we naturally require cache mechanisms in our framework, which stores request and response trees and evicts entries appropriately when the total cache size reaches the set limit.

We cache the response data associated with the corresponding request data. That means we assume the same request data always returns the same response data. This assumption is justified because a Web Service is usually designed in a stateless fashion, i.e., a request message contains all of the information required to invoke a service.

Note that we do require a "pin" mechanism for our cache system for lazy or asynchronous update modes. We have to pin the request data to prevent its eviction until the updating operation for the response data has been completed.

2.4 Response Transformation

The raw response data from a Web Service does not usually fit the requirements for browsing. In such cases, transforming the response data is desirable for client-side programs. This style agrees with separation of concerns. Ideally, the client-side programs concentrate on presentation issues and delegate the other parts to the transformation program. In addition, this architectural style is robust against changes of the Web Service interfaces. The design goal is that we will only have to update the transformation programs and that such changes will not affect the client user programs.

XSLT and XQuery are good candidates for performing this kind of XML transformation. In configurations of our client framework, we can specify these languages for each service. If we can apply different transformations to different locations in the DOM tree, it may be helpful for various presentations. For example, we may want to present stock prices differently in text and in a table, and this mechanism is convenient in such a case. The client framework automatically applies the specified transformations to the response messages before storing them in the DOM tree.

Otherwise, as an alternative design choice, we could apply such transformations to the entire DOM tree. This choice may be convenient for tightly integrated presentations, since each service query can affect the whole presentation. However, we have to carefully organize the transformations and user programs in order to avoid conflicts with each other.

3 Implementation

We implemented a Web Service framework integrated with Web browsers, *WebDrasil*, in accord with the architecture described in Section 2. In Fig. 10, we describe the components of WebDrasil. Our WebDrasil has two DOM trees: 1) a Web Service DOM representing the Web Service request and response messages, which is constructed using client user programs written in JavaScript via standard APIs and which is transparently updated with response messages; and 2) an HTML DOM representing an HTML document, which is provided by a special Web browser supporting the W3C DOM programming [9].

3.1 Applications on WebDrasil

A DHTML application supporting a Web Service is executed by following these steps (see Fig. 10): (1) WebDrasil sends the URI to the server to retrieve the application document; (2) The corresponding document is received from the server. The HTML DOM is created by parsing the HTML part of the retrieved document, and the JavaScript engine loads the received JavaScript code; (3) WebDrasil renders the constructed HTML DOM; (4), (5) If any event caused by user input is detected, the JavaScript function associated with that event is called; (6) To use a Web Service API, the Web Service DOM is updated by a JavaScript function using the DOM APIs, or by XPath functions; (7) When updating a Web Service DOM, a SOAP request is created and sent to the Web Service server; (8) Then a SOAP response message is sent to WebDrasil and the Web Service DOM is updated by examining the SOAP response message; (9),(10) The elements of the Web Service DOM that are necessary for updating the HTML DOM are retrieved, and the HTML DOM is updated based on them; (11) Finally WebDrasil renders the updated HTML DOM, and dynamically presents the changes.

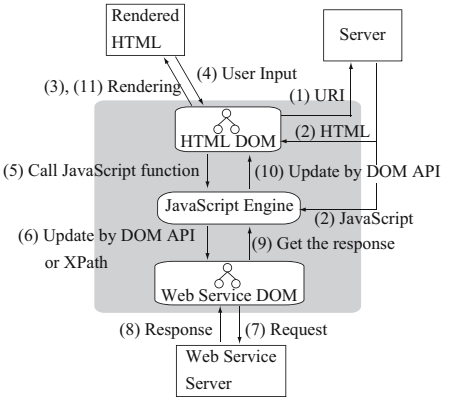

Fig. 10. The components of WebDrasil

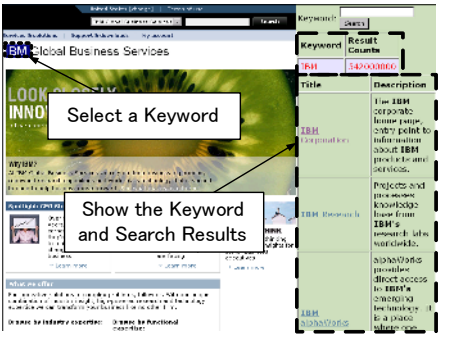

Fig. 11. Mashed-up Google Web Search

3.2 Evaluation of XPath

In order to query services with XPath, WebDrasil has a special XPath interpreter. When our interpreter successfully matches the given XPath expression with the DOM tree, it simply returns the matched part. Otherwise, our interpreter splits the XPath expression into location steps. For each location step, if the current context position is in a `ws:Request` element, our interpreter checks whether or not the matched node exists. Note that there is at most one matched node when the XPath expression obeys the restrictions in Subsection 2.1. If it does not exist, we create a new node from this location step. After evaluating all of the steps, we have an updated DOM tree that has a node that matches the given XPath expression.

We need to maintain the validity of the Web Service DOM after this update process. Our interpreter checks the validity of the updated DOM tree by using the schemas in the WSDL definitions. If it is not valid, our interpreter cancels the entire update.

3.3 JavaScript Examples for a Web Service

In Table 1, we show examples of JavaScript code for updating the Web Service DOM and invoking a Web Service. Even though these code samples include all of the essential steps to use the Web Service, they are written by using the XPath APIs, without using the service-specific interfaces.

The JavaScript code for updating `ws:Request` element by using the XPath API is Example 1 in Table 1. The request is created in the Web Service DOM as shown in Fig. 7. The `ws:WebService` is the root node of the Web Service DOM, and has the URL for the WSDL file as its attribute. The WSDL file is used for checking whether or not the DOM tree is valid. A node like `ws:Query` represents each query of the Web Service. Such a node has two children, an element for generating a SOAP request, and an element for storing a SOAP response corresponding to the request element. The node `ws:Request` holds the request as converted to a SOAP request message. The content of this element is the same as the body of the corresponding SOAP request message.

We can obtain the result of the Google Web Service request by using the code shown in Example 2 in Table 1, which uses only the XPath API. By this code, the framework transparently sends the SOAP request message, and changes the Web Service DOM as shown in Fig. 8, so that `ws:Response` represents the SOAP response. The content of this element is the same as the body of the corresponding SOAP response element.

Table 1. Examples of JavaScript codes for handling Web Service DOM, where `ws` and `gws` are prefixes of the namespaces for Web Service DOM and for Google Web Service, respectively; and webService, response are variables of the Web Service DOM, and the `Response` element, respectively

Example 1 : Update *Request element* using XPath	var doGetCachedPage =webService.selectSingleNode("./ws:Query[1]/ws:Request/gws:doGetCachedPage [key='0000' and url='http://www.ibm.com']", webService);
Example 2 : Get the response using XPath	var textNode = response.selectSingleNode("./gws:doGetCachedPageResponse/return[1]/text()[1]", response);

3.4 Application: Google Search Web Service Composed with Another Website

The DHTML application in Fig. 11 shows the benefits of using WebDrasil. This application has two visual components: (1) a web page, from which we can extract keywords, and (2) tables for showing the search history of keywords and the detailed search results. When the user selects a keyword from the web page, the search for the keyword is done with Google Web Service, and then the corresponding detailed results are shown in the detailed result table.

In this application, WebDrasil holds the response messages of the Google Web Service in the Web Service DOM. Because of this, the browser can change the presentation

dynamically by updating or retrieving the Web Service DOM according to the user's input. This application is written using only HTML and JavaScript so that we can easily mash-up it with other websites as shown in Fig. 11.

By using the DOM and XPath APIs, Web developers can make smart and interactive Web Service DHTML applications such as this example.

4 Related Work

Our framework is built on top of various important studies in such areas as Web browser integration, Web and Web Service programming models, lazy DOM processing, and asynchronous interactions.

Web browsers are now such widespread components that integration with Web browsers is of great interest in various fields such as interactive programming environments [10] and collaboration tools [11]. Ponzo and Gruber integrated Web browser technologies with a rich client platform, Eclipse [9], for a better programming model. JSON (JavaScipt Object Notation)-RPC [12] is another service invocation method on Web browser. But it uses another data format familiar to JavaScript instead of XML.

There are some studies on Web Service programming models based on document processing. ActiveXML [13] introduced *dynamic XML documents* which consist of explicitly specified data and dynamic portions to be changed by querying Web services. Our work is similar to ActiveXML in that we use dynamically changing DOM. However, ActiveXML also proposed new query language and programming models. In contrast, we stick to use DOM and XPath fitting well with the DHTML programming model to integrate with Web browsers. Fox *et al.* proposed a collaborative Web Service [14] and Qui *et al.* proposed a collaboration framework using W3C DOM on top of their Web Service [15]. Their approach is somewhat similar to ours in that we use DOM for applications, but their framework is designed for collaboration by using the Web Service architecture, and does not provide a general Web Service programming model.

Asynchronous update mode involves asynchronous Web Service interactions. The correlation and coordination issues of asynchronous Web Services are studied in [16].

5 Conclusions and Future Work

We proposed a new client programming model for Web Services, a model which is document-centric and similar to that of Web browsers. In this programming model, we manipulate the document tree with uniform APIs such as DOM and XPath rather than by explicitly sending messages or by invoking non-uniform APIs. We prototyped a new Web Service client framework, WebDrasil, based on this architecture and provided some examples of client programs to show that the approach is intelligible and natural to Web developers.

Since this new programming model is based on many elementary technologies such as XML processing and distributed systems, we still have a lot of work to do:

XPath Query Model
In Subsection 2.1, we offered a rudimentary XPath-based query method for Web Services. However, we would like to have a more convenient and complete model for this purpose, one that would allow us to statically check or supplement XPath queries while considering the constraints of schemas. For example, if a schema specifies only one "key" element is allowed in the request message, then any XPath query for this service must contain a predicate only for that "key" element. We think match-identifying tree automata [17] could be used for this model.

Server-side Approach
In this paper, we introduced our framework on the client side of the Web applications, specifically in the Web browsers. However, our framework could also be applicable to the server side. For example, we could build a Web-Service-gateway server which feeds DHTML applications to the Web browsers and accepts requests from the browsers, and which then appropriately forwards them to external SOAP Web Service servers. By using such a gateway, Web browsers without support for SOAP Web Services could access those services. Also, in such a gateway, our framework would work well because the Web browsers would present XML documents rather than JavaScript objects.

Web Service Coordination and Security
When we use multiple services in our client framework, we need some coordination and security framework for reliable communications. Since, in our framework, we store the responses in one DOM tree, we require transactional cache mechanisms to maintain consistency and security mechanisms to prevent illegal accesses for DOM tree processing. Since our framework allows client-side scripting, such an access control mechanism is important to prevent XSS (Cross-Site Scripting) attacks.

Acknowledgments

We warmly thank to Makoto MURATA for helpful advises to improve this paper. This research was partly supported by the National Institute of Information and Communications Technology (NICT) of Japan as a part of the Multimedia Browsing Project for People with Visual Impairments.

References

1. J. Whatcott, "SOA's next wave: Service-oriented clients," May 2006, http://www.cio.com/weighin/column.html?CID=21201.
2. H. Dhurvasula and M. Galli, "Mozilla and web services," http://www.mozilla.org/projects/webservices/.
3. T. O'Reilly, "What is web 2.0," September 2005, http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html.
4. J. J. Garrett, "Ajax: A new approach to Web applications," February 2005, http://www.adaptivepath.com/publications /essays/archives/000385.php.
5. "Java api for xml web services (JAX-WS)," 2005, http://java.sun.com/webservices/jaxws/index.jsp.

6. K. Ballinger, D. Ehnebuske, M. Gudgin, C. K. Liu, M. Nottingham, and P. Yendluri, "WS-I basic profile version 1.1," http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html.
7. S. Loughran and E. Smith, "Rethinking the Java SOAP stack." in *ICWS*, 2005, pp. 845–852.
8. J. Clark and S. DeRose, "XML Path Language (XPath) Version 1.0," w3C Recommendation 16 November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116.
9. J. Ponzo and O. Gruber, "Integrating Web technologies in Eclipse," *IBM Systems Journal*, vol. 44, no. 2, pp. 279–288, 2005.
10. M. Jambalsuren and Z. Cheng, "An interactive programming environment for enhancing learning performance," in *Databases in Networked Information Systems*, 2002, pp. 201–212.
11. K. M. Anderson and N. O. Bouvin, "Supporting project awareness on the www with the iscent framework," *SIGGROUP Bull.*, vol. 21, no. 3, pp. 16–20, 2000.
12. R. Barcia, "Build enterprise soa ajax clients with the dojo toolkit and json-rpc," http://www-128.ibm.com/developerworks/websphere/library/techarticles/0606_barcia/0606_barcia.html.
13. S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo, "Dynamic XML documents with distribution and replication," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003, pp. 527–538.
14. G. Fox, H. Bulut, K. Kim, S.-H. Ko, S. Lee, S. Oh, S. Pallickara, X. Qiu, A. Uyar, M. Wang, and W. Wu, "Collaborative web services and peer-to-peer girds," in *Collaborative Technologies Symposium*, 2003.
15. X. Qiu, B. Carpenter, and G. Fox, "Internet collaboration using the w3c document object model." in *International Conference on Internet Computing*, 2003, pp. 643–647.
16. M. Brambilla, S. Ceri, M. Passamani, and A. Riccio, "Managing asynchronous web services interactions." in *ICWS*, 2004, pp. 80–87.
17. M. MURATA, "Extended path expressions for XML," in *PODS*, 2001, pp. 126–137.