Enabling Data Transport between Web Services through alternative protocols and Streaming

Spiros Koulouzis skoulouz@science.uva.nl
Edgar Meij E.J.Meij@uva.nl
M. Scott Marshall S.Marshall@uva.nl
Adam Belloum A.S.Z.Belloum@uva.nl
Informatics Institute
ISLA
University of Amsterdam

Abstract

As web services gain acceptance in the e-Science community, some of their shortcomings have begun to appear. A significant challenge is to find reliable and efficient methods to transfer large data between web services. This paper describes the problem of scalable data transport between web services, and proposes a solution: the development of a modular Server/Client library that uses SOAP as a control channel while the actual data transport is accomplished by various protocol implementation, as well as a simple API that developers can use for data-intensive applications. Apart from file transport, the proposed approach offers the facility of direct data streaming between web services, an approach that could benefit workflow execution time by creating a data pipeline between web services. Finally, the performance and usability of this library is evaluated, under the indexing application that the Adaptive Information Disclosure Application (AIDA) Toolkit offers as a Web Service.

1 Introduction

Web services offer an appealing paradigm for developing scientific applications by providing interoperability and flexibility in a large scale distributed environment. Through the use of XML based protocols (SOAP) and interfaces (WSDL), web services can expose all or part of any application in a language independent fashion across heterogeneous platforms. Those features enable them to be combined in a loosely coupled way so that more complex operations may be achieved [23].

Scientific applications can be created from workflows by combining and coordinating a set of web services so that a more complex goal is met. In other words, a workflow brings together web services (and/or applications) in a consistent manner to provide a description of execution of a higher level application.

Currently, two approaches exist in workflow implementation: Service Orchestration and Service Choreography. Service Orchestration (shown in Figure 1) describes how web services can interact at the message level, including the application logic and execution order of the functionality exposed by the WSDL a web services provides [6, 17]. In orchestration, the process is always controlled by a workflow engine, so all invocations (and replies) are made by (and to) that workflow. On the other hand, choreography (shown in Figure 1) is more collaborative, because it describes the message exchange among interacting — yet independent — web services [1]. Regardless which architecture is chosen, any workflow execution can be effectively reduced into 3 stages: (i) generating or obtaining data, (ii) processing that data, and (iii) transferring or storing the results. Usually, those three stages are performed by an individual web service.

This scenario, however, presents a *data transport problem*. During orchestration, any call to a producing web service results in a reply back to the workflow engine, which then needs to transfer the data to a consuming web service. This results in unnecessary "data hops" between the web service and workflow engine. In the case of choreography, although data is delivered directly to the consuming web services, this is usually done through SOAP. SOAP, although suitable for service invocation, can be inefficient for data transport. Significant performance issues for web services can occur when binary data must be encoded in XML, which measurably slows down applications and absorbs bandwidth [18].

The problems mentioned above are typically addressed by the introduction of a third party file transfer. For example in grid environments, where data is moved around with

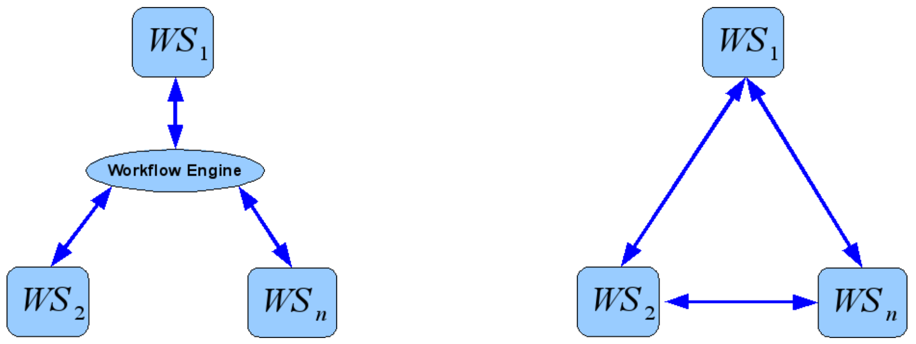

Figure 1. (left) Service Orchestration — web service calls are always controlled by a workflow engine and (right) Service Choreography — which describes the message exchange among interacting web services.

the help of GridFTP and the Reliable File Transfer (RFT) service [5]. This approach, however, might present another problem. Taking into consideration the three stages of the workflow mentioned above and a simple workflow shown in Figure 2. WS_1 would generate data, save it locally

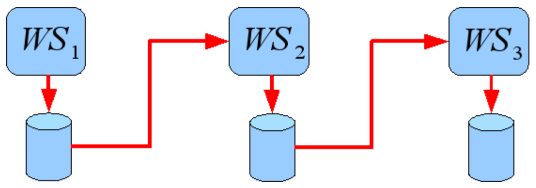

Figure 2. A simple workflow where web services exchange data through file exchange. This approach results in unnecessary temporary files.

and pass a file reference to WS_2. WS_2 would process the data received, save the result locally, pass the file reference to WS_3 and so on until the final result is obtained. In this process, temporary files are saved to each web service location, which results in an unnecessary demand on storage resources and slower execution time of the workflow.

An example of an application that follows this scenario may be seen from the Montage toolkit [14]. This application is developed in order to assemble science-grade mosaics, located at distributed file repositories, by composing multiple astronomical images. In other words, this application integrates multiple images taken from different parts of a galaxy in order to produce one representation of that galaxy. Apart from the complex algorithm that ensures that the separate images will fit together while preserving some vital data, the workflow of this application, seen in Figure 3, produces some intermediate images, that go on to further processing until they are composed into the final image.

Another application that fits into the simplified workflow execution mentioned above is the one proposed in [10]. In this application, web services are used for scientific visualization where the visualization pipeline[1] is broken down into a number of web services.

The *data transport problem* that web services face can be summarised in the following way:

1. In service orchestration, all data is passed to the workflow engine before delivery to a consuming web service.

2. Data transfers are made through SOAP, which is unfit for large data transfers.

3. third party file transfer is suitable for transferring large data sets, but is restricted to files. This results in unnecessary intermediate transfers that slow down workflow execution and place excessive demand on storage resources.

In order to address the above problems, this paper introduces the use of streaming between web services. First of all, it employs the approach of delivering data directly to a consuming web service with alternative protocols to SOAP, thus addressing the first two problems. Second of all, we describe streaming as a way to deliver data to a web service without the need for intermediate file transfers.

Our Streaming library enables web services to stream data directly to each other, creating a data pipeline (such as seen in Figure 4) that speeds up workflow execution and eliminate the need for allocating space on local disks. This library is realised by a simple design of a client/server architecture, that is contained in a web service and can take care of transfers using multiple protocols. Since it is essential that its use is as simple as possible, it provides a very

[1]A visualization pipeline entails the process followed for generating images from data.

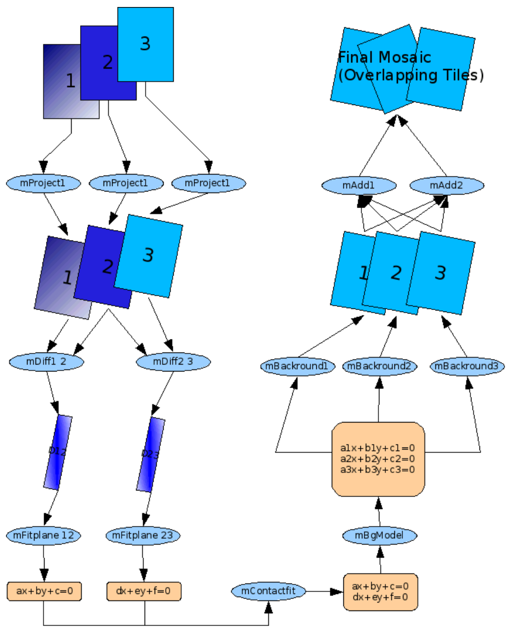

Figure 3. The Montage abstract workflow. In this workflow, there are three intermediate file transfers.

simple API to web service, that can use it as any other I/O stream, without having to worry about the transfer itself.

The remainder of this paper is organized as follows. Section 2 gives an overview of various protocols WS have at their disposal for data transport. Section 3 gives a description of our Streaming library, and how WS may use it for file transfer, or direct streaming. Next in Section 4 we give a description of the experimental setup, along with an overview of AIDA, an application that makes use of our library. Section 5 provides some performance measurements of our library, under two use cases. Finlay Sections 6 and 7, provide the future work and conclusions. The implementation and specifics are discussed in the following Sections.

2 Data Transport Methods

This section presents an overview of currently available transport methods and protocols that web services can use, for either file transfer or direct data streaming.

2.1 SOAP

SOAP is an XML-based protocol for exchanging messages over decentralized, distributed environments, and can potentially be used over a variety of protocols, although the most common implementation is made over HTTP(S). SOAP forms the basic messaging framework upon which

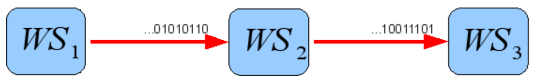

Figure 4. A simple workflow where web services communicate through a data pipeline.

web services may communicate, through several different types of messaging patterns. One of these patterns is the Remote Procedure Call (RPC), in which a client sends a request message to a web service. After processing that request, the web service sends back a response to the client. A SOAP message consists of three parts: (i) a mandatory envelope element, which is the top element of the XML document, (ii) an optional SOAP Header that defines how a recipient should process the message, and (iii) a mandatory SOAP body element, that contains the actual message, as it represents remote procedure calls and responses. [8,3].

Using SOAP to transfer data between web services is probably the most straightforward approach to sending and receiving data because no extra library development is required. In particular, current implementations (e.g. Axis) provide a solid framework for message delivery and error handing. Additionally, SOAP is platform independent: as long as the data types sent are supported by XML or have the appropriate schema definition, they can be handled by any application. Another advantage of using SOAP is that it makes it possible to add metadata to the message. Nevertheless, SOAP can introduce a significant overhead, thus slowing down its overall performance as a data transport protocol. In addition to the overhead of additional message size, the serialization and deserialization process of a SOAP message may cause additional performance delays [19].

2.2 SOAP with Attachments

SOAP with attachments (SwA) is an abstract model for SOAP, that enables the transport of binary formatted data along with a SOAP envelope. The SOAP message is still described by the three parts mention in Section 2.1, but with the addition of one or more Multipurpose Internet Mail Extensions (MIME) parts that are not defined in the SOAP envelope but are related to the message. Each MIME part contains some header information that may be used for identifying the type of the embedded data, the encoding used for this MIME part, and the content location. [21]. An alternative specification to MIME is the Direct Internet Message Encapsulation (DIME). The attached data in a DIME message is defined as a payload and a single message may contain multiple payloads. The contents of DIME messages are defined by records. Each record specifies the payload size in bytes, the payload content type, and other information [22]. SwA has a clear advantage over SOAP because at-

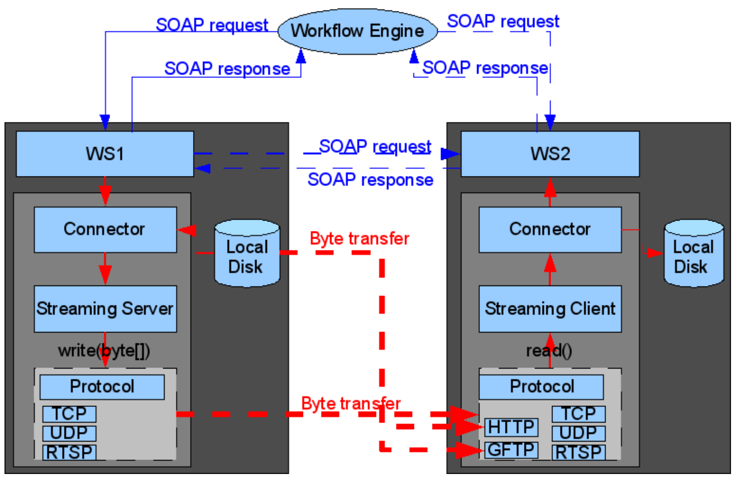

Figure 5. The Streaming library architecture. Dotted lines represent possible SOAP calls or byte transfer paths.

tachments don't require deserialization, while the message retains SOAP's advantages. Additionally, SwA can handle large data sets [22], but requires significant storage space for handling the attachment (at least in the Axis implementation).

2.3 HTTP

Both SOAP and SwA are, in most cases, transmitted over HTTP, which is a well-established and stable protocol that defines a set of headers for exchanging data. An abundance of high quality software is available that supports high speed transfers of large data sets in HTTP, without having to worry about firewalls. Furthermore, in the context of web services, Tomcat containers already handle HTTP communications, thus providing a solid framework for exploiting this facility. On the down side, HTTP does not provide any facilities for handling metadata or specify how that data should be handled by the recipient side. Thus HTTP may require some additional effort in implementing extensions that could deliver data to web services.

2.4 GridFTP

GridFTP is a protocol developed specifically for the grid environment. It is an extension of the FTP that supports security using public-key-based Grid Security Infrastructure(GSI) and Kerberos, parallel data transfer using multiple TCP streams, striped data transfer, automatic adjustment of the TCP window size, and data transfer monitoring [15,4]. Although GridFTP has set the standard for data transfers in grid environments, it is not usable in a web services scenario, where these are a loose collection of services, rather than a structured Virtual Organization [12]. In other words, most web service owners don't offer this kind of infrastructure. Another problem with using GridFTP is that developers would be restricted to this particular type of file transfer, as well as having to implement a client that would request files from a GridFTP server.

2.5 TCP

TCP is a low level protocol in which all of the previously mentioned protocols are implemented. It is the oldest and most established protocol, providing a reliable and in-order delivery of data that makes it suitable for a wide area of applications such as File Transfer Protocol, Secure Shell, and some streaming media applications. In order to offer reliability and in-order delivery, TCP assigns a sequence number to each packet. This number is used by the receiving end for ordering packets. Also the receiving side sends back an acknowledgement for packets that have been successfully received. Apart from re-transmitted unacknowledged packets, TCP also checks that no bytes are corrupted by using a checksum. TCP comes with a variety of tuning options that enable it to achieve full bandwidth utilization, and minimum delay [2,13]. Being a low level protocol, TCP should be ideal for transporting large volumes of data, with zero overhead, but it has no facility for metadata. Furthermore, it requires custom development in order to be incorporated in a web service. Extra attention to its tuning properties (e.g. appropriately configuring TCP buffer sizes) is necessary in order to achieve its full potential.

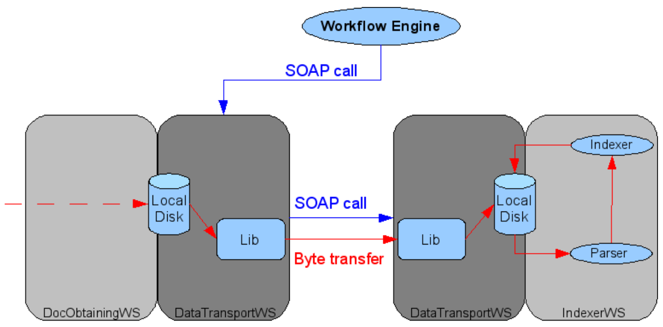

Figure 6. A use case for indexing documents, while using the Streaming Library. The documents are first transferred to the IndexerWS and then indexed locally.

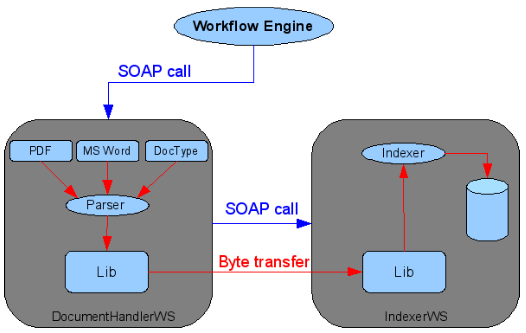

Figure 7. A use case for indexing documents, while using streaming. The contents extracted from documents are directly streamed to the IndexerWS.

3 The Streaming Library

In an effort to address the *data transport problem*, we have developed a library in Java that employs the protocols mentioned in Section 2. Our library provides a simple API that developers and legacy web services can use to transport data. The Streaming library (seen in Figure 5) is a modular, client/server design that uses SOAP as a control channel while the actual data transport is accomplished by the various protocol implementations, which are developed as plug-ins. The basic components that make up the library are:

3.1 Connector

This module provides the functionality offered by the Streaming client/server to a consuming/producing web service, in the form of a standard I/O stream. Alternatively, this module may use these streams to send or store data to or from a local disk.

3.2 Streaming Client/Server

The role of these components is fairly simple. The server only needs to send data received from the Connector module to a connected client, while the client passes the received data to the Connector. The details about authentication, and the actual transfer, are left to the connection component.

3.3 Connection

We abstract the connection component, such that different protocol implementations may be added in the design. For this reason, it is assumed that a connection offers a read and write method, as well as one that enables the authentication and/or the encryption of the channel used for the transfer.

3.4 Establishing a Data Stream

Having described the basic components that make up the design, it is time to see how these components interact to establish a data stream between two web services:

1. The workflow engine invokes the producing web service. That service will start producing data while passing them to the connector, which starts a Streaming Server.

2. At this point, there are two alternatives: (i) the producing web service might respond back to the workflow engine with a reference of the data stream[2] or (ii) pass the reference directly to the consuming web service as a SOAP call. In the first case, it is the responsibility of workflow engine to invoke the consuming method of the second web service, which will use the Connector module to connect to the server and get the data. In the second case, the producing web service will contact the consuming web service making the same call. For this second case, the workflow engine should provide the consuming endpoint to the web service.

3. If the protocol implementation provides authentication and/or encryption mechanisms and the stream reference instructs the client/server to do so, the connection between them would first have to authenticate each side. In the case where no authentication and/or encryption is required, the connection is established and the client starts receiving the data stream.

[2]This reference is an XML document, containing the server's address, port, and protocol specific configurations.

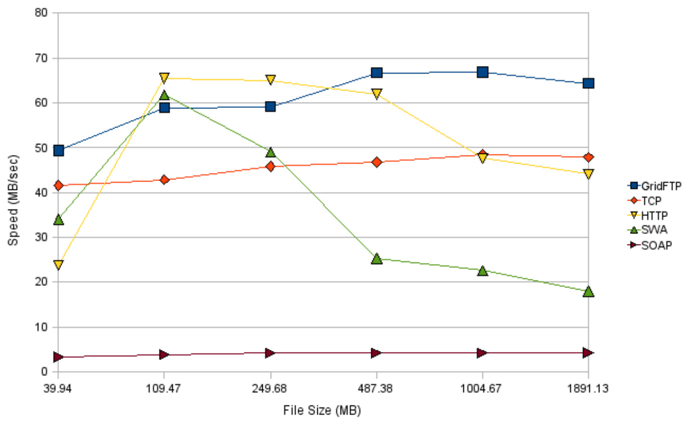

Figure 8. Speed measured in MB/sec, for file transport for GridFTP, HTTP, TCP, SwA and SOAP.

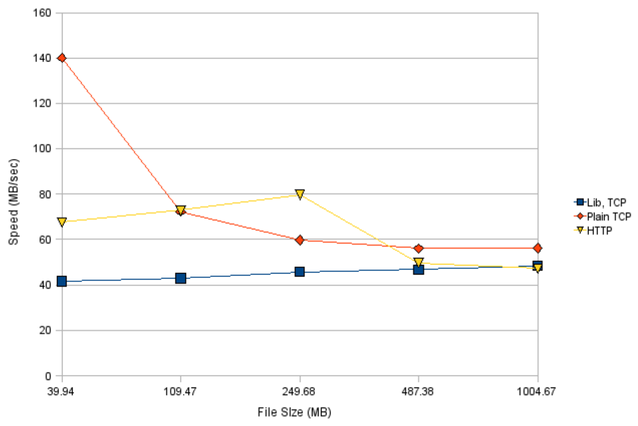

Figure 9. Speed measured in MB/sec, for file transport in a tuned TCP Java implementation, an HTTP, and an untuned TCP implementation (the current implementation of the library).

4. As soon as the producing web service is done producing data, it simply closes the stream provided by the Connector. As a consequence, the Connector must close the connection between server and client. The specific approach to closing the connection depends on the protocol implementation.

5. Finally, the producing web service should return information about the transfer to the workflow engine, such as any errors that might have occurred during that transfer.

3.5 Streaming web services VS File-oriented web services

As mentioned in Section 1, one of the problems web services face while transferring data using third party file transfers is the fact that temporary intermediate files must be stored as a side effect of data transport. Streaming data directly to web services could solve this problem, but there are a number of issues to be considered before adopting that approach. The main hurdle to adopting streaming comes from the nature of the application a web services implements. If, for example, the application is designed to operate with large files that somehow are loaned, processed and passed to a next web service, streaming might not be applicable. This is because the loading and processing overhead might be significantly larger that the actual transfer itself. If this is the case, streaming is only able to save storage capacity in computational nodes and does not improve workflow execution time dramatically. Furthermore when reliability is the issue, streaming could also prove inefficient. Consider the example where a web service provides data from a scientific instrument, (e.g. a telescope, or a scanner) and that web service uses streaming to deliver data to a consuming web service and that service consumes data directly. If the data stream is broken for any reason, all the data produced so far would be lost, thus resuming the consuming web service from the last known good checkpoint would be impossible. Another case where streaming is not applicable is when a consuming web service must obtain the entire data set before operating on it. Nevertheless, when a web service is designed to produce live data (e.g. cluster loads, or stock prices), streaming would significantly benefit the workflow execution time.

4 Experimental Setup

The Adaptive Information Disclosure Application (AIDA) Toolkit is a generic set of components that can perform a variety of tasks such as learn new pattern recognition models, specialized search on resource collections, and store knowledge in a repository. AIDA provides a set of components which enable the indexing of text documents in various formats, as well as the subsequent retrieval given a query. The Indexer and Search components are both built upon Apache Lucene, version 2.1.0 [16]. AIDA's Indexer component — called IndexerWS — is a webservice which is able to index[3] a variety of document formats while taking care of the preprocessing (the conversion, tokenization, and possible normalization) of the text of each document as well

[3]Indexing is the process of analysing and extracting content from a document. These contents are stored in an index in order to optimize the speed and performance of finding documents relevant to a search query

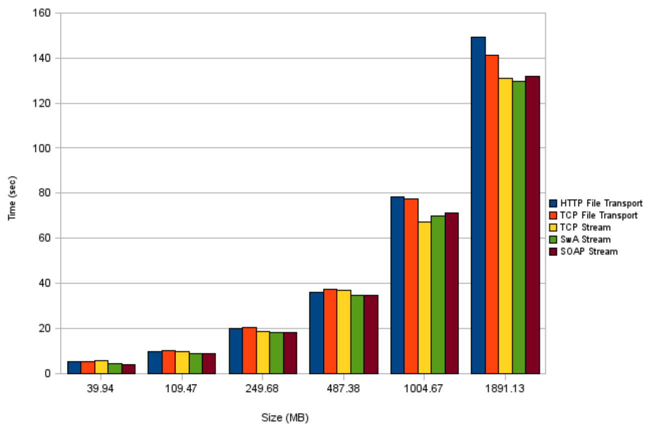

Figure 10. Workflow execution time for various file sizes. The two first bars for every file represent execution time for file transport with HTTP and TCP respectively

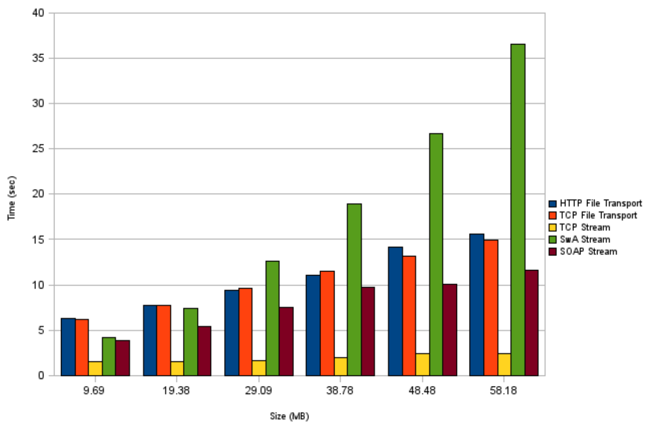

Figure 11. Workflow execution time for generating various file sizes. The two first bars for every file represent execution time for file transport with HTTP and TCP respectively.

as the subsequent index generation. The so-called "DocumentHandlers" which handle the actual conversion of each source file are loaded at runtime, so a handler for any other proprietary document encoding can be created and directly used [20]. The task of the IndexerWS might be potentially data intensive, in which case SOAP is not able to meet the IndexerWS's demands. To enable the IndexerWS to index a set of documents, the Streaming library was utilized for two use cases:

1. A set of documents is obtained by a producer web service (e.g. from a database), and transferred to the IndexerWS for indexing.

2. A PDF DocumentHandler is implemented as a web service, for extracting text from a set of PDF files. This text is streamed directly to the IndexerWS for indexing.

For the first case, illustrated in Figure 6, a Data Transport web service was developed that is able to transfer a set of files to the IndexerWS location, in a third party file transfer manner, using the API provided by the Streaming library and Axis. For the second case seen in Figure 7, the PDF DocumentHandler and the IndexerWS used the APIs to stream data between them.

In order to assess the performance of each protocol and transfer method in the two use cases mentioned in Section 4, the IndexerWS and the DataTransportWS have been deployed[4] on the DAS-3 Distributed Supercomputer [11]. For the DataTransportWS four protocols were tested in terms of speed. This metric was acquired for disk-to-disk transfers. For the IndexerWS, two simple workflows were developed. The first workflow transfers a set of PDF files from a producer location, to the IndexerWS, which is then invoked to start the indexing process. The second workflow, invokes a PDF DocumentHandler that extracts the content from a set of PDFs and directly streams it to the IndexerWS. These two workflows were measured in terms of execution time [5].

[4]All web services were deployed in Axis 1.4 running in Apache Tomcat 6.0.16

5 Results and Discussion

This section describes the actual performance results we obtain using our proposed approach on the two tasks: file and streaming transport.

5.1 File Transport

Figure 8 shows the transport speed for the protocols described in Section 2. As expected, GridFTP outperforms all of the protocols when transferring larger files, as it is now an optimized mature application. However, SwA and HTTP are faster than GridFTP in file sizes of 109.47 and 259.48 MB. This could be explained, by the fact that HTTP and in extension SwA has zero start-up time, since many web servers and, in this case Tomcat, do not close the connection after the first file request resulting in better performance for the subsequent transfer. For the remainder of the files, HTTP exhibits lower speeds. This is probably because

[5]In all tests, the producer was located at the cluster site at the University of Amsterdam, while the consumer was at the Vrije Universiteit, also located in Amsterdam

disk I/O starts to introduce a significant overhead (at least in the library's current implementation). Disk I/O latency affects SwA more than any other protocol, since SwA has to first save the attachment into the local disk, and then save it again with the specified name. As a result it could be said that SwA misuses storage resources. Although one would expect TCP to have at least the same performance as HTTP, this is not the case. Start up time is probably the cause of the speed reduction. Our TCP implementation uses a separate server to transfer data, and its start up time is approximately 60 msec. Additionally, the lack TCP tuning parameters, is responsible for this suboptimal performance, together with the blocking read/writes. This may be seen Figure 9, where a simple Java TCP file transfer is compared to our implementation, as well as with the HTTP. The results for this simple Java TCP file transfer were obtained by trying various sizes for the TCP buffer. Furthermore, the performance of this simple file transfer was even better when the delay introduced by reading the file was eliminated[6]. Thus, if the read/write of a file was done in a non-blocking way, the TCP performance would reach the expected levels. When looking at SOAP's performance, it is dramatically lower than any other protocol. This is because SOAP introduces a significant amount of overhead in each message. Additionally, in order to prevent SOAP from crashing, "chunking" had to be introduced. In this approach, the file is sent into small chunks of data (approximately 15MB), preventing crashing and bandwidth utilization. Finally, all protocols except SOAP exhibit some speed reduction for file sizes of 1891.13 MB. In this case the transfer concerns five separate files, thus introducing the latency of five discrete requests, a problem that is identified and addressed in [9].

5.2 Streaming Transport

As mentioned earlier, two simple workflows were developed for measuring workflow execution time in file and streaming transport. Figure 10 shows the execution time, of the complete indexing operation (obtain the contents, analyze them, and save them to an index). For each file size, the first two measures concern the case where the PDFs are transferred to the IndexerWS and indexed (file transfer). The rest of the measures were obtained by streaming the PDF contents directly to the IndexerWS. Although even SOAP performs better than any other file transfer, the time difference in execution time, is no more than 20 sec in a total execution time of approximately 140 sec. This small improvement may be attributed to the time needed to load a PDF file. When eliminating the time necessary to load the PDF's, the performance of streaming is much better than the one of file transport. In Figure 11, execution times were obtained by having a producing web service generate a simple text file and sending it, for the file transport case, while for the streaming case, the text was directly streamed to the IndexerWS. All protocols except of SwA outperform HTTP and TCP file transfers. SwA's low performance may be again blamed on the way it handles messages. SwA gets slower as more attachments are introduced to the SOAP message and because they are first saved to the local disk and then read by the consumer, SwA proves more inefficient than just transferring the file and then index it. GridFTP was not compared in this scenario, as the nodes able to run a Tomcat container didn't offer a GridFTP server, while the ones offering a GridFTP server could not run a Tomcat container, or didn't have adequate storage space.

6 Future Work

The work presented in [7] proposes an interesting approach in workflow development. More specifically the introduction of a proxy web service in the vicinity of producing web service would make sure data delivery directly to a consuming web service. The combination of the Streaming library and this proxy web service, could enable legacy web services to exchange large data sets, while using the most appropriate protocol for optimizing performance. This approach, however, calls for some investigation of whether SOAP is fast enough at delivering data to web services located in the same container. Another approach that would enable legacy web services to transfer large data sets, could be the extension of Axis through some plug-in, that would transform a SOAP message containing data to a stream reference, thus using an alternative route for data transport. As HTTP offers an attractive solution for file transport, the need to develop a mechanism that would also enable direct data streaming is apparent. Additionally integrating an XML header to streams, would enable developers to include more information (e.g. metadata) in those streams. Another benefit from the inclusion of XML headers in the stream would be the potential optimization of the streaming performance for multiple file transfers, because the current implementation of the streaming library starts a new connection for every file request. On a higher level, the introduction of a registry service which producing web services could use to register stream and file references, would offer more flexible workflow designs in terms of data transport.

7 Conclusions

We have identified and addressed the problem of transferring large data sets between web services. We have described a modular and extensible Streaming library with a simple API that is able to transfer large files, as well as connect web services in a continuous data pipeline as an alternative approach to data transport. In our proposed approach,

[6]This was done by just reading data from /dev/zero

SOAP is used as a control channel, while data is transferred using the most suitable protocol for either file or streaming transfers. In addition, the use of streaming could potentially speed up workflow execution time by eliminating disk I/O latency and enabling web services to work on data as it is generated, rather than waiting for an entire file to be delivered.

8 Acknowledgment

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). Part of this project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science and is part of the ICT innovation program of the Ministry of Economic Affairs.

References

[1] W3C. Web Service Choreography Interface (WSCI) 1.0. http://www.w3.org/TR/wsci/.

[2] Advanced Networking Pittsburgh Supercomputing Center. Enabling high performance data transfers, 2002.

[3] Asif Akram, Rob Allan, and David Meredith. Best practices in web service style, data binding and validation for use in data-centric scientific applications. In *UK e-Science All Hands Meeting 2006*, September 2006.

[4] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. In *Parallel Computing*, 2002.

[5] William E. Allcock, Ian Foster, and Ravi Madduri. Reliable data transport: A critical service for the grid. In *In Building Service Based Grids Workshop, Global Grid Forum 11*, 2004.

[6] Adam Barker and Jano van Hemert. Scientific workflow: A survey and research directions. In *Proceedings of the Third Grid Applications and Middleware Workshop (GAMW'2007)*, LNCS, page In press, 2007.

[7] Adam Barker, Jon B. Weissman, and Jano van Hemert. Orchestrating data-centric workflows. In *CCGRID*, pages 210–217, 2008.

[8] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple Object Access Protocol (SOAP) 1.1, 2000.

[9] John Bresnahan, Michael Link, Rajkumar Kettimuthu, Dan Fraser, and Ian Foster. Gridftp pipelining. In *Proceedings of the 2007 TeraGrid Conference*, 2007.

[10] S. Charters, N. Holliman, and M. Munro. Visualization on the grid: A web service approach. In *Proceedings UK eScience third All-Hands Meeting*, 2004.

[11] DAS-3. http://www.cs.vu.nl/das3/.

[12] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Jounral of Supercomputer Applications*, 15(3), 2001.

[13] M. K. Gardner, S. Thulasidasan, and W. Chun Feng. User-space auto-tuning for tcp ow control in computational grids. *Computer Communications*, 2004.

[14] J. C. Jacob, D. S. Katz, G. B. Berriman, J. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams. Montage: A grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 2006.

[15] Rajkumar Kettimuthu, William E. Allcock, Lee Liming, John-Paul Navarro, and Ian T. Foster. Gridcopy: Moving data fast on the grid. In *IPDPS*, pages 1–6. IEEE, 2007.

[16] Lucene. http://lucene.apache.org.

[17] Chris Pelz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, October 2003.

[18] W3C. Three Web Services Recommendations. http://www.w3.org/2005/01/xmlp-pressrelease.html.

[19] Robert van Engelen. Pushing the soap envelope with web services for scientific computing. In *ICWS*, 2003.

[20] Adaptive Information Disclosure web site. http://www.adaptivedisclosure.org.

[21] W3C. SOAP Messages with Attachments. http://www.w3.org/TR/SOAP-attachments.

[22] Ying Ying, Yan Huang, and David W. Walker. A Performance Evaluation of Using SOAP with Attachments for e-Science. In *Proceedings of the UK e-Science All Hands Conference*. Engineering and Physical Sciences Research Council, 2005.

[23] Jianting Zhang, Ilkay Altintas, Jing Tao, Xianhua Liu, Deana D. Pennington, and William K. Michener. Integrating data grid and web services for e-science applications: A case study of exploring species distributions. *e-science*, 0:31, 2006.