Enabling web services to consume and produce large distributed datasets

Spiros Koulouzis, Reginald Cushing, Konstantinos Karasavvas, Adam Belloum, Marian Bubak

Abstract—Service Oriented Architectures (SOAs) and Web Services (WSs) are well established paradigms for developing distributed applications. WSs may offer a solution that can take advantage of resources offered by Grids and Clouds. However, WSs face problems when accessing, moving and processing large data. This is especially true for legacy scientific WSs which need to process large datasets in complex data-intensive applications. To address this problem we present a WS called ProxyWS that uses a multitude of protocols to transport large data. The ProxyWS undertakes data transfers, on behalf of legacy WSs. Moreover, the ProxyWS can be used as an interface for developing WSs able to stream data. To illustrate the benefits of the ProxyWS we present two data-intensive applications. The experimental results show how this approach facilitates scalable data transports for such applications.

Index Terms—Data intensive applications, SOA, Web Services, Data Transfers, Distributed Indexing.

1 INTRODUCTION

Service Oriented Architecture (SOA) refers to a style of building distributed systems that deliver functionality as a service which can be discovered and used by clients through standardized descriptions [1]. Web Services, being an implementation of SOA, may be described as distributed stateless computation elements. SOAP-based services are a popular implementation of web services. These services use a standardized XML-based invocational protocol called SOAP [2] and description language called WSDL [3]. This enables web services to expose all or part of any application to consuming clients. These features are used to combine services in a loosely coupled way so that more complex operations may be constructed. The adoption of the web service paradigm allows developers to encapsulate the internal implementation within the service and provide a descriptive interface of its methods and data types to its consumers. This approach enables service owners to adapt the internal implementation without changing the way service clients access it. This is a desirable feature that provides the bases for building large scale, complex distributed applications [4].

However, web services face some problems regarding data transfers. The SOAP protocol, although appropriate for remote method invocation, is not suitable for large data transfers as it does not offer a scalable and fast way of transporting large datasets which is particularly necessary in legacy services that tend to use SOAP for transporting large data. Trying to fit data larger than some megabytes in a SOAP message, can cause the client or the service to run out of memory since SOAP can bloat the message size by a factor of ten. Also, parsing large XML documents is also expensive in terms of processing time which exacerbates the load on the service [5].

The inefficiency of SOAP for data transfers results in an abundance of highly descriptive interoperable application components that are unable to consume and produce substantial amounts of data. This also affects e-Science applications such as in bio-informatics, where web services are widely adopted for composing complex workflow based experiments. Using enactors such as Scientific Workflow Management Systems (SWMS) to orchestrate web services results in passing SOAP messages through this centralized service enactor [6], [7]. Although this approach provides a better control over the execution, it is inefficient for transferring data [8]. A more effective way of orchestrating data-centric workflows would be to deliver data directly to consuming services, preferably by creating connections between services with data streams [9].

In this paper, we describe an efficient solution for transporting large data sets between web services, implemented as a data-aware web service called ProxyWS. The ProxyWS coordinates the transportation of large data volumes from remote resources (such as Secure FTP, GridFTP, etc.), as we have presented in our previous work [10]. The ProxyWS also provides support for data transfers and data streams between web services. Additionally with the ProxyWS legacy web services can stretch their current potential by referencing data that would otherwise be delivered via SOAP. Finally, we demonstrate that web services can process and deliver larger datasets with streaming between web services.

The remainder of this paper is organized as follows. In Section 2 we analyze related work on data transports

- S. Koulouzis, R. Cushing, A. Belloum, are with the Institute for Informatics, University of Amsterdam. E-mails: S.Koulouzis@uva.nl A.S.Z.Belloum@uva.nl, R.S.Cushing@uva.nl
- Kostas Karasavvas is with the Netherlands Bioinformatics Center. E-mail:kostas.karasavvas@nbic.nl
- M. Bubak is with the AGH University of Science and Technology Krakow, Poland. E-mail: bubak@agh.edu.pl and the Institute for Informatics, University of Amsterdam. E-mails: M.T.Bubak@uva.nl

This work was partially funded by the VPH-Share project
for web services, in Section 3 we present the concept of ProxyWS, its functionality and the main modules. Section 4 describes the use cases which validate the proposed solution. In Sections 5 we discuss the obtained performance results, while in Sections 6 we give a summary of this work and an overview of future investigations.

2 RELATED WORK

Many different approaches have being introduced, in an attempt to address data transport problems web services face.

The Styx Grid Services (SGS) [11] is a service implementation based on the Styx protocol [12]. It exposes legacy command-line programs using a direct streaming approach for data exchange between SGSs. SGS can operate as a web service or within the Web Service Resource Framework [13]. However, this approach is limited by the Styx protocol for data transfers.

The Data Proxy Web service [14] is a solution for orchestrating workflows. This service wraps existing web services and SOAP messages are directed to the Data Proxy Web service instead of the target service. This enables large SOAP messages coming from existing web services to be referenced via an HTTP URL. Although this approach solves the problem of passing large data to consuming services, it does not address the situation where data need to be stored or obtained from remote locations. Moreover, constructing large SOAP messages in a memory may lead to memory exhaustion.

Flex-SwA [15] extends the SOAP with attachments specification [16] while preserving the interoperability of the existing specification. Flex-SwA attempts to increase the flexibility of bulk data transmissions for web services by using data protocols such as TCP, GridFTP, and BitTorrent. This approach enables new service implementations to exchange large data between each other and from remote storage locations, but it is not suitable for legacy web services.

A solution presented in [8] addresses the shortcomings of orchestrating data-centric workflows and proposes a choreography language called Multiagent Protocol. To enable legacy web services to participate in a choreography model, the authors propose the deployment of a choreograph interface at the location or near a target web service. However, this approach does not address the inefficiencies of SOAP or the issue of accessing data from remote locations.

XML compression may be used to reduce the size of SOAP messages, however, if not used properly it is computationally expensive, and often, performance gains are not significant [5]. In addition, compression is not a scalable solution since it only postpones the problem. Eventually the sheer size of data produced will cause the service or the enactor to crash, regardless of the compression ratio achieved. Finally, SOAP was not designed as a data transport protocol but rather as an invocation protocol.

3 THE PROXYWS FUNCTIONALITY AND MODULES

3.1 Concept and implementation

We propose an approach which addresses the shortcomings described in Section 2 by providing a mechanism for delivering data directly to web services which can either be a legacy or a new web services. The functionality of this solution should manage to circumvent data passing through enactors and to enable data stream processing and accessing data from variety of storage resources. It should be implemented as a web service to avoid installing any modified SOAP engine that can support data transfers with protocols other than SOAP. Such approach should enable proxy SOAP calls to existing web services as well as direct streaming if used as an interface.

We implemented the proposed solution as a web service called ProxyWS. Its main modules are: a web service frontend to handle SOAP calls, a client able to access any data resource, and a server for direct streaming. Figure 1 shows the functionality of the ProxyWS while deployed in a service container. On one hand the ProxyWS can provide larger data transfers for legacy services and on the other hand data streaming for new service implementations that use the ProxyWS’s interface.

The frontend of the ProxyWS is a web service implementation which enables web service consumers to maintain their application logic without significant modifications. This interface performs calls to legacy web services with the use of its main method –called callService–. This method is meant to invoke target services deployed in the same container. Incoming calls to callService, may provide URI references to large data. To facilitate data transfers from remote locations, the ProxyWS uses a Virtual Resource System (VRS) [17] which has a Java API to enable homogeneous access on multiple resources used for Grid computing such as file systems (local file systems, SFTP, GridFTP, etc.) and HTTP(S). The VRSServer is a simple module that enables the ProxyWS to stream data to other data consuming resources, including other ProxyWSs.

3.2 Scaling data transfers for legacy web services

To call a legacy web service, the ProxyWS should be deployed in the same container as a target web service so there is no need to modify deployed services while preserving the benefit of diverse data access and large data production. To call a legacy web service via the frontend the callService method is used. More specifically, a proxy call takes the following steps: (1) The client invokes the frontend via a SOAP call providing the necessary arguments needed for the target service. These arguments can be references to data that point to remote locations. (2) When receiving the call, the frontend resolves any references and retrieves them with the VRS. Next, the frontend will replace the references
in the call with the actual data for invoking the target service. (3) The frontend locates the target service and invokes its method. As soon as the target service has returned the result the frontend passes the data either to the VRS where they can be consumed via streaming or to a remote location such as a GridFTP server, with the use of the VRS. (4) The frontend returns a SOAP message containing a URI referring to the data location.

### 3.3 Data Streaming between web services

The ProxyWS can also be used as an interface to implement data streaming web services. The following steps are executed to establish a data stream connection: (1) The client creates an asynchronous SOAP call to the producing web service to start generating data. (2) The producing web service returns the URI where the data will be available. This URI points to its VRS server. (3) The client creates a SOAP call to the consuming web service, with the location of the produced data (the URI provided by the producing service). (4) The consuming web service obtains a data stream to the producing web service and processes it.

### 4 Examples of usage of ProxyWS

#### 4.1 Scalability tests

To verify and compare the performance and scalability of the proposed solution two simple web services were created: the first produces a random string of predefined size, the second encodes that string and returns the encoded results to the enactor. This application is used to measure three different approaches while increasing the string size of the producing service. In the first approach we used only SOAP so the enactor first receives the string from the producing service, then it passes it to the consuming service, and finally it retrieves the encoded string from the consuming service. In the second approach we carried out the same steps but this time the ProxyWS is used to reference the data. Finally, in the third approach the two services where re-implemented using the ProxyWS to enable data streaming, so the consuming service encodes the generated string as it is produced from the corresponding service.

Although this application is simple, it encapsulates typical scenarios of data-intensive web service applications.

#### 4.2 Indexing Name Entry Recognition

Indexing is a computationally and data intensive procedure which analyzes and extracts content from documents. An index can be searched given a query and the results obtained can be processed for Named Entity Recognition (NER). NER aims to extract and classify information units in a text such as names, biological species, numeric expressions, etc. [18]. The indexing applications provided by the AID project [19] enables the indexing of Medline documents obtained from the Medline bibliographic database produced and maintained by the U.S. National Library of Medicine. Medline’s annual dataset contains more than 17 million publication documents and requires approximately 64 GB of disk space. To enable users to perform queries on Medline indexes, AID provides a search web service (named SearchWS) that returns a maximum number of results relative to a query. The number of the results returned is set as an argument to the SearchWS. This input argument has a maximum limit at the initial web service imple-
Figure 2: Indexing and Name Entry Recognition use case. This figure depicts three different approaches: SOAP, proxy, and streaming. For the streaming approach the SearchWS and NERWS services are re-implemented with the ProxyWS interface.

The first version of the search and NER phase uses only SOAP, to perform steps 7.1 and 8.1 shown in Figure 2. In more detail the use case includes the following: (1) The enactor invokes the SearchWS to query the index. (2) The SearchWS returns the results to the enactor via SOAP. (3) The enactor passes these results to the NERrecognizeWS, and obtains the recognized text via SOAP.

The second version of the search and NER phase performs the same actions, but instead of calls being directed to the SearchWS and NERrecognizeWS, they are directed to the ProxyWS, and are executed as follows (see steps 7.2 to 9.2 in Figure 2): (1) The enactor invokes the ProxyWS on SearchWS side to query the index. (2) The SearchWS returns the results to the ProxyWS. The ProxyWS references the returned data and returns that reference back to the enactor via SOAP. (3) The enactor passes this reference to the ProxyWS of the NERrecognizeWS side that will obtain the results directly from the SearchWS, and invoke the NERrecognizeWS to recognize the text. (4) Finally the results created by the NERrecognizeWS are delivered to the enactor via SOAP.

Finally, in the third version we use the ProxyWS to re-implement the two services, and create a direct data stream between them. This data stream is used to pass documents relative to a query from the SearchWS to the NERrecognizeWS, and is executed as follows (see steps 7.3 to 10.3 in Figure 2): (1) The enactor invokes the SearchWS to query the index. (2) The SearchWS returns a URI where the data stream can be obtained, and starts feeding the stream with results. (3) The enactor invokes NERrecognizeWS to start processing the documents coming from the SearchWS by providing the URI so that the two services can establish the data stream. (4) Finally the results created by the NERrecognizeWS are delivered to the enactor via SOAP.
5 RESULTS AND DISCUSSION

In the scalability tests described in Section 4 the services were deployed in Axis service container [20] on nodes of the same network, with Intel Dual-Core E2180 CPU at 2 GHz and 2 GB of RAM. The enactor performing the invocations is hosted in a similar node but with 1 GB of RAM.

For the scalability tests we measure the execution time as a function of data size. For each of the three approaches, (SOAP, ProxyWS, and streaming) the producing service is set to return a string of 1 up to 1000 MB. The consuming service is set to return the encoded string back to the enactor which is approximately 34% larger than the initial string. As shown in Figure 3, the SOAP approach failed to transfer data larger than 2 MB under our specific hardware setup. This is because the enactor runs out of memory while trying to obtain both SOAP messages which for a string of 4MB would mean that the enactor had to receive approximately 9.4 MB. Using the ProxyWS we could reach data sizes up to 18 MB. This increase is because the enactor no longer needs to first receive the random string, passes it to the consuming service, and then receives the encoded string. Instead the random string goes directly to the consuming service. This approach, however, is limited because the entire message has to be instantiated in the enactor. Streaming had no problem reaching 1000 MB because the consuming service and the enactor were able to consume the string as it is being produced. Moreover, the results reveal that the time differences between the three approaches for the same data size are negligible, which is a result of hosting all web services on the same network.

Next, we present the results for the indexing phase of the use case described in Section 4.2. Figure 4 shows the total execution time, pre-stage time, execution time and post-stage time. The most time-consuming step of the indexing operation is the post-stage phase. This dramatic drop of performance is more evident when 32 nodes are used where the post-stage time makes up for most of the execution time. This is attributed to the I/O latency at the IndexerWS performing multiple index merge of smaller sub-indexes in order to construct the universal index.

Finally, we investigated the last two phases of the indexing use case. For each of the three approaches (SOAP, ProxyWS and streaming) the experiment is repeated for 100 up to 8300 documents (the number of documents returned by the SearchWS), using the same query. Figure 5 shows that SOAP failed to scale for more than 1100 documents. This is because the enactor could not cope with the size of the two returned SOAP messages (the query results and the NER). A possible solution for enabling larger data to fit into SOAP would be to increase the memory on the enactor and of the service container. This however is not a scalable solution as the enactor would eventually run out of memory when even larger data would be produced. Proxying and streaming managed to cope with the data requirements of 8300
documents. This is attributed to the data referencing capabilities of ProxyWS which makes it possible for data to be sent directly to the consuming services. As depicted in Figure 5, SOAP seems to have a faster rate of increase in terms of execution time. This is an indication that as the SOAP message becomes larger the commutation also becomes costly. Streaming on the other hand retains a much slower rate and proxying seems to be between the two approaches. It would be possible to optimize the execution time of the streaming approach by synchronizing the streaming rate, with the processing rate of the NERecognizeWS.

6 Conclusions and Future Work

From our experiments, we found that ProxyWS is able to successfully facilitate the data transport for indexing, where normal SOAP messages would have failed. Additionally, our approach does not require services clients to modify their application logic or the way web services are used. This enables web services to take advantage of SOA while also implementing data-intensive applications.

Through ProxyWS, data transfers in legacy web services were shown to scale better by first avoiding data delivery via SOAP, and second by delivering data directly from producing to consuming web services. This approach can also be beneficial in orchestrating workflows since data circumvents the enactor. Additionally we showed that with the creation of streams, it is possible for web services to produce and process even larger data that is potentially faster than exchanging SOAP messages. Moreover, the ProxyWS is an easy solution to adopt. Being implemented as a web service, it can be deployed in any Axis container, and provide data transports on behalf of any service within that container.

As a next step we will apply our solution on RESTful services. REST has emerged as a popular alternative to SOAP mainly due to its simpler interfacing since any HTTP client can be used to access and invoke RESTful services. RESTful services are accessed through standard HTTP GET, POST, PUT, DELETE methods. With REST, messages tend to be less bloated than their SOAP counterparts and thus REST can be more efficient in transporting data [21]. However, transporting large datasets between RESTful scientific services is still a challenge especially when considering workflows of services. A common approach to orchestrating scientific web service workflows is that a SWMS pipes the data between service invocations which could lead to a bottleneck due to the large datasets being transported through the SWMS enactor. In this case the model behind ProxyWS is also applicable to RESTful services since references to data rather than the actual data is encapsulated in the HTTP response. A REST-capable ProxyWS can then apply the same techniques described above to transport large data between RESTful services. Streaming between RESTful services can also be accomplished using the ProxyWS model. In this case the URI of the data on the VRS Server is returned by the producer, encapsulated in the HTTP message, and the enactor invokes the RESTful consumer by passing the HTTP encoded URI of the data.

References