August 10, 2007

RT0747

Engineering Technology; Network 10 pages

Research Report

Efficient Web Services Message Exchange by SOAP Bundling Framework

Toshiro Takase, Keishi Tajima

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Efficient Web Services Message Exchange by SOAP Bundling Framework

Toshiro Takase[1, 2], Keishi Tajima[1]

[1]*Department of Social Informatics, Kyoto University,*
Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan

[2]*IBM Research, Tokyo Research Laboratory,*
1623-14, Shimotsuruma, Yamato, Kanagawa, 242-8502, Japan

e30809@jp.ibm.com, tajima@i.kyoto-u.ac.jp

Abstract

Web services use an interoperable and loosely coupled data exchange architecture. Web services processing can cause significant runtime overhead, especially when the number of fine-grained transactions becomes large. Although some best-practice guidelines recommend coarse-grained messages to improve the performance of Web services, coarse-grained services may interfere with the componentization of the services. Service granularity should be designed for reusability and modularity. In this paper, we propose a SOAP message bundling framework. This framework enables bundling multiple messages into one message. With this framework, application developers do not have to consider the service granularity. Instead, the framework bundles some fine-grained messages into a single coarse-grained message. To support this framework, we provide for service providers (1) a WSDL conversion tool and (2) a skeleton wrapper generator. These tools let service providers receive bundled messages without modifying existing service implementations. We also provide (3) a stub wrapper generator that allows service requesters to use bundled services easily. The existing message exchanges are not influenced by this framework. We evaluated the performance gain in experiments using the Google SOAP API. The results showed that our approach improves the performance of Web services.

1. Introduction

Currently, Web services are spreading widely throughout the world. Web services are an enabling technology for interoperability within distributed, loosely coupled, and heterogeneous computing environments. However, previous studies have shown that the performances of Web services are relatively poor because of the encoding processing [1, 4] and network latency [2, 3, 5]. Therefore, technologies to improve the performance of Web services are needed. In this paper, we describe a message bundling framework appropriate for Web services architectures.

The most important point of Web services is interoperability. Web services are based on certain specifications, such as XML [8] as a message format, SOAP [9] as a message layer protocol, and WSDL [10] as a description of the interfaces. Once a service application is developed as a Web service, the interface description can be published for external partner companies and the service can easily be provided to new partner companies.

A stock quote service is often used as an example Web service. A typical stock quote service gets an input ticker symbol, and then returns a stock price. In practice, it is reported that exchanging many fine-grained messages of this type often causes performance problems [2, 3]. Listing 1 shows an example of request and response message for a stock quote service. These messages are selected from the examples in the SOAP specification. The important data is only the ticker symbol, “DIS”, for the request message, and the stock price, “34.5”, for the response message. SOAP messages have many redundant parts. Therefore some best practice guidelines recommend using coarse-grained services to avoid large numbers of transactions [16, 17, 18]. However, some services may be hard to build using coarse-grained services. The granularity of the service interface should be designed based on the granularity of the service componentization. Fine-grained services are easy to understand and easy for service requester to use. Such fine-grained services have high reusability and modularity.

Listing 1. Examples of fine-grained messages

```
Request message:

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePrice xmlns:m="Some-URI">
           <symbol>DIS</symbol>
       </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

Response message:

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
       <m:GetLastTradePriceResponse xmlns:m="Some-URI">
           <Price>34.5</Price>
       </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this paper we propose a SOAP message bundling framework. This framework enables bundling multiple messages into one message. With this bundling framework, application developers do not have to consider the service granularity for performance reasons. Instead, the bundling framework can be applied to the developed services afterward, and then the framework bundles some fine-grained messages into a single coarse-grained message. That is to say, the application developers can design their services with fine-grained services at development time, though the actual messages will be bundled by the framework at run time. Using this bundling framework, the total message size, runtime overhead such as parsing time, and the number of message exchanges will be reduced.

Today, there are many existing Web services using standardized specifications such as SOAP. One of the advantages of Web services is loosely coupled bindings. Some other research [19, 20] proposes performance improvements for Web services by using tightly coupled bindings. Such techniques go beyond the SOAP specifications to some extent. In this paper, however, we do not define any new specifications and thus avoid breaking any existing services. Instead, we only introduce new tools for a SOAP message bundling framework. These tools can be applied to existing services.

The rest of this paper is structured as follows: First, we describe the design of our SOAP message bundling framework in Section 2. Section 3 explains in detail our prototype implementation using the Apache-Axis SOAP engine [14] and examples for the Google SOAP API [12]. Section 4 shows the results of experiments measuring various kinds of overhead for various numbers of requests per bundle. We discuss related work in Section 5 and future work in Section 6. Finally, in Section 7 we conclude the paper.

2. Architecture of the bundling framework

In this section, we describe the design of our SOAP message bundling framework. In our framework, we assume that an existing service implementation already exists. Our framework does not break any existing services but just adds bundled service interfaces. Also, our prototype implementation described in Section 3 is based on the Java Apache-Axis platform, but this framework is not limited to that target platform.

2.1. Design and usage scenario

Listing 2 shows an example of the original fine-grained messages and an example of the bundled messages. These messages are messages for the Google SOAP API which is used in our performance evaluation in Section 4. The bundled message includes three fine-grained messages. The bundled message is more efficient than three separate fine-grained messages in terms of message size, parsing time, and the number of message exchanges. Although these messages do not have SOAP headers in this example, if SOAP headers for routing or security are used, the bundled message is much more efficient than the fine-grained messages.

Listing 2. An example of bundled message

```
Original fine-grained message:

<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:doGetCachedPage
     soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:ns1="urn:GoogleSearch">
      <key xsi:type="xsd:string">00000000000000000000000000000000</key>
      <url xsi:type="xsd:string">http://www.google.com/</url>
    </ns1:doGetCachedPage>
  </soapenv:Body>
</soapenv:Envelope>

Bundled message:

<soapenv:Envelope
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <ns1:doBundledService
     soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:ns1="urn:GoogleSearch">
      <doGetCachedPages soapenc:arrayType="ns1:DoGetCachedPage[3]"
       xsi:type="soapenc:Array"
       xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        <item xsi:type="ns1:DoGetCachedPage">
         <key xsi:type="xsd:string">00000000000000000000000000000000</key>
         <url xsi:type="xsd:string">http://www.google.com/</url>
        </item>
        <item xsi:type="ns1:DoGetCachedPage">
         <key xsi:type="xsd:string">00000000000000000000000000000000</key>
         <url xsi:type="xsd:string">http://www.yahoo.com/</url>
        </item>
        <item xsi:type="ns1:DoGetCachedPage">
         <key xsi:type="xsd:string">00000000000000000000000000000000</key>
         <url xsi:type="xsd:string">http://www.ibm.com/</url>
        </item>
      </doGetCachedPages>
    </ns1:doBundledService>
  </soapenv:Body>
</soapenv:Envelope>
```

In this example, the bundled message has only one array of `doGetCachedPage` operation. Our bundling framework generates new operations for all combinations of the original operations in the WSDL. For example, the Google SOAP API has three operations, so the framework generates 2^3-1 operations. If an original WSDL has N operations, 2^N-1 operations are generated. In this way, we can bundle any combination of the operations. Although we describe our prototype implementation in Section 3, we explain the details of only an operation which includes all of the operations in the original WSDL. The other combinations of the operations are omitted because they can be explained in a similar way.

We assume that some Web services transactions are already running. Existing service providers can employ this bundling framework by using our WSDL converter and skeleton wrapper generator. They do not have to re-implement their service application. New service providers do not have to avoid fine-grained services for performance reasons. They can design an application interface based on the componentization of the service. They can apply this bundling framework to the services afterwards.

Existing service requesters have to change their application a little to take advantage of the bundled operations. By using our stub wrapper generator, the changes are simplified. New requesters are free to develop applications by using the bundled operations.

Scenarios with the highest potential for benefit involve transactions between a portal site as the service requester and a primitive service provider, such as a weather forecast or a search service. Figure 1 shows this scenario.

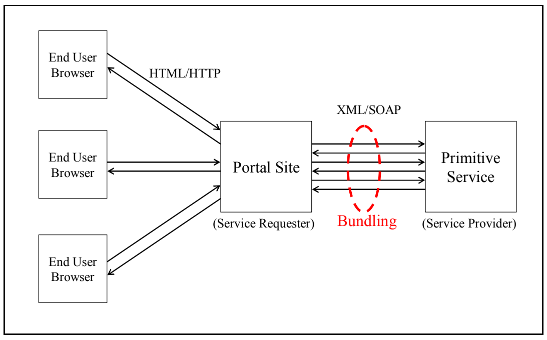

Figure 1. An example of a usage scenario

For example, a mash-up site might use a zip-code search and a baggage tracking service. The site can receive unpredictable numbers of requests at any time. In this situation, many very small messages are exchanged between the site and the primitive service provider. If this bundling framework is applied, the mash-up site as a service requester can bundle messages from many different end users' requests. Also, the mush-up site could implement bundling the messages received in each 0.1-second time interval. Thus the user experience will be little affected but more requests can be processed.

2.2. Service providers' role

Originally, the role of a Web services provider is to provide a service and to publish a service description using WSDL. In our framework, we assume that an original service implementation already exists. The granularity of that service should be designed based on the service componentization.

In this framework, we provide a WSDL converter and a skeleton wrapper generator for service providers. The WSDL converter adds new bundled service interfaces to an existing WSDL document. The added bundled service interface bundles all of the existing service interfaces in the WSDL document. Specifically, a bundled operation is added. Here, "operation" is a WSDL term, but generally this has the same meaning as function or method. The input parameters for the bundled operation include parameters for the arbitrary number of operations which exist in the service. The return value of the bundled operation can also hold arbitrary numbers of return values for all existing operations. For example, Google SOAP API has three operations, `doGoogleSearch`, `doSpellingSuggestion`, and `doGetCachedPage`. The new operation, `doBundled-Operation`, created by this converter can bundle any number of operations of these three types. For details, we will describe this conversion in Section 3.1.

A skeleton wrapper generated by our skeleton wrapper generator implements the added operation for the WSDL converter. In the most naïve implementation, the skeleton wrapper sequentially calls the existing operations internally. If the platform this skeleton is deployed on offers advantages for multi-thread operations, as in a multi-core processor environment, the implementation should exploit this. However, even in the most naïve implementation, our bundling approach improves the performance. Therefore, we only discuss the naïve implementation here. The naïve implementation is described in Section 3.2. In any case, the skeleton wrapper provides an implementation for the bundled operation by using the existing operations.

The service provider's steps are: (1) Convert the existing WSDL document using the WSDL converter and re-publish the converted WSDL. (2) Generate the skeleton wrapper with the skeleton wrapper generator and deploy the generated skeleton wrapper into the service platform. Figure 2 shows this architecture of the SOAP message bundling.

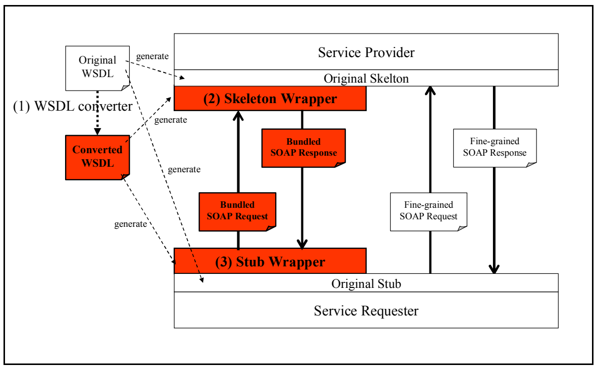

Figure 2. Architecture of the bundling framework

2.3. Service requesters' role

Once the WSDL modified by the service provider is re-published, service requesters can use the added bundled operation. All existing service requester applications can also continue to use the original services, because the re-published WSDL is just adding new operations. If a service requester uses the bundled operation, improved performance is expected. To use the bundled service, a service requester has to modify the application. The input parameter set for the added bundled operation is not easy to use, so a stub wrapper generator has been provided for the convenience of service requesters.

The stub wrapper generator provides a stub wrapper implementation with a "holder" to use the bundled operation. The holder can hold arbitrary numbers of input parameters for all of the original operations. The service requester can sequentially add request parameters into the stub holder, and after that, at the proper time, the service requester can order the stub wrapper to send a bundled request to the service provider. This stub implementation is described in Section 3.3. This stub is similar to a Stored Procedure for a DB. We discuss the differences in their behaviors in Section 5.

3. Implementation

In this section, we explain our implementation using the Apache-Axis SOAP engine. The Google SOAP API is used as a target service. The Google SOAP API is an rpc/encoded style service, so this example shows behavior for rpc/encoded style service. However, the WS-I Basic Profile [11] currently recommends document/literal style services for better interoperability. Although this example is rpc/encoded style, our implementation would be also applied to document/literal style services in the same way.

The WSDL of the Google SOAP API is used as the original WSDL document. This WSDL is available at the Google website. Apache Axis has a WSDL2Java tool. First, we generated the original skeleton and stub from the Google WSDL document by using the WSDL2Java tool. Here, we assume that the original implementation already exists for both a service provider and a service requester. Then we converted the Google WSDL document with our WSDL converter. The converted WSDL has an added bundled operation. After that, we generated the skeleton wrapper and stub wrapper to use the new bundled operation. Our skeleton and stub wrapper generator is implemented for a Java skeleton and stab based on Apache Axis. Although currently our generators can be applied only to an Apache-Axis-based service, it is possible to implement these generators for other platforms.

3.1. WSDL converter

Listing 3 shows the original WSDL of the Google SOAP API. Some parts were omitted. Our WSDL converter adds the bundled operations.

Listing 3. Original WSDL of Google SOAP API

```
<?xml version="1.0"?>
<definitions name="GoogleSearch"
             targetNamespace="urn:GoogleSearch"
             .....>
  <types>
    <xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
                targetNamespace="urn:GoogleSearch">
      <xsd:complexType name="GoogleSearchResult">
        <xsd:all>
          <xsd:element name="searchQuery" type="xsd:string"/>
          .....
        </xsd:all>
      </xsd:complexType>
      .....
    </xsd:schema>
  </types>

  <message name="doGoogleSearch">
    <part name="key"   type="xsd:string"/>
    <part name="q"     type="xsd:string"/>
    <part name="start" type="xsd:int"/>
    .....
  </message>

  <message name="doGoogleSearchResponse">
    <part name="return" type="typens:GoogleSearchResult"/>
  </message>

  <portType name="GoogleSearchPort">
    <operation name="doGoogleSearch">
      <input message="typens:doGoogleSearch"/>
      <output message="typens:doGoogleSearchResponse"/>
    </operation>
    .....
  </portType>

  <binding name="GoogleSearchBinding" type="typens:GoogleSearchPort">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="doGoogleSearch">
      <soap:operation soapAction="urn:GoogleSearchAction"/>
      <input>
        <soap:body use="encoded" namespace="urn:GoogleSearch"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded" namespace="urn:GoogleSearch"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
    .....
  </binding>

  <service name="GoogleSearchService">
    <port name="GoogleSearchPort" binding="typens:GoogleSearchBinding">
      <soap:address location="http://api.google.com/search/beta2"/>
    </port>
  </service>
</definitions>
```

Listing 4 and Listing 5 show the description of the added operation, `doBundledOperation`, in <binding> and <portType>. The added operation description in <binding> is almost same as the other operation except for its name. The added operation in <portType> defines the new input and output messages, `doBundledOperation` and `doBundledOperation-Response` respectively. Message refers to the input parameters and return values in WSDL terms.

Listing 4. Added bundled operation in <binding>

```
<operation name="doBundledOperation">
  <soap:operation soapAction="urn:GoogleSearchAction"/>
  <input>
    <soap:body use="encoded" namespace="urn:GoogleSearch"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  </input>
  <output>
    <soap:body use="encoded" namespace="urn:GoogleSearch"
           encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
  </output>
</operation>
```

Listing 5. Added bundled operation in <portType>

```
<operation name="doBundledOperation">
  <input message="typens:doBundledOperation"/>
  <output message="typens:doBundledOperationResponse"/>
</operation>
```

Listing 6 shows the description of the new added input and output messages, `doBundledOperation` and `doBundledOperationResponse`. The input message includes three types of data, `DoCachedPageArray`, `DoSpellingSuggestionArray`, and `DoGoogleSearch-Array`. These arrays are sets of input parameters for the three original operations, `doGetCachedPage`, `doSpellingSuggestion`, and `doGoogleSearch`. The output message also represents arrays of the return values of the three original operations, but for output messages, the data are put together into the `BundledResult` type.

Listing 6. Added bundled input/output <message>

```
<message name="doBundledOperation">
  <part name="doGetCachedPages"
        type="typens:DoGetCachedPageArray"/>
  <part name="doSpellingSuggestions"
        type="typens:DoSpellingSuggestionArray"/>
  <part name="doGoogleSearchs"
        type="typens:DoGoogleSearchArray"/>
</message>

<message name="doBundledOperationResponse">
  <part name="return" type="typens:BundledResult"/>
</message>
```

Listing 7 shows some of the added type descriptions. Some added types are omitted. One of the types in the added input message is `DoGoogleSearchArray`, which is an array of `DoGoogleSearch` type. The `DoGoogleSearch` type is a set of input parameters for the original `doGoogleSearch` operation. For the other types in the input message, `DoCachedPageArray`, and `DoSpellingSuggestionArray`, their definitions are described in the same way.

`BundledResult` is the type of the added output message. The `BundledResult` type includes three types of data. These data represent an array of the set of return values of the three original operations. The rest of the data structure for return values is the same as for the input parameters. In this way, any types defined in WSDL can be wrapped and the bundled operation can be defined.

Listing 7. Added complexType schemas in <types>

```
<xsd:complexType name="DoGoogleSearchArray">
  <xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
       <xsd:attribute ref="soapenc:arrayType"
                      wsdl:arrayType="typens:DoGoogleSearch[]"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DoGoogleSearch">
  <xsd:all>
    <xsd:element name="key" type="xsd:string"/>
    <xsd:element name="q" type="xsd:string"/>
    <xsd:element name="start" type="xsd:int"/>
    .....
  </xsd:all>
</xsd:complexType>
.....

<xsd:complexType name="BundledResult">
  <xsd:all>
    <xsd:element name="doGetCachedPageResults"
                 type="typens:XSD_base64Binary_Array"/>
    <xsd:element name="doSpellingSuggestionResults"
                 type="typens:XSD_string_Array"/>
    <xsd:element name="doGoogleSearchResults"
                 type="typens:GoogleSearchResultArray"/>
  </xsd:all>
</xsd:complexType>

<xsd:complexType name="GoogleSearchResultArray">
  <xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
       <xsd:attribute ref="soapenc:arrayType"
                      wsdl:arrayType="typens:GoogleSearchResult[]"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
.....
```

For the Google example, each return value is wrapped as one data item. This is not a SOAP or WSDL limitation. This is just because a normal Java method has one return value. However, JAX-WS [13] defines how to support multiple return values in Java by using holder objects. Even if a return value has multiple data values, we can define the bundled operations in the same manner.

3.2. Skeleton wrapper generator

Listing 8 shows the Java interface generated from the original WSDL of the Google SOAP API, using the Apache-Axis WSDL2Java tool. The methods in the interface are fine-grained. For componentization, this interface is easy to reuse and modularize. However, from the performance perspective, this interface is inefficient. Listing 9 shows the new method corresponding to the added bundled operation. The method can handle multiple requests for arbitrary original methods at the same time. This can improve the performance.

Listing 8. Generated interface from original WSDL

```
public interface GoogleSearchPort {

    public byte[] doGetCachedPage(String key, String url);
    public String doSpellingSuggestion(String key, String phrase);
    public GoogleSearchResult doGoogleSearch(String key, String q,
                                              int start, .....);
}
```

Listing 9. Added method with converted WSDL

```
    public BundledResult doBundledOperation(
                DoGetCachedPage[] doGetCachedPages,
                DoSpellingSuggestion[] doSpellingSuggestions,
                DoGoogleSearch[] doGoogleSearchs);
```

The provider of an existing service can semi-automatically generate an implementation of the bundled operation by using our skeleton wrapper generator. Listing 10 is a skeleton wrapper generated this way. The wrapper implements the added method shown in Listing 9, `doBundledOperation`. This implementation uses a naïve approach. The implementation just calls the original methods sequentially. In other words, it retrieves each input parameter from the bundled input parameters, calls each original method sequentially, then puts the results together into a `BundledResult` object, and finally, returns the bundled result. Although Listing 10 does not include the code for `doGetCachedPage` and `doSpellingSuggestion`, the code is almost same as the code for `doGoogleSearch`.

Listing 10. Generated skeleton wrapper code

```
public class GoogleSearchBindingSkeleton implements GoogleSearchPort{
    public BundledResult doBundledOperation(
                DoGetCachedPage[] doGetCachedPages,
                DoSpellingSuggestion[] doSpellingSuggestions,
                DoGoogleSearch[] doGoogleSearchs) {

        GoogleSearchResult[] doGoogleSearchResults =
             new GoogleSearchResult[doGoogleSearchs.length];
        for (int i=0; i<doGoogleSearchs.length; i++) {
            String key = doGoogleSearchs[i].getKey();
            String q = doGoogleSearchs[i].getQ();
            int start = doGoogleSearchs[i].getStart();
            .....
            doGoogleSearchResults[i] = doGoogleSearch(key, q, start, ...);
        }
        //..... call doGetCachedPage(s) and doSpellingSuggestion(s)

        BundledResult bundledResult = new BundledResult();
        bundledResult.setDoGoogleSearchResults(doGoogleSearchResults);
        //..... set doGetCachedPageResults and doSpellingSuggestionResults

        return bundledResult;
    }
    //..... the code for original methods
}
```

The code in Listing 10 is naïve, so the code does not include error handling, etc. In particular, in a case where only one method in a bundle failed or only one method spent too much time, the naïve implementation would perform poorly. There are various techniques to address such problems. We discuss these problems as future work in Section 6.

3.3. Stub wrapper generator

On the service-requester side, the Java interface generated from the original WSDL is the same as the provider's, as shown in Listing 8. An existing service requester already has an implementation based on this original interface. The added bundled method in the converted WSDL is also the same as on the service provider's side, as shown in Listing 9. Service requesters can by themselves migrate to an implementation using the added operation. However if the requesters use our stub wrapper generator, the migration is easier.

Listing 11 shows methods included in our stub wrapper code. The first three methods, `addXXX`, add the parameter of each original operation into the internal holder in the stub wrapper. Each input parameter is exactly the same as the original one. When the method `executeBundledOperation` is called, the contained operations are invoked by using the added bundled method, `doBundledOperation`, in Listing 9. After the invocation, the return values for each original operation can be obtained using the last three methods, `getXXXResult`, in Listing 11. The types of the return values are the same as the original types.

Listing 11. Methods included in stub wrapper code

```
    public void addDoGetCachedPage(String key, String url);
    public void addDoSpellingSuggestion(String key, String phrase);
    public void addDoGoogleSearch(String key, String q, int start, .....);
    public void reset();

    public BundledResult executeBundledOperation();

    public byte[] getDoGetCachedPageResult(int index);
    public String getDoSpellingSuggestionResult(int index);
    public GoogleSearchResult getDoGoogleSearchResult(int index);
```

With this stub wrapper, existing service requesters can replace the original methods with the `addXXX` methods in this wrapper. At the proper time, a requester can then call the execute method, and get the results.

In some cases, the logic of the requester application may make it hard to bundle certain requests into one message. For example, if one request depends on the results of a previous request, then the two requests cannot be bundled, because the results needed by the next request are not known until the first request is finished. Still, our approach can be very useful for many frequent scenarios, such as portal site scenarios, as described in Section 2.1.

4. Performance evaluation

In this section, we evaluate the performance gain from our SOAP message bundling framework. We measured the results from three perspectives. First, in Section 4.1, we measure the message sizes for each bundled message. Second, Section 4.2 measures the XML Parsing overhead as a rough metric of XML performance. Finally, end-to-end response time for each bundled message exchange is evaluated in an end-to-end scenario in Section 4.3.

For these experiments, we set up a simulated service provider and simulated service requester. The simulated service provider was implemented from the original WSDL of the Google SOAP API as a simulated Google SOAP service. This service implements the interface in Listing 8. However the simulated service always returns the same results. These results are the same as the example messages in the developer's kit provided by Google. Therefore, the service application itself does almost nothing. Almost all of the overhead on the service provider's side is platform overhead, such as overhead for processing SOAP messages. Then we converted the WSDL and used the generated skeleton wrapper. On the service-requester side, we implemented a simple requester based on the original WSDL. The requester application generates the same parameter values as the example message in the developer's kit. Then our stub wrapper was used.

4.1. Message size

We captured the exchanged SOAP request and response messages for both the original and bundled operations. The request and response message sizes are shown in Figure 3 and Figure 4, respectively. In the graphs, "fine-grained" means the original messages and "bundled" means the bundled messages. Note that the size in the graph is the value per request. For example, the value for "bundled (3req/1bundle)" is the message size of one bundled message which includes 3 requests, divided by 3. Since "bundled (3req/1bundle)" has almost same size as the original fine-grained message, the actual message size of a bundled message including 3 requests is three times larger than the fine-grained message.

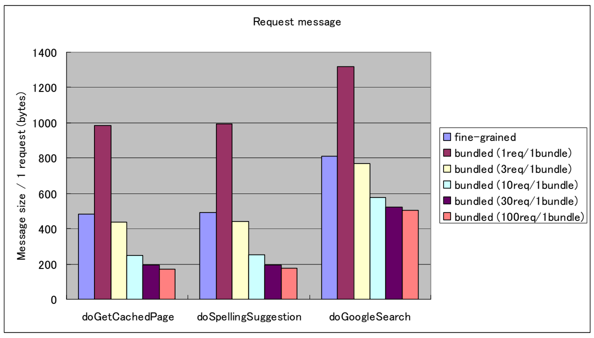

Figure 3. Message sizes of requests

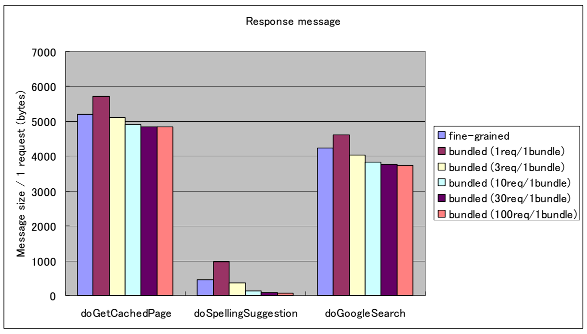

Figure 4. Message sizes of responses

In the graphs for request and response message sizes, especially for the request messages, the size of the "bundled (1req/1bundle)" message is larger than the fine-grained message. This is because the bundled message has some wrapper elements for the array of the set of input parameters.

In contrast, bundled messages that include more than three requests are effectively smaller than the fine-grained messages. This is because the bundled requests in one message share one SOAP envelope. In these experiments, the messages did not have SOAP header elements. If the bundled requests can also share the SOAP header, this approach would work even more effectively.

Response message sizes are not much different, especially for `doGetCachedPage` and `doGoogleSearch` operations. This is because the message sizes of these two operations are quite large. The data for the `doSpellingSuggestion` responses shows a similar pattern to the request message data, because these responses are also very small.

Overall, if the messages bundle more than three requests, there is no disadvantage due to the message size. If the message bundles only one request, the messages are larger than the original messages. However we can avoid this in our stub wrapper. The stub wrapper can check the number of requests before the bundling operation is invoked. If there is only one request, then the stub can invoke the original fine-grained operations.

4.2. Parsing overhead

We measured XML parsing overhead as an example of primitive XML processing. In this experiment, the Apache Xerces [15] XML parser was used as the SAX parser. Xerces is one of the most popular open source XML parsers in Java. The Apache-Axis SOAP engine uses the SAX parser internally for XML processing.

Figure 5 and Figure 6 show the parsing times of request and response messages, respectively. The values in the graph are elapsed times for 10,000 requests. The taller bars are slower. Here, "request" is the same as in Section 4.1. For example, the value of "bundled (3req/1bundle)" means the parsing time for the same 10,000 messages, which include 3 requests each, divided by 3.

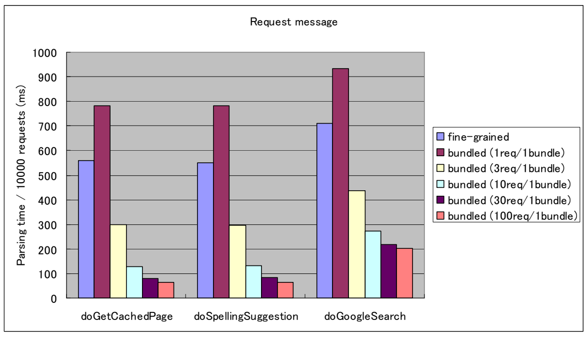

Figure 5. Parsing times for request messages

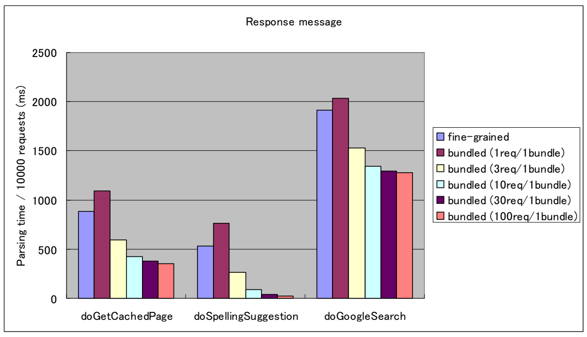

Figure 6. Parsing times for response messages

The graphs show almost the same pattern as the graphs for message size. This is because the parsing time is almost proportional to the message size. When "fine-grained" and "bundled (3req/1bundle)" are compared, the bundled messages are faster, even though the message sizes are almost the same. This result seems to show that many iterations of small XML parses are inefficient if the total size is the same. However, it is known that processing extremely large XML objects is very slow. In particular, the parsing for memory objects like DOMs is inefficient.

4.3. End-to-end response time

Finally, we evaluated the end-to-end response times. For the SOAP engine, we used Apache Axis. The service provider was the same simulated Google SOAP service. The service was developed on Axis and deployed with Tomcat on the service-provider machine. The service requester was also developed using Axis. The requester just repeats the same requests. The two machines, the provider and the requester, were on the same local network. The response times were measured as intervals from the requester's invocation to the time when the requester received the return value.

HTTP 1.1 Keep-alive is a technique to hold an open connection to a certain server. Multiple HTTP message exchanges can be processed on the one connection. HTTP Keep-alive looks similar to our technique, but they are in different layers, the HTTP transport layer versus the SOAP messaging layer. These techniques can coexist together. The experimental results were measured with HTTP Keep-Alive.

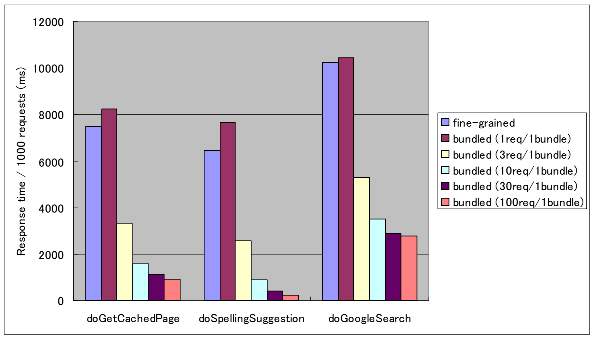

Figure 7. End-to-end response time

Figure 7 shows the total response times for 1,000 requests. The results show basically the same pattern as the results for message size and parsing time. However, this results show that bundled operations are much faster than fine-grained operations, compared with the results for message size and parsing time. This is because this response time includes network latency, which is incurred for each request-response message exchange. Because the fine-grained operation has many message exchanges, it is less efficient.

In these results, "bundled (3req/1bundle)" is much faster than "fine-grained". Actually, even the bundled messages with 2 requests are faster than "fine-grained". Since the bundled messages with only one request can be avoided as mentioned in Section 4.1, there are no cases where this bundling framework has any disadvantage.

5. Related work

Stored procedures for relational databases can bundle multiple SQL statements into one procedure. Although it seems similar to our approach, the point of the stored procedure is to bundle a sequence of statements. SQL statement can be regarded as data processing logic. Although we discuss this as future work in the next section, this paper does not focus on bundles of multiple related functions. Currently, our approach only aims to bundle some input parameters and return values into single messages. The data is not sequentially related and is without dependences.

Cook et al. [3] argue that a document-oriented style of communication is handled well in Web services. Overall, in document-oriented communications, larger messages are exchanged than when using RPC messages. Also, in the RPC style, the number of messages exchanged tends to be larger. In our approach, the number of exchanges is reduced even in the RPC style.

Takase et al. [6] proposed client-side caching for Web services. Caching is one of the effective techniques for performance improvement. When our bundling framework is used, the cache-hit ratio becomes lower if the bundled XML messages are themselves cached. However, they also proposed an operation-level cache. In the operation-level cache, the response for each operation is cached. In this case, the cache-hit ratio is not reduced, because the operations are cached before bundling.

6. Future work

In our approach, we cannot bundle messages that depend on each other. BPEL (Business Process Execution Language) may play a role to support such a feature in the Web services world. BPEL is a language to describe the workflow of Web services. By using BPEL, we can combine Web services. If the combined services can be processed on one physical machine at the same time, then the process can be very efficient.

As we described in Section 3.2, in some bundled operations, we would have to address the problems when individual operations fail or take too much time. According to the SOAP specification, a SOAP fault message should be returned when an operation fails. The SOAP fault message cannot include the normal response message. In such a case, an asynchronous SOAP message exchange could be considered as one of the solutions. In an asynchronous exchange, when only one operation has failed or timed out, the results of the other operations will be returned without the result of the failed operation. After that, a separate response for the failed operation can be returned. To implement this functionality, both the service provider and service requester have to support the asynchronous message exchange. Although asynchronous support is not yet popular, the asynchronous exchanges would be very efficient in these cases.

7. Concluding remarks

In this paper we introduced a SOAP message bundling framework. This framework enables existing service providers to add a bundled operation into their existing service implementations without any modifications. The bundled operation can receive messages that bundle multiple operations. Therefore, the service providers do not have to avoid designs with fine-grained services for performance reasons. Also, service requesters can easily use the bundled operation by using our framework. We implemented three tools for existing service providers and requesters, and evaluated the implementation to demonstrate the benefits of our framework. Finally, the results of our experiments showed improvements in the performance of SOAP message exchanges.

References

[1] C. Kohlhoff, and R. Steele. “Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems,” In *Proc. of the 12th International World Wide Web Conference*, WWW2003, Budapest, Hungary, 20-24 May 2003. ACM, 2003.

[2] D. Davis and M. Parasha. “Latency performance of soap implementations,” In *Proc. CCGrid’02, Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems*, Berlin, Germany, May 22-24, 2002, pages 407–412. IEEE Computer, 2002.

[3] W. R. Cook, and J. Barfield. “Web Services versus Distributed Objects: A Case Study of Performance and Interface Design,” *IEEE International Conference on Web Services* (ICWS 2006), pages 419-426, 18-22 September 2006, Chicago, Illinois, USA. IEEE Computer Society 2006

[4] K. Chiu, M. Govindaraju, and R. Bramley. "Investigating the limits of SOAP performance for scientific computing," *11th IEEE International Symposium on High Performance Distributed Computing* (HPDC-11 2002), 23-26 July 2002, Edinburgh, Scotland, UK, pages 246–254, IEEE Computer, 2002.

[5] S. Shirasuna, H. Nakada, and S. Sekiguchi. “Evaluating web services based implementations of GridRPC,” *11th IEEE International Symposium on High Performance Distributed Computing* (HPDC-11 2002), 23-26 July 2002, Edinburgh, Scotland, UK, pages 237-245, IEEE Computer, 2002.

[6] T. Takase, and M. Tatsubori. “Efficient Web Services Response Caching by Selecting Optimal Data Representation,” *24th International Conference on Distributed Computing Systems* (ICDCS 2004), 24-26 March 2004, Hachioji, Tokyo, Japan, pages 188-197, IEEE Computer Society 2004.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. "Hypertext transfer protocol – HTTP/1.1," *IETF RFC2616*, 1999.
http://www.ietf.org/rfc/rfc2616.txt

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. "Extensible Markup Language (XML) 1.0 (Second Edition)," W3C Recommendation, October 2000,
http://www.w3.org/TR/REC-xml

[9] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielson, S. Thatte, and D. Winer. "Simple Object Access Protocol (SOAP) 1.1," W3C Note, May 2000,
http://www.w3.org/TR/SOAP/

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. "Web Services Description Language (WSDL) 1.1," W3C Note, March 2001,
http://www.w3.org/TR/wsdl

[11] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, and P. Yendluri. WS-I, "Basic Profile Version 1.0," Final Material, April, 2004,
http://www.ws-i.org/Profiles/BasicProfile-1.0.html

[12] Google SOAP Search API (Beta),
http://code.google.com/apis/soapsearch/

[13] R. Chinnici, M. Hadley, R. Mordani. The Java API for XML-Based Web Services (JAX-WS) 2.0, Final Release, April 19, 2006,
http://jcp.org/en/jsr/detail?id=224

[14] The Apache Software Foundation, Apache <Web Services/> project, Apache-Axis,
http://ws.apache.org/axis/

[15] The Apache Software Foundation, Apache XML project, Apache-Xerces,
http://xerces.apache.org/

[16] H. Adams, "Web services performance considerations, Part 1," IBM developerWorks, Feb 2004.
http://www.ibm.com/developerworks/library/ws-best9/

[17] O. Zimmermann, S. Milinski, M. Craes, and F. Oellermann. "Second Generation Web Services-Oriented Architecture in Production in the Finance Industry," *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, (OOPSLA 2004), pages 283-289, October 24–28, 2004, Vancouver, British Columbia, Canada. ACM 2004.

[18] <soaprpc/>: Performance best practices for Web services, May 2004.
http://www.soaprpc.com/archives/000020.htm

[19] F. Lelli, G. Maron, and S. Orlando. "Improving the performance of XML based technologies by caching and reusing information," *IEEE International Conference on Web Services* (ICWS 2006), pages 689-700, 18-22 September 2006, Chicago, Illinois, USA. IEEE Computer Society 2006

[20] I. Matsumura, T. Ishida, Y. Murakami, and Y. Fujishiro. "Situated Web Service: Context-Aware Approach to High-Speed Web Service Communication," *IEEE International Conference on Web Services* (ICWS 2006), pages 673-680, 18-22 September 2006, Chicago, Illinois, USA. IEEE Computer Society 2006