Flex-SwA: Flexible Exchange of Binary Data Based on SOAP Messages with Attachments *

Steffen Heinzl, Markus Mathes, Thomas Friese, Matthew Smith, Bernd Freisleben
Dept. of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str., D-35032 Marburg, Germany
{heinzl, mathes, friese, matthew, freisleb}@informatik.uni-marburg.de

Abstract

SOAP is the standard protocol for message exchange in web service environments. As an XML-based protocol, SOAP is not suitable for the transmission of large amounts of binary data. This fact has been addressed by the SOAP Messages with Attachments specification, which regulates the transfer of a SOAP message together with an arbitrary number of binary attachments composed within a MIME multipart/related message. Although this leads to a reduction of transmission overhead, web service communication using SOAP Messages with Attachments still lacks communication and processing flexibility. In this paper, we present a novel and more flexible way of handling attachments in SOAP-based web service environments. In contrast to SOAP Messages with Attachments, our approach offers message forwarding without additional communication cost and demand-driven evaluation and transmission of binary data, thus providing the opportunity to save time by overlapping service execution and data transmission.

1 Introduction

The web service paradigm has been adopted in diverse application areas such as enterprise application integration, business-to-business communication, multimedia content distribution or service-oriented Grid computing [7]. The main reasons are interoperability, the self-describing nature of XML-based interfaces and message formats, and the smooth alignment with successful Internet standards such as the HTTP protocol. Many of these applications require the transmission of large amounts of binary data, such as audio/video data in multimedia content distribution or experimental measurement data in Grid computing.

*This work is partially supported by the German Research Foundation, DFG (SFB/FK 615, Project MT), the German Ministry of Education and Research, BMBF (D-Grid Initiative, In-Grid Project), and Siemens AG, Corporate Technology, München.

Embedding such data in the SOAP messages used for service invocation is not a reasonable approach, because the XML formats that SOAP is built on are not suitable to hold large binary objects. This fact has been addressed by the introduction of the *SOAP Messages with Attachments (SwA)* [4] specification. It defines how a SOAP message and several binary objects can be composed into a Multipurpose Internet Mail Extensions (MIME) message of type multipart/related. Each message part is separated by a unique delimiter string defined at the beginning of the message [9].

Unfortunately, MIME does not allow random access to arbitrary message elements without prior reception of the entire message. A web service cannot decide which binary objects to receive and it cannot decide the order in which binary objects are transferred, since each SwA message is transferred as a whole using the underlying transfer protocol (in most cases HTTP).

In this paper, we present *Flex-SwA*, a new approach to handle SwA messages in a more flexible way. Flex-SwA has been designed to preserve interoperability with the existing SwA specification while it allows adopters to benefit from more flexible attachment handling mechanisms. It enables the choice of the actual transport protocols used for individual message parts, design of applications with reduced latency and easy redirection of service invocations without prohibitive increase in network load. Both remote and local files can be referenced, often saving the effort for one transmission (e.g. the download of the file prior to sending it to the service as an attachment of the SOAP message). An implementation is presented, demonstrating that using our approach can be nearly transparent for both client and server.

The paper is organized as follows. Section 2 presents the simple web services interaction model based on SwA and Apache Axis. The overall architecture of our approach is presented in section 3. Implementation issues and results of some performance experiments are described in section 4. Section 5 discusses related work. Section 6 concludes the paper and outlines topics for future work.

2 Web Services Interaction Model

Web service interaction takes place by exchanging SOAP messages. Fundamentally, a SOAP message is a stateless one-way message. By combining one-way message exchanges, more complex interaction patterns can be created, e.g. request/response [13]. These patterns are often based on the client/server-paradigm. Clients perform web service (WS) invocation by passing SOAP messages to the server. These messages contain the target service to invoke, as well as any number of parameters to be passed to the service. Reply messages may be transmitted either synchronously or asynchronously from the server to the client. The SOAP protocol defines an XML-based format for the messages to be used in a WS invocation. This XML format cannot handle large binary content well, since it requires an encoding of the content prior to embedding it into a message.

For this reason, the SwA specification has been proposed to transmit large binary data objects outside of the XML part. To attach binary objects to a SOAP message, SwA uses MIME multipart/related messages. The SOAP message always resides in the first part of the multipart message. MIME uses a delimiter to separate the different message parts from each other, such that the message must be parsed until the delimiter of the desired part of the message is found.

2.1 Apache Axis

Apache Axis is an implementation of SOAP [16]. Its main purpose is to handle the transfer and processing of SOAP messages. Axis provides a server and a client both consisting of a set of handler chains. Each handler is capable of changing an incoming or outgoing message and passing it to the next handler in the chain. This enables pre- and postprocessing of incoming or outgoing messages. Handlers pass messages in a so-called *message context* that can hold additional information. A chain is a composition of handlers and other chains. Three chains are predefined in the Axis client and server: the transport chain, the global chain and a chain where the service resides.

When a message arrives at the server, it is passed through these three chains which handle incoming messages. First, it is put in a message context which is forwarded to the transport chain. This chain handles transport specific issues, like the protocol being used to send the SOAP message (HTTP by default). Then, the message context is forwarded to the global chain implementing, for example, security policies. If the processing in the global chain took place without errors, the message context is passed to the service specific chain. Handlers in this chain may manipulate the message before it is passed to the actual service. A reply message from the service is passed along a response handler chain to the client.

Axis provides a `Call` object for service invocation, which handles creation of a message and invocation of a service. After starting the invocation, the client-side message processing takes place. The `Call` object contains the message context which again is passed through the chains.

2.2 WS Interaction Patterns

The general WS interaction pattern using SwA is shown in figure 1. A main disadvantage of this approach is that the server has to wait for the transmission of the entire message before it can decode every attachment part. As a consequence, the transmission of large attachments can take a considerable amount of time and attachment processing requires large amounts of main or external memory. Depending on the implementation of the underlying WS framework, processing of the SOAP message and invocation of the target service is often deferred until the entire message containing all attachments has been completely received.

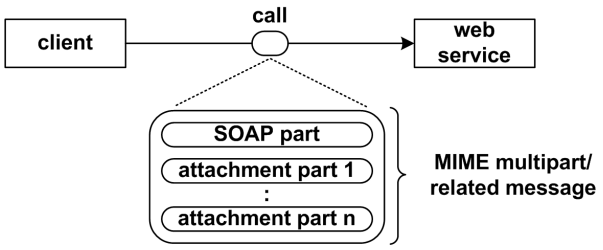

Figure 1. General WS interaction pattern.

In application areas where large amounts of data often have to be transferred, e.g. multimedia computing [5, 6] or Grid computing, these disadvantages are not acceptable. Therefore, other implementation patterns handling the transmission of large amounts of data have been implemented in an application specific manner. A common implementation pattern to be found in such an environment is the transmission of data location pointers (URIs or more complex reference structures) that are interpreted in the application logic of the service and that are also application specific. The service implementation then uses other data access components, e.g. OGSA-DAI [18] or RFT [17] in Grid environments.

3 Flex-SwA

The primary objective of the approach presented in this paper is to provide a more flexible mechanism for handling large data attachments in SOAP-based interactions. Instead of leaving the concrete transmission of large data objects to

the SOAP engine or the application developer, our approach offers functionality to handle such attachments.

In practice, the invocation of a WS is based on the exchange of SOAP messages. SOAP itself offers two binding styles: *rpc* and *document*. The first one indicates that an operation is RPC-oriented, i.e. a request message contains parameters, a response message return values. The latter indicates that an operation is document-oriented, i.e. a message contains documents (e.g. facsimiles). Flex-SwA supports both styles.

Figure 2 shows the entire protocol stack from the service user's and the service provider's point of view. Each layer of the protocol stack uses lower layers and offers special functionality to higher layers. The client resides on top of the service user's protocol stack, which is using the Flex-SwA layer to invoke remote services. The Flex-SwA layer uses a markup generator to create an XML description, which refers to the actual location of the file and the protocols used to transmit the binary data objects. The service provider can retrieve the data directly from the remote server instead of transferring data from a remote server to the client and from the client to the service provider.

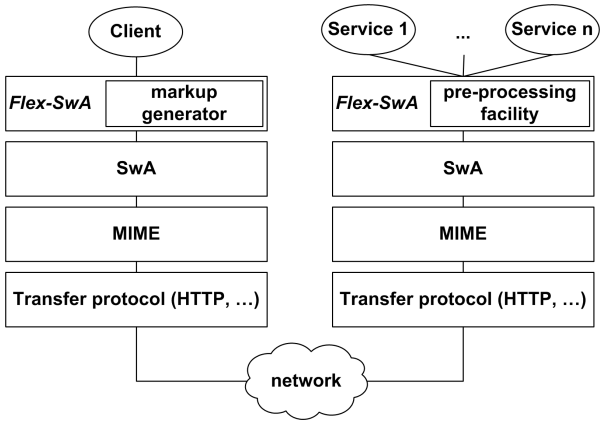

Figure 2. The Flex-SwA protocol stack.

A message containing references is created by adding the markup in one of the parts of the multipart MIME message, as shown in figure 3.

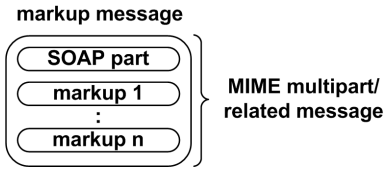

Figure 3. Structure of a markup message.

On top of the service provider's stack, several services are offered. All these services are based on the Flex-SwA layer. It encapsulates messaging specific functionality. At the service provider, Flex-SwA uses a preprocessing facility to parse markup received by a service caller and subsequently prepares data transmission. The preprocessing facility does not need to handle all of the markup. The unhandled markup can be forwarded to other service providers, thus providing *message forwarding without additional communication cost*, since the markup – compared to the data tranferred – is very small. Both Flex-SwA layers at the service user's and service provider's site are using SwA to send and receive service invocations, respectively. SwA messages are encapsulated using MIME. Different transfer protocols can be used, e.g. HTTP.

From an application developer's point of view, WS invocation and data transmission remain coupled in a single service invocation operation. The concrete behavior of the platform regarding the handling of data transmission and service execution can be controlled by specifying a *behavioral policy*. As an additional benefit, service developers can use the protocol handling capabilities of Flex-SwA to leverage high performance binary protocols by simply specifying a policy to use them, without having to deal with the protocol details in the application code. Binary protocols can be selected for each message part individually. In contrast to a realization in a more traditional application environment where the developer has to handle every aspect of the communication, most of the functionality needed to handle a specific transport mechanism are realized in the Flex-SwA layer.

Such a policy can be specified as a default behavior for the entire platform (e.g. regarding the selection of a preferred transport protocol) or as a service policy. Two possible behaviors exist regarding the handling of data transmission and execution of a service. In *non-overlapping* mode, the platform performs all data transfer prior to invocation of the service; in *overlapping* mode, data transmission and service execution are performed in parallel. If a service needs to make sure that all data is available on the service provider's platform before it starts processing, it requests the platform to handle invocations in non-overlapping mode. If initialization of the service requires time and is independent of the attachment data, a service developer can specify the service to use overlapping invocation mode, causing the platform to start data transmission and service execution in parallel and thus reducing latency to service execution.

The service can instruct the platform to perform *eager* or *lazy* transmission of attachments, meaning that attachments are collected and stored as soon as possible or only upon a real attempt to access the attachment content. Additionally, the service can prioritize attachments that are tagged to be transmitted in eager mode, leading to a transmission plan for the attachments. The resulting execution and transmis-

sion patterns are shown in figure 4.

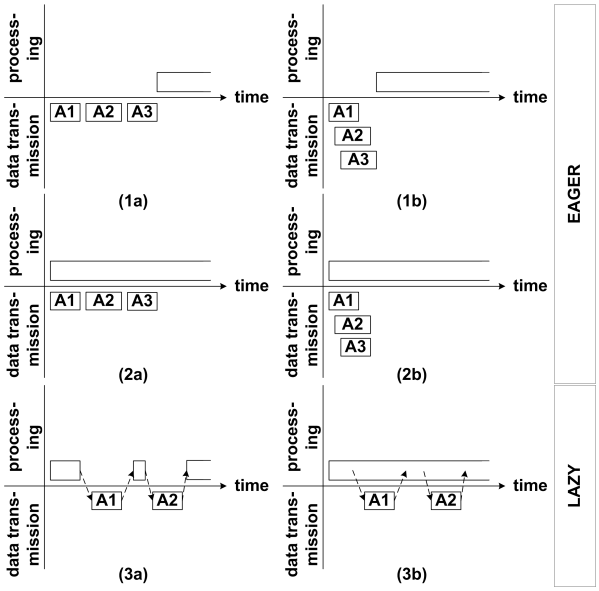

Figure 4. Data transmission and service execution patterns.

A combination of non-overlapping transmission handling and the eager transmission mode (1a) results in transferring every attachment before the service starts. This scheme is similar to the original transmission via SwA. Transmission of the attachments can also be done concurrently (1b), for example by using several threads, thus providing the possibility of improving the transfer rate.

Combining overlapping and eager transmission handling (2a) results in the immediate start of data transmission and the service. This mode is useful if the service has a certain warm-up time or does not need any attachments at service start. Here again, a concurrent transmission of attachments (2b) possibly provides a better transfer rate than the iterative approach.

Lazy data transmission in combination with overlapping transmission handling results in an on-demand transmission of attachments. If the service needs an attachment, transmission is triggered at that time. Again, there are two modes: *Blocking* (3a) – The service is blocked until data is retrieved from the remote source and stored locally by the infrastructure. *Non-blocking* (3b) – The service only triggers the transmission and continues directly, transferred data may be accessed by the service upon reception. Blocking mode is used if the service needs the complete attachment before the service can resume execution. Non-blocking mode can be used if only a part of the attachment is needed by the service, e.g. an IDEA algorithm [11] needs the next 8-byte blocks for encryption. The service execution patterns shown in figure 4 enable the *demand-driven evaluation and transmission* of binary data.

3.1 Compatibility

A crucial requirement for the Flex-SwA approach is interoperability with existing servers using SwA. There are four possible combinations of SwA and Flex-SwA. A server based on Flex-SwA has to handle incoming requests from a SwA client just like a standard server. A client needs a way to find out whether the service supports Flex-SwA or not. In the latter case, it has to transmit data as a standard attachment without resorting to other transport protocols. In order to inform the client about the service capabilities, the service description is used to signal the client if a server supports Flex-SwA. While it can be interpreted by Flex-SwA aware clients, standard SwA clients will ignore the extensibility elements. Every Flex-SwA server supports standard MIME multipart/related messages as the protocol for attachment transmission, therefore every SwA client can communicate with a Flex-SwA server.

In addition to the information that a service supports Flex-SwA, the client also needs to know which concrete protocols the service supports for the transmission of an attachment. In general, this property is platform dependent and varies over time, i.e. some protocols might only be accessible at a certain time. If the service interface description is generated on request prior to a service call, these protocol capabilities may also be embedded in the WSDL document. The interface definition may, however, be prefetched and cached in the client, depending on the application needs. Therefore, the Flex-SwA platform can be configured to embed protocol information in dynamically generated WSDL descriptions as well as to provide them through a special WS. Flex-SwA clients can query to determine a suitable protocol on demand.

3.2 Protocol Decision

Some transport protocols used for access to attachment data may require *preparation overhead*. For example, consider the transmission of binary data objects via a GridFTP [3] server that is shared by both service provider and service user. In this scenario, the client has to upload the data to the GridFTP server before the service provider can access the data and retrieve it from the GridFTP server. Other protocols that enable access to the attachment data from the client node do not require such a preparation overhead. Service developers can specify a priority order for acceptable protocols for attachment transmission.

The client calculates the intersection of the sets of protocols that both client and service support. It may then decide to make the attachment data available by any number

of protocols in the order of protocol preference expressed by the service. At least one protocol must be supported; the client is free to include other protocols if they, for example, require no or only limited preparation overhead.

4 Implementation

This section presents an implementation of the Flex-SwA architecture based on Apache Axis 1.2.1 and Tomcat 5.5.9. Since Flex-SwA is still based on the client/server paradigm, the interaction between a client and a service, as shown in figure 5, will be explained.

Client. The client instructs the `AttachmentFactory` to create a new attachment. The `AttachmentFactory` is a class using the static method `createAttachmentPart()` to create an `AttachmentPart` from one or more URIs, at least one for each resource referenced. For each resource, an XML markup is created, including information for the service provider regarding the protocol to use, such as e.g. GridFTP, BitTorrent, Resource Management Framework [10], TCP, etc., and where the resource can be downloaded. To support the implementation of further protocols, a link to an attachment factory implementation can be provided by a configuration file. A markup generated by the `AttachmentFactory` is transmitted by using SwA over MIME multipart/related messages. An example message with two attachments describing resources to be received via TCP or GridFTP and TCP or BitTorrent transmission is shown below.

```
------=_Part_0_9938272.1130318994709
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: binary
Content-Id: <319782F5FEDC2E63C058F1E31AA21F79>


<?xml version="1.0" encoding="UTF-8"?>
<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>
    <putFile soapenv:encodingStyle="
      http://schemas.xmlsoap.org/soap/encoding/"/>
  </soapenv:Body>
</soapenv:Envelope>

------=_Part_0_9938272.1130318994709
Content-Type: text/xml
Content-Transfer-Encoding: binary
Content-Id: <Callback>


<?xml version="1.0" encoding="UTF-8"?>
<resource ID="1">
  <protocol type="GRID_FTP">
    <source>
      gsiftp://remote.machine1.net/out/media1.mpeg
    </source>
    <tcp-buffer-size>2097152</tcp-buffer-size>
    <parallel>4</parallel>
  </protocol>
  <protocol type="SOCK_STREAM">
    <host>137.248.132.25</host>
    <port>8091</port>
    <filename>media1.mpeg</filename>
    <filelength>2256896643</filelength>
  </protocol>
</resource>

------=_Part_0_9938272.1130318994709
Content-Type: text/xml
Content-Transfer-Encoding: binary
Content-Id: <Callback>


<?xml version="1.0" encoding="UTF-8"?>
<resource ID="2">
  <protocol type="SOCK_STREAM">
    <host>137.248.132.25</host>
    <port>8091</port>
    <filename>media2.mpeg</filename>
    <filelength>1243267576</filelength>
  </protocol>
  <protocol type="BITTORRENT">
    <torrentURL>
      http://torrent.host.org/test.torrent
    </torrentURL>
  </protocol>
</resource>
------=_Part_0_9938272.1130318994709--
```

After the SOAP envelope, the attachments with XML markup begin. Each markup contains a `resource` element as its root. In the example above, two resources with resource ID 1 and 2 are defined. Resource 1 offers two protocols for download: GridFTP and a simple TCP socket connection. To call the GridFTP globus-url-copy command-line tool, several options are provided. The second resource also offers two download protocols: a simple TCP connection to a file server and the URL of a BitTorrent .torrent file. By defining several locations for a resource, the implementation can flexibly select one protocol depending on different system metrics, such as e.g. network load.

Server. The server is configured by a policy file which associates a behavioral policy with each service. These associations are read by a `PolicyManager` at server start. At message arrival, the `PreprocessingHandler` for this service consults the `PolicyManager` and retrieves the corresponding policy. Then, for each markup a transport protocol is chosen. At this point, four different policies are feasible:
(1) If the service policy is eager and non-overlapping, the transmission of the attachment with the highest priority begins. Then, the attachment with the next lower priority is transmitted until every attachment has arrived. After the transmission is completed, the service is started.
(2) If the service realizes an eager and overlapping policy, a

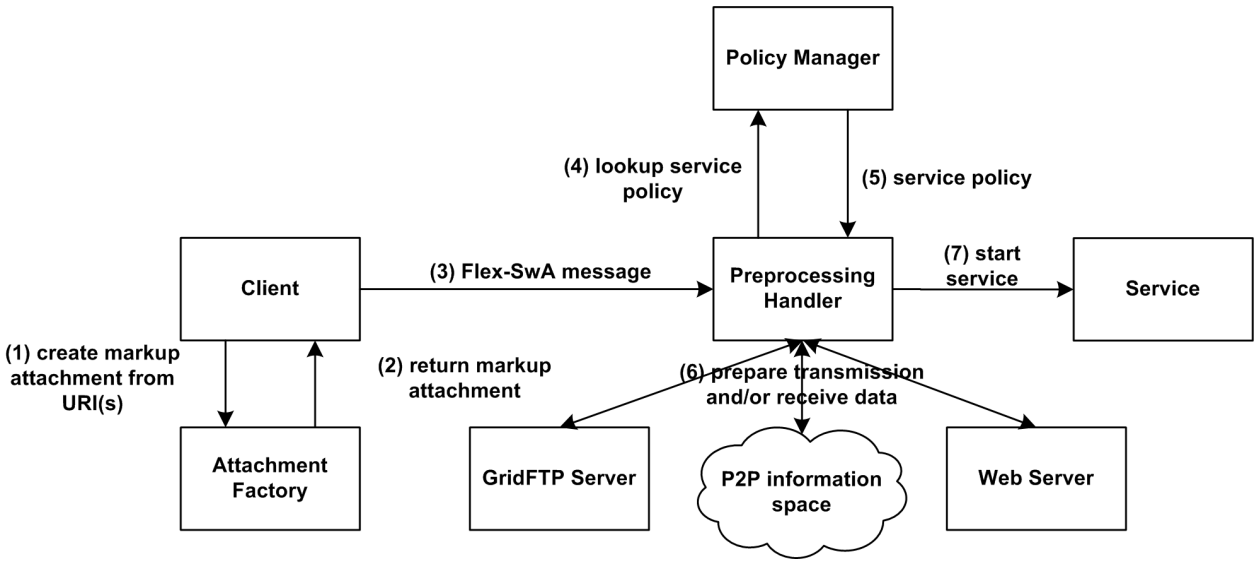

Figure 5. An implementation of the Flex-SwA architecture.

thread is started to initiate the transmission of the attachments in priority order and then the service immediately starts.
(3) If a lazy, overlapping policy in blocking mode is used, the service is started immediately. If the service needs an attachment, it blocks until the attachment is completely transferred and then proceeds execution.
(4) If a lazy, overlapping policy in non-blocking mode is used, a thread is started which prepares the transmission of the attachments. Thereafter, the service begins execution. If the service needs an attachment, the thread is triggered to start the transmission concurrently.

Interaction. In order to transfer bulk data from client to target service, several steps are necessary, which are shown in figure 5. The client calls the `AttachmentFactory` to create a markup attachment for a given URI (1). This step may occur several times if the client wants to send more than one resource. The newly created markup attachment is returned by the `AttachmentFactory` (2). Now, the client is able to compose a multipart message of the attachments and send it to the service (3). After arrival of the message at the service provider, the `PreprocessingHandler` needs a behavioral policy to process the message correctly. Therefore, it calls the `PolicyManager` to lookup the behavioral policy for the target service (4). The `PolicyManager` uses its internal mapping table to retrieve a behavioral policy for the target service and returns it to the `PreprocessingHandler` (5). The `PreprocessingHandler` is able to prepare transmission depending on the received markup attachments and the policy returned by the `PolicyManager` (6). Finally, the target service is started by the `PreprocessingHandler` to process the received data (7).

4.1 Policies

The following example shows a behavioral policy file (policy.xml) configuring two services (`LazyService` and `EagerService`) on the service provider's site:

```
<?xml version="1.0"?>
<policy xmlns="http://www.fb12.de/flex-swa"
  xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.de.fb12.flex-swa policy.xsd">

  <default>
    <protocol name="SOCK_STREAM" priority="30"
              impl="SocketImpl"/>
    <protocol name="GridFTP" priority="20"
              impl="GridFTPImpl"/>
  </default>

  <service name="LazyService">
    <transmission type="lazy" blocking="false"/>
    <processing type="overlapping"/>
    <attachment id="1" priority="15">
      <protocol name="GridFTP" priority="1"
                impl="GridFTPImpl"/>
      <protocol name="RFT" priority="2"
                impl="RFTImpl"/>
      <protocol name="SOCK_STREAM" priority="3"
                impl="SocketImpl"/>
    </attachment>
    <attachment id="2"/>
    <attachment id="3" priority="7"/>
  </service>

  <service name="EagerService">
    <transmission type="eager"/>
```

```
    <processing type="non-overlapping"/>
    <attachment id="1"/>
    <attachment id="2"/>
  </service>
</policy>
```

By default, every service supports the transport protocols TCP and GridFTP. TCP is used as the preferred transport protocol if it is supported by the resource server, since it has the highest priority value assigned. The attribute `impl` describes the class file to use for the implementation of the protocol (package names are omitted for brevity in the example). While these default parameters have a platform scope, the following special configurations are expressed for the two services:

`LazyService` uses the transmission mode `lazy` in a non-blocking manner. The handling of data transmission and service start is `overlapping`. The service expects three attachments. The first one with the highest priority supports three protocols. The second one uses the standard protocols as defined in the default section and a standard priority of 10 as defined in the related schema (policy.xsd). The third attachment also uses the standard protocols and has the lowest priority.

The `EagerService` uses the transmission mode `eager` and is `non-overlapping`. It expects two attachments to be transferred with the standard protocols. Since eager and non-overlapping means that all attachments have to arrive before service start, a priority does not have to be specified.

4.2 Experimental Results

Flex-SwA offers flexibility in the transmission of binary data, but increased flexibility often comes along with reduced performance. To show that the performance is comparable to the performance of SwA, we have conducted an experiment with the example implementation described above. The experiment measures the overhead introduced by using the `AttachmentFactory` at client-side and the `PreprocessingHandler` at server-side with respect to the time needed throughout a SOAP call containing attachments. An Intel Pentium M notebook with 1.5GHz processor speed and 1024MB of RAM running Windows XP Professional SP2 acted as the client. As the server, an Intel Pentium M notebook with 2GHz and 512MB of RAM running Windows XP Professional SP2 was used. The server runs Apache Tomcat 5.5.9 in combination with Apache Axis 1.2.1 as SOAP engine. To connect client and server, a standard 100 Mbps fast-ethernet LAN was used. Since operating speed may fluctuate, the experiment was repeated 1000 times to compute an average value. The experimental results show that usage of the `AttachmentFactory` takes *12.78ms* on the average, whereas the usage of the `PreprocessingHandler` takes *6.16ms*. In the worst case, the `AttachmentFactory` introduced *113.67ms* of overhead, whereas the `PreprocessingHandler` needed an additional overhead of *180.69ms*. This additional overhead is negligible in comparison with the time required for the entire client/server-interaction and data transmission.

Furthermore, using Flex-SwA may lead to a performance gain under certain conditions. Consider a scenario where the service needs a remarkable amount of setup time t_s, receiving of binary data takes t_r units of time, the execution of the service takes t_e units of time and setting up the service requires no data from the client. Using SwA means that receiving of binary data, setting up the service, processing and execution takes place in a sequential manner. Hence, $t_s + t_r + t_e$ units of time are needed. On the other hand, using Flex-SwA enables overlapping of receiving data and setting up the service. Hence, only $t_r + t_e$ units of time are required for large amounts of data.

5 Related Work

Seshasayee et al. introduce SOAP-bin and SOAP-binQ as a combination of SOAP with an efficient binary protocol [15]. Although they are alternatives to SwA, they have the same disadvantages. The order of attachment transmission cannot be chosen and overlapping of service execution and data transmission is not possible. Hence, the proposals are not as flexible as our approach.

Nielsen et al. present an alternative to MIME named Direct Internet Message Encapsulation (DIME) [14]. DIME enables random access to each message part of a multipart message. Flex-SwA can easily be built upon DIME, since the underlying message representation protocol is transparent to Flex-SwA.

The work presented by Ying et al. shows that pure SOAP produces too much overhead to be considered efficient [20]. SwA, on the other hand, is significantly faster if the amount of data increases [19]. Our approach increases the flexibility of the fast approach without creating too much overhead.

One other reason for the bad performance of SOAP are the serialization- and deserialization processes, respectively. Abu-Ghazaleh et al. present an approach named differential serialization, which reuses a serialized SOAP message as a template for further messages [2]. Furthermore, an approach to improve deserialization called differential deserialization is discussed [1]. These approaches become interesting if the size of the required XML markup grows. Since our approach only uses small SOAP messages, the performance gain of differential serialization/deserialization is negligible for Flex-SwA.

NaradaBrokering is an open-source messaging infrastructure based on a network of message brokers. In NaradaBrokering events are used to encapsulate different information, i.e. NaradaBrokering is an event brokering sys-

tem. Fox et al. present an approach to interface NaradaBrokering with WS [8]. Since NaradaBrokering is based on brokers collaborating in a broker network, it differs significantly from our architecture.

To boost efficiency Lu et al. present an implementation of a generic SOAP engine which supports textual and binary XML as encoding scheme of messages [12]. Since our approach is based on textual XML no additional support of the binary encoding schema is necessary.

6 Conclusions

There are several disadvantages associated with SwA proposed for transferring large amounts of data in WS environments: it is not possible (a) to overlap service execution and data transmission, (b) to let the service provider decide when and in which order attachments should be transferred, and (c) to forward messages to other service providers without additional costs. These issues are obstacles to the ongoing adoption of the WS standards in other application fields, e.g. service-oriented Grid computing.

In this paper, the Flex-SwA approach for increasing the flexibility of bulk data transmissions in service-oriented environments has been presented. Flex-SwA allows the use of different protocols for data transmission including e.g. TCP, GridFTP, RFT and BitTorrent. A service provider is offered the opportunity to select which attachments to transfer in which order during preprocessing. Attachments can even be omitted, for example, if an error occurs after having transferred the first part of an attachment. Furthermore, a server-side preprocessing facility may only process some of the markups and forward the rest to another service provider. While transferring attachments, it is possible to start service execution.

Flex-SwA is compatible to SwA. Our implementation described in this paper shows that using Flex-SwA can be nearly transparent for both client and server. Furthermore, Flex-SwA creates only a negligible overhead compared to SwA.

There are several areas for future work, including (a) performing a more detailed performance evaluation regarding the different execution and transmission patterns (b) developing applications that leverage Flex-SwA's ability to redirect SOAP invocations without prohibitive transmission overhead and (c) investigating autonomic strategies for selecting optimal protocols.

References

[1] N. Abu-Ghazaleh and M. J. Lewis. Differential Deserialization for Optimized SOAP Performance. In *Proc. of the Int'l. Conference for High Performance Computing, Networking, and Storage*, page 21, 2005.

[2] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential Serialization for Optimized SOAP Performance. In *Proc. of the 13th IEEE Int'l. Symposium on High Performance Distributed Computing*, pages 55–64, 2004.

[3] W. Allcock, J. Bester, J. Bresnahan, S. Meder, P. Plaszczak, and S. Tuecke. GridFTP: Protocol Extensions to FTP for the Grid, April 2003. `http://www.gridforum.org/`.

[4] J. J. Barton, S. Thatte, and H. F. Nielsen. SOAP Messages with Attachments. W3C Note, 2000. `http://www.w3.org/TR/SOAP-attachments`.

[5] R. Ewerth and B. Freisleben. Video Cut Detection without Thresholds. In *Proc. of the 11th Workshop on Signals, Systems and Image Processing*, pages 227–230, 2004.

[6] R. Ewerth, M. Schwalb, P. Tessmann, and B. Freisleben. Estimation of Arbitrary Camera Motion in MPEG Videos. In *Proc. of the 17th Int'l Conf. on Pattern Recognition*, pages 512–515, 2004.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *The International Journal of High Performance Computing Applications*, 15:200–222, 2001.

[8] G. Fox, S. Pallickara, and S. Parastatidis. Towards Flexible Messaging for SOAP Based Services. In *Proc. of the IEEE/ACM Supercomputing Conf.*, page 8, 2004.

[9] N. Freed, N. Borenstein, K. Moore, J. Klensin, and J. Postel. RFC 2045-2049: Multipurpose Internet Mail Extensions (MIME), 1996. `http://www.ietf.org/rfc.html`.

[10] T. Friese, B. Freisleben, S. Rusitschka, and A. Southhall. A Framework for Resource Management in Peer-to-Peer Networks. In *Proc. of the Int'l. Conf. NetObjectDays 2002*, pages 4–21, 2002.

[11] X. Lai and J. L. Massey. A Proposal for a New Block Encryption Standard. *Lecture Notes in Computer Science*, 473:389, 1991.

[12] W. Lu, K. Chiu, and D. Gannon. Building a Generic SOAP Framework over Binary XML. In *Proc. of the 15th IEEE Int'l. Symposium on High Performance Distributed Computing*, 2006. To Appear.

[13] N. Mitra. SOAP Version 1.2 Part 0: Primer, 2003. `http://www.w3.org/`.

[14] H. F. Nielsen, H. Sanders, R. Butek, and S. Nash. Direct Internet Message Encapsulation (DIME), 2002.

[15] B. Seshasayee, K. Schwan, and P. Widener. SOAP-binQ High-Performance SOAP with Continuous Quality Management. In *Proc. of the 24th Int'l. Conf. on Distributed Computing Systems*, pages 158–165, 2004.

[16] The Apache Software Foundation. Apache Axis. `http://ws.apache.org/axis/`.

[17] The Globus Alliance. GT Data Management: Reliable File Transfer (RFT), 2005. `http://www.globus.org/toolkit/data/rft/`.

[18] The University of Edinburgh. The OGSA-DAI Project. `http://www.ogsadai.org.uk/`.

[19] R. van Engelen, G. Gupta, and S. Pant. Developing Web Services for C and C++. *IEEE Internet Computing*, 7:53–61, 2003.

[20] Y. Ying, Y. Huang, and D. W. Walker. A Performance Evaluation of Using SOAP with Attachments for e-Science. In *Proc. of the UK e-Science All Hands Meeting*, pages 796–803, 2005.