Streaming data between Web Services.
Comparison of streaming protocols over a stream-enabled Web Service.

Spiros Koulouzis <skoulouz@science.uva.nl>,
Evangelos Angelou <eangelou@science.uva.nl>

Grid Computing Master's Programme
University of Amsterdam

Contents

1. Introduction **3**

2. Data Transfer Protocols **4**

3. Design and Implementation of a streaming enabled WS **5**
3.1 Design Overview . . . 5
3.2 Basic Component Description . . . 6
3.3 Implementation Issues . . . 8

4. Testing Methodologies **12**

5. Results **13**

6. Conclusion **14**

Appendices **19**

A Full WS Sequence Diagram **19**

B How to use the Streaming Library with the Template WS **22**
B.1 Overview . . . 22
B.2 Prerequisites . . . 22
B.3 The Template WS . . . 22
B.4 The Template Resource . . . 23
B.5 The Backend Application . . . 24
B.6 Developing and Testing a streaming enabled WS in 11 steps . . . 25

List of Figures

1 The Montage abstract workflow. In this workflow there are three intermediate file transfers. . . . 4
2 An example workflow. . . . 6
3 A visualization pipeline from a darkmatter simulation . . . 7
4 Design of the stream library and the producing/consuming WS. The doted line indicates the flow of data from the producing to the consuming backend application . . . 8
5 The sequence diagram on the consuming side. . . . 10
6 The sequence diagram on the producing side. . . . 11
7 TCP and RTSP speed. . . . 14
8 Speed of the total transfer through UvA. . . . 15
9 RTSP speed as measured at the UvA. . . . 15
10 UDP speed as measured at LU. . . . 15
11 UDP speed as measured at the UvA. . . . 16
12 The two choices to transfer data. . . . 17
13 The full sequence diagram. . . . 21

List of Tables

1 Transfer speed measured in local cluster. . . . 13
2 Remote transfer speed UvA-VU. . . . 13
3 Remote transfer speed TU Delft - LU. . . . 14
4 Three party remote transfer speed TU Delft - UvA - LU. . . . 14

Abstract

The ability to stream data between web-services is vital for the implementation of complex workflows on the Grid. In this work we investigate various protocols as to their suitability for streaming, taking into consideration issues such as security, reliability and speed over different Grid configurations. To perform these comparisons we have implemented a Server/Client web service that provides a simple API through which Java applications can stream data. As an added bonus, this architecture can be reused by scientific programmers who want to stream data without having to deal with protocol or web service trivialities. Finally, an evaluation of streaming as an alternative to file transfer on Grid environments is offered, based on a number of test case scenarios.

1. Introduction

The e-science community is moving towards web services (WS) to develop its applications. This is because web services are able to offer a flexible, platform independent and a large scale distributed environment. Additionally web services, and more specifically Web Services Resource Framework (WSRF) should be able to cope with the intensive computational demands of such applications. But just computational resources are not enough to develop a sophisticated e-science applications. Data transfer in grid environments is proving to be as important as gaining access to computational or specialized resources [1]. The increased computational possessing power a grid environment offers, has as an outcome an also increased size of data sets. Thus data produced in e-science applications are in the scale of terabytes, and are not aways the final result. An example of such an application may be seen from the Montage toolkit. This application is developed in order to put together science-grade mosaics, located at distributed file repositories, by composing multiple astronomical images. In other words this application puts together multiple images taken from different parts of a galaxy in order to produce one representation of that galaxy. Apart from the complex algorithm in use, that makes sure that the separate images will fit together, while preserving some vital data, the workflow of this application, seen in Figure 1, produces some intermediate images, that go on further process until they are composed into the final image [2]. Considering Montage, that falls into a e-Science application framework, a need is created for efficient and reliable data transfer between web-services.

As seen from the Montage application a workflow is need to coordinate all the separate actions that participate to produce the final outcome. Thus workflows are used to describe the execution of complex applications, that are made from individual modules which are designed to perform specific tasks, in this case WS in different processing sites (containers [1]) [3]. Providing a more formal definition of a workflow: "the term workflow can be defined as the orchestration of a set of activities to accomplish a larger and sophisticated goal, referred as a business process" [4].

The traditional paradigm of running a Grid workflow could be reduced into 2 steps: where to process data and how to move data between processing sites. The first consideration is usually taken care by creating the appropriate application bundled as a WS. The second by using buffer files on each computational node and then moving them around with the help of GridFTP and the Reliable File Transfer (RFT) service. It has been argued that adding a simple streaming mechanism that can be used efficiently and reliably by a WS can speed up the execution time of such a workflow [5]. Intuitively such a mechanism would also shorten the development time of a grid application, since the user needs not take care of each data transfer.

Although much work has been done on streaming, this hasn't reach the grid comunity. A large number of publications deal with the problems of plain TCP for data streaming over modern network infrastructures and how to overcome them [6, 7, 8, 9, 10]. Some of their authors propose new protocols that provide better performance under certain circumstances, such as UDP based Data Transfer (UDT) [11, 12], or improvements on the basis of TCP, such as FAST-TCP, Fast Data Transfer (FDT) [13] etc. Protocols for data streaming have been developed and established, such as Real Time Streaming Protocol (RTSP) and Real-time Transport Protocol (RTP) [14] for media streaming, although they are little mentioned outside the multimedia domain.

All of these protocols could be used for streaming, but the added requirements of the Grid, force the majority of applications to resort to a "hopping" data transfer policy, in which each separate instance/WS has to wait for the completion of its task and then transfer the data using RFT/GridFTP. As a consequence, this adds up the execution and transfer time, as opposed to a strategy that by streaming the data to the next WS as soon as they become available would save idle execution time. In order to address this problem without tampering with the underlying transfer protocol (usually TCP), a number of

[1] A container is a server that allows clients to access WS via HTTP. The most widely used container is Apache Tomcat that may run as a stand-alone application, and listen for HTTP requests on some port. An an additional software, the Apache Axis, holds the responsibility of delivering and transmitting SOAP messages to the appropriate WS

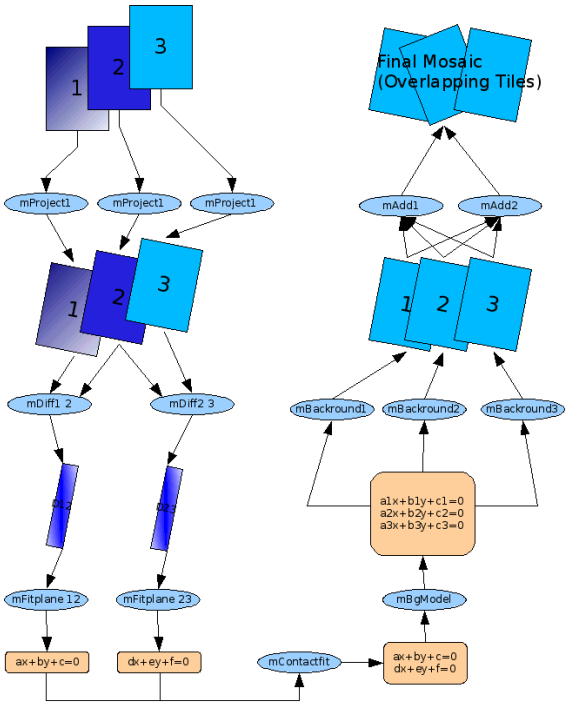

Figure 1. The Montage abstract workflow. In this workflow there are three intermediate file transfers.

projects such as Griddles [15, 16, 17] and STYX [18, 19] provide a way to deal with remote files as streams of data, with some small limitations. An extra benefit is that if set up properly such an approach could also Grid-enable much of the legacy source in existing projects that use file input-output streams.

It is obvious then that streaming data offers significant benefits to Grid workflows. The toughest part though is that Grid users think of integrating streaming capabilities to their application is much to burdensome and customized to be reused by other applications that the same Virtual Organization (VO) would produce in the future. For this we propose a simple design of a client/server architecture, that poses as a WS and can take care of transfers using a multitude of protocols. Since it is essential that its use is as simple as possible, it provides a very simple API to back-end applications, that can use it as any other I/O stream, without having to worry about the transfer itself. The implementation and specifics are discussed in the next sections.

2. Data Transfer Protocols

Aiming for a design that enables streaming in the framework of WS, some protocols were investigated. A variety of protocol implementations have being proposed, but almost all of them are some version of TCP or a combination of UDP and TCP.

TCP is the most established protocol, because it provides a reliable and in-order deliver of data, making 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 is assigning to each packet a sequence number. This number is used by the receiving end for ordering packets. Also the receiving side sends back an acknowledgment 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. These properties seem ideal for streaming in a grid environment, but some modifications might be necessary for full bandwidth utilization. Because TCP must acknowledge all the packets sent, there is some concern about the time spent on these acknowledgments. This drawback can be solved be appropriately configuring the TCP receive and transmit buffers size. [8].

Another well established protocol is the UDP. UDP is a connectionless protocol, and in contrast with the TCP is doesn't

offer in-order reliable data transfer. Instead each packet is sent on the network with no sequence number, and no acknowledgment is sent back from the receiver. Because UDP is a connectionless protocol, it is able to broadcast or multicast packets on a network. Furthermore by avoiding the overhead of acknowledgment UDP is faster and more efficient than TCP, at least for applications that do not need guaranteed delivery.

RTSP is a statefull protocol, used for media streaming. This protocol uses two channels, one for control, and the other for the actual data transfer. The control channel is implemented over the TCP protocol, and it is used to send commands to a media server (play, pause, stop). The data channel is using the RTP protocol, that extends the UDP, by adding a time-stamp and a sequence number to each packet for synchronizing the streaming media.

FDT is an application for Efficient Data Transfers which is capable of reading and writing at disk speed over wide area networks using standard TCP. It is based on an asynchronous, flexible multithreaded system and is using the capabilities of the Java NIO libraries. Its main advantage is that it streams a dataset (list of files) continuously, so that a large dataset composed of thousands of files can be sent or received at full speed, without the network transfer restarting between files, and transfers data in parallel on multiple TCP streams, when necessary. Its operation is asynchronous and provides the ability to resume a file transfer without loss. Although not a streaming application its inclusion in our tests was meant to establish the norm of file-transfer applications performance in comparison to streaming, since GridFTP alone, although the de-facto standard, is only one example.

The Styx protocol is a variant of a protocol called 9P that was developed for the Plan 9 distributed operating system from Bell Labs. In Styx everything is a file, be it a device that produces data as a stream, a local or remote actual file. In essence it is one of the best candidates to consider for streaming purposes as it was designed to facilitate network file transfer in real time for Plan 9. In that sense, it should provide an easy and reliable way to stream data over a network. The user would then have to deal only with the Styx programming API that is as intuitive as simple file input-output. Another advantage is that there is no need to provide reliability and extra mechanisms as the protocol takes care of this. Since there are both Java and Python implementations of Styx we are not obliged to use the Plan 9 OS and the Java implementation in particular provides platform independence.

The only protocol developed specifically for the grid environment is the GridFTP. It is an extension of the FTP that takes under consideration issues like security by using public-key-based Grid Security Infrastructure(GSI) and Kerberos support, parallel data transfer using multiple TCP streams, striped data transfer, automatic adjustment of the window size and data transfer monitoring [20], [1]. Although this protocol is not suitable for data streaming, it is examined because it clearly has set the standards for data transfers in grid environments.

UDT is an application level data transport protocol that uses UDP to transfer bulk data and it has its own reliability control and congestion control mechanism. It is therefore fast but without sacrificing reliability. It is designed for high bandwidth and delay product (BDP) networks, that is a configuration only too common in Grid computing environments. UDT is also fair and TCP friendly when multiple streams share the same channel. Taking all this into consideration, it is a promising candidate protocol for streaming on the Grid.

3. Design and Implementation of a streaming enabled WS

3.1 Design Overview

Web services are described as a distributed computing technology, that allows for the implementation of client/server applications. More specifically, they provide the means for working between different software applications, running on a variety of platforms and/or frameworks. They may be defined as self-describing modular applications that can be invoked over a network using the Simple Object Access Protocol (SOAP). Extending this idea, the WSRF is a framework that enables WS hold and manipulate a sate (e.g. the result of a previous addition, or the reservation state of a booking flight) [21].

Developing an e-Science application holds more than just writing a bunch of WS and deploying them in distributed processing nodes. As mentioned in the introduction workflows are used to orchestrate the flow of control and data. Thus an e-Science application must first produce, or discover raw data that may come from an experiment, or a specialized device, pass them for processing in some distributed framework that enables individual processing nodes to exchange data and control, and produce the final product of the application.

According to a simple but common example of such a process, described in [4],if a user wishes to run an e-science application, as illustrated in Figure , she/he needs to transfer a set of data from resource A, to resource B. After that the output from resource B is stored locally, and then transferred into resource C. Finally after resource C has processed the final result data are stored back in resource A. It is evident that in this process, the data flow is made through file transfers, that

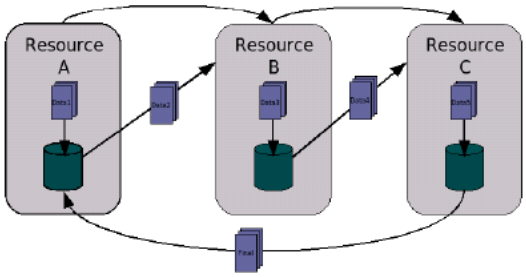

Figure 2. An example workflow.

are not the final product of this workflow. Enabling streaming between WS is a solution to this problem, as it would skip unnecessary file transfers.

A real application that fits in the simplified example above is the one proposed in [5]. In this application, WS are used for scientific visualization, where the visualization pipeline (the process followed for generating images from data), is broken down to WS. As seen from the pipeline in Figure 3 the separate stage of the visualization are assigned to a WS that acts as a frontend for that visualization process. In this pipeline data from one visualization stage to the next are transferred via files with the use of HTTP file transfer something that the authors see as problematic, "To efficiently visualize very large data sets it is felt that all services in the pipeline should be able to operate on data when it is ready rather than operate in a batch mode... a streaming mechanism that can be easily integrated with a service do not exist yet and so work arounds (currently HTTP Streaming File Transfer) have been used".

From the scenario described above and from the visualization application, two entities are identified. The producing and the consuming side. These two entities are encapsulated in the form of a WS that holds a backend application that either needs to consume or produce data. An approach that enables the producing side to stream data to the consuming side, is the development of a steam server/client streaming library that may utilize the wealth of all the available protocols to transfer data between these to WS. In this design there two separate parts: the streaming library, which encloses a backend application, a server, a client and a connection. The second part of the design, is front-end WS that manipulates the application, the client and the server, depending on the role it has being assigned (producing or consuming). The aim of the front-end WS in not only to be able to manipulates the modules of the streaming library, but also to offer an easy API where future developers may easily use extend. In this sense the WS is a template that can take care of data transfers transparently, while the streaming library is an extensible modular design that can be adapted to the specific needs of an e-Science application (e.g. speed, reliability, security).

Developing a streaming library that will work in a grid environment, one should consider the following issues:

- Authenticate producing and consuming entities. Authentication plays a key role in data transfer, as it ensures that the "sensitive" data is not going to end up in the wrong hands.
- Reliable transfer of data. As data sets in e-Science applications, get bigger and more complicated, there must be a way of ensuring that a large data set will reach its destination, and the underlying transfer mechanism will be able to cope with faulty or noisy channels.
- Full Bandwidth utilization. Usually the grid environment, is characterized by high speed channels, a feature that should be fully exploited if one wishes to take advantage of the processing power offered by such a distributed environment.
- Secure data transfer. Apart form being fast reliable and sure of were the data are going, a data transfer mechanism should provide privacy. The channel that transfers data should be secured with encrypting mechanisms

3.2 Basic Component Description

Bearing in mind the points mentioned above the streaming library together with the template WS is designed with the following components:

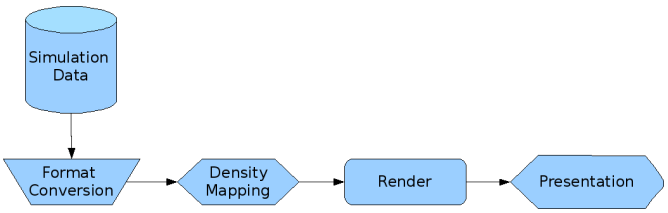

Figure 3. A visualization pipeline from a darkmatter simulation .

WS Controller. This component coordinates the two WS. In a real application this is the role of the workflow, so this component is developed for testing reasons, and under the assumption that it "knows" which WS will start streaming data with each other, what protocol is going to be used, and whether or not data the transferred data are going to be encrypted or not.

Producing/Consuming WS. These two components act as an interface between the controller and the streaming library. They take preferences and options provided by the controller and preform the necessary actions to start their server or client, depending on their role.

Backend application. Although the WS it self may play this role, some applications may not fit in this framework. This is why an abstraction of such an application is made, assuming that it will at least provide a flow of data in the form of bytes (the most primitive data type is used for obvious reasons), or request such a data flow.

Streaming Server/client. The role of these components are fairly simple. The server must only send data received from a backend application to a connected client, and the client must pass the data received to the application. The details about authentication, and the actual transfer, are left to the connection component.

Connection. An effort to abstact a connection is made here so that different protocol implementations may be added in the design. For that reason it is assumed that a connection at least 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.

The steps followed to establish a connection between the two WS is depicted in Figure 4, and it involves the following:

1. The WS controller (workflow) is setting the desired options for the connection and the producing application, by updating the WS's Resource Properties (RP) [22].
2. When the appropriate settings are passed from the WS controller to the template WS, the WS controller sends a message for the initialization process in the producing side to start. This means that the backend application is configured to produce data and the sever connects with the application in order to receive that data. The server also initializes its connection according to the properties set from the WS controller (protocol, port, etc.).
3. The producing WS notifies the controller that the initialization process is completed, and now it is the consumer's turn to be initialized according to the desired properties, again by updating the WS's resource properties.
4. After the consuming side is successfully initialized, the controller is notified that it is ready to start both applications for transmitting and receiving data.
5. A start message is sent to both WS. The stream server accepts the connection from the client, and a mutual authentication takes place.
6. If the authentication succeeds, then the producing application starts passing data to the server, which in its turn uses its connection to send them over to the client. The client is reading the data from its connection and passes them to the consuming application. This flow of data is shown in Figure 4 in the doted line.
7. When the streaming is completed the controller is notified and then it may stop the server and shutdown the application, or go on in starting the process again depending on the needs of the particular e-Science application.

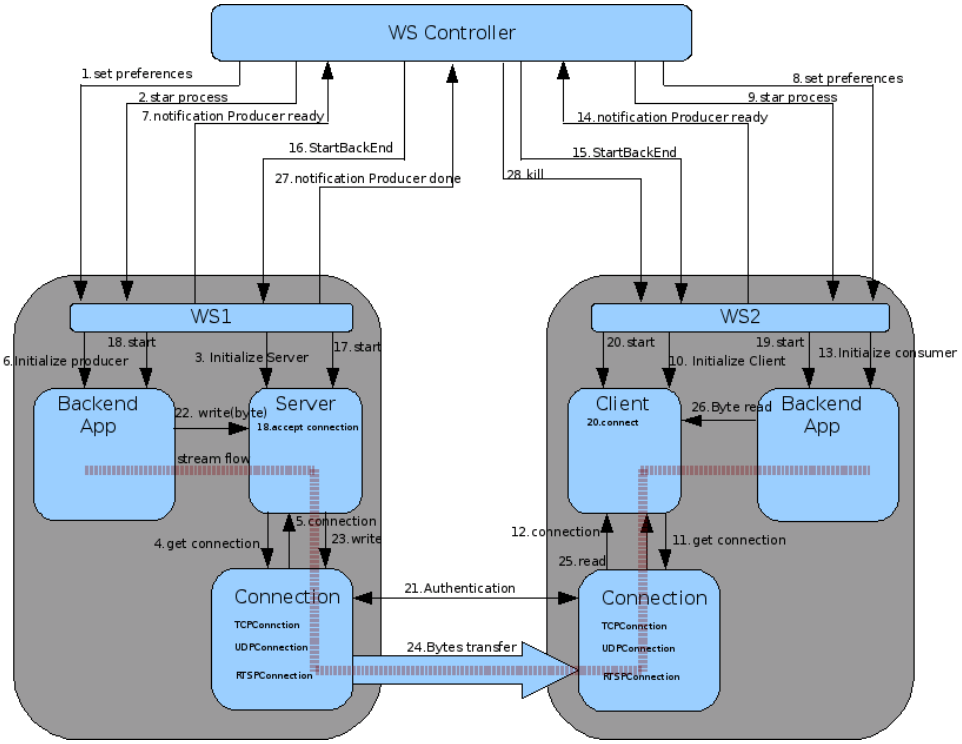

Figure 4. Design of the stream library and the producing/consuming WS. The doted line indicates the flow of data from the producing to the consuming backend application

A more detailed description of the above components is given in the sequence diagram at the appendix. This design for the streaming library was chosen because it offers some benefits in terms of extensibility, as more protocol implementations may be added, and modularity, as the backend application may be separated and from a particular machine and either replaced with one that better suits the needs of the e-Science application, or placed in a specialized resource .Additionally, the server and client components may also be separated and work outside a WS container. On the other hand the template WS is as simile as possible, offering an easy to use API. Thus if a developer wishes to stream data from one WS to an other, all they have to do is to extend, or add the desirable piece of code to the backend application module, while the WS implementation remains the same.

3.3 Implementation Issues

One of the key issues of the implementation, is the authentication mechanism used. Although this process is left to the specific protocol implementations, we have used the same authentication mechanism for the TCP UDP and RTSP protocols. By taking advantage of the interfaces offered by Commodity Grid Kits (CoG) [23] we used the provided GSISocket. This is a TCP socket wrap that provides some very useful tools. Among others it can perform mutual authentication and secure TCP streams. The mutual authentication mechanism is established through the use of a proxy certificate [2]. To authenticate each side, first the proxy certificate is loaded to create a GSI credential instance, then the client connects to the server, and sends its GSI credential instance. This GSI credential is compared to determine whether or not that credential is signed from a trusted Certificate Authority (CA). Once the server has made sure that the credential is singed from a trusted CA, then it generates a random message (token) and send it to the client for encryption. The token is encrypted with the clients private key (taken from its proxy credential) and send back to the server, which decrypts with the clients public key (also taken from the sent credential). If the result is the original token, then the clients identity is confirmed (the client is the one that it claims

[2] A proxy certificate is an instance of a new, temporary public/private key pair, sign with the certificates long-term private key. Proxies normally have a rather short lifetime.

to be). The same process is repeated but with the server sending its credential this time, and the client authenticating the server's identity. Once both entities have authenticated each side, then mutual authentication is established. From now on the public keys of each side may be used for encrypting the channel (at a severe transfer speed cost). Up to this point only the TCP protocol supports data encryption over the communication channel, although the security context (the key offered by the GSI credentials) may be used to wrap and unwrap the data to be sent and received respectively.

Implementing and configuring the TCP protocol to achieve optimal speed, involves setting the receive and transmit buffer size according to this formula:

$$buffer_size = bandwidth * RTT$$

where RTT is the round trip time of the used channel. Although it is argued that by not tuning TCP buffers according to the above formula the bandwidth utilization is less than 5% [6], this has showed not to have an effect on the speed achieved by that protocol. The reason for that is because recent versions of Linux (version 2.6.17 and later) have full buffer sizes autotuning, and thus manual tuning is unlikely to substantially improve the performance of these kernels over most network paths except in some rare cases [8].

Looking at UDP details, one may see that this protocol offers no synchronization mechanisms between client and server as it is a connectionless protocol. As a result of that there is a substantial data loss. This would mean that while tasting, UDP would stream all the data from the server side, while the client would receive less than half of that stream. Although the aim is not to create a new protocol, some degree of reliability was necessary. This called for a simple synchronization mechanism. In order for the server to send packets to the client, it must know on which port the client is listening. The port is retrieved by sending a "test" packet and the by receiving the answer from the client. A timer starts from the moment the packet is sent, and it stops when the replay packet is received. The elapsed time is then used as a time interval for sending packets in a rate the client can receive them.

RTSP doesn't face synchronization problems, as some would expect from a UDP based protocol, because it uses special RTP packets and a TCP control channel that is used to pass commands from the client to the sever. Although it is expected for RTSP to have the best performance, this is not the case, as it is shown in the results section, one possible explanation for this is the "custom" RTP packets created that perhaps take substantial CPU time to create.

In order to incorporate the Styx protocol to our design we chose to use the JStyx library developed primarily by J. Blower [24]. JStyx has most of the Styx protocol implemented, such as file I/O using Java streams and permission checking. However, we found the library, that is still undergoing development, lacking in two key features: security and buffered access.

Since security is critical for every application on the Grid, the fact that there is no provision for this in JStyx was a major concern. To by-pass this, we chose to implement a security mechanism based on GSI authentication on an extra TCP stream before the actual Styx connection would take place. This would also serve to pass Styx-specific information between the server and client. However, the connection through which the data are transferred is not encrypted (although there are provisions for using SSL in later versions) and the mechanism described above (or a similar way of providing authentication/authorization) should be implemented in the JStyx library.

The second shortcoming proved to be more serious than security. JStyx provides an interface to read and write to a file using buffers which would allow it to give a "streaming" effect while accessing files. In essence, parts of the file are assigned a sequence number and are thus named "chunks". By fetching one chunk of the file after another we can stream the file between the client and the server. The methods provided though in JStyx were unable to provide a consistent way of transferring binary files using this method, and even when transferring text files the transferred files tended to have artifacts. We later opted to transfer parts of the file sequentially by splitting it in different Styx files and fetching them one by one. This was achieved using different methods of the JStyx library that work for both binary and text files but return the whole file contents at once.

Although both of our concerns were by-passed in this way, the resulting implementation was way too slow, as even in the preliminary tests on the same computer moving 1 MB of data from one folder to another took approximately 1 minute. Clearly, our attempt to use Styx, a protocol that is designed for file transfer, for a general purpose streaming library has failed, although this failure could be attributed to the poor documentation. Therefore, although Styx can still be utilized from our code for file transfers, we chose not to test it since it has been reported [19] that its performance could reach the performance of TCP over the same channel.

Although there is hardly a need for a different client or server implementation of GridFTP, as the software is both mature and feature-rich, we tried building a small connection library that would provide similar functionality. Of course, building a full-feature server or client was never the goal, but rather exploring the available mechanisms in Java that could be incorporated to our design. We were unable to find a server interface, but it is quite straightforward to request a file from a GridFTP

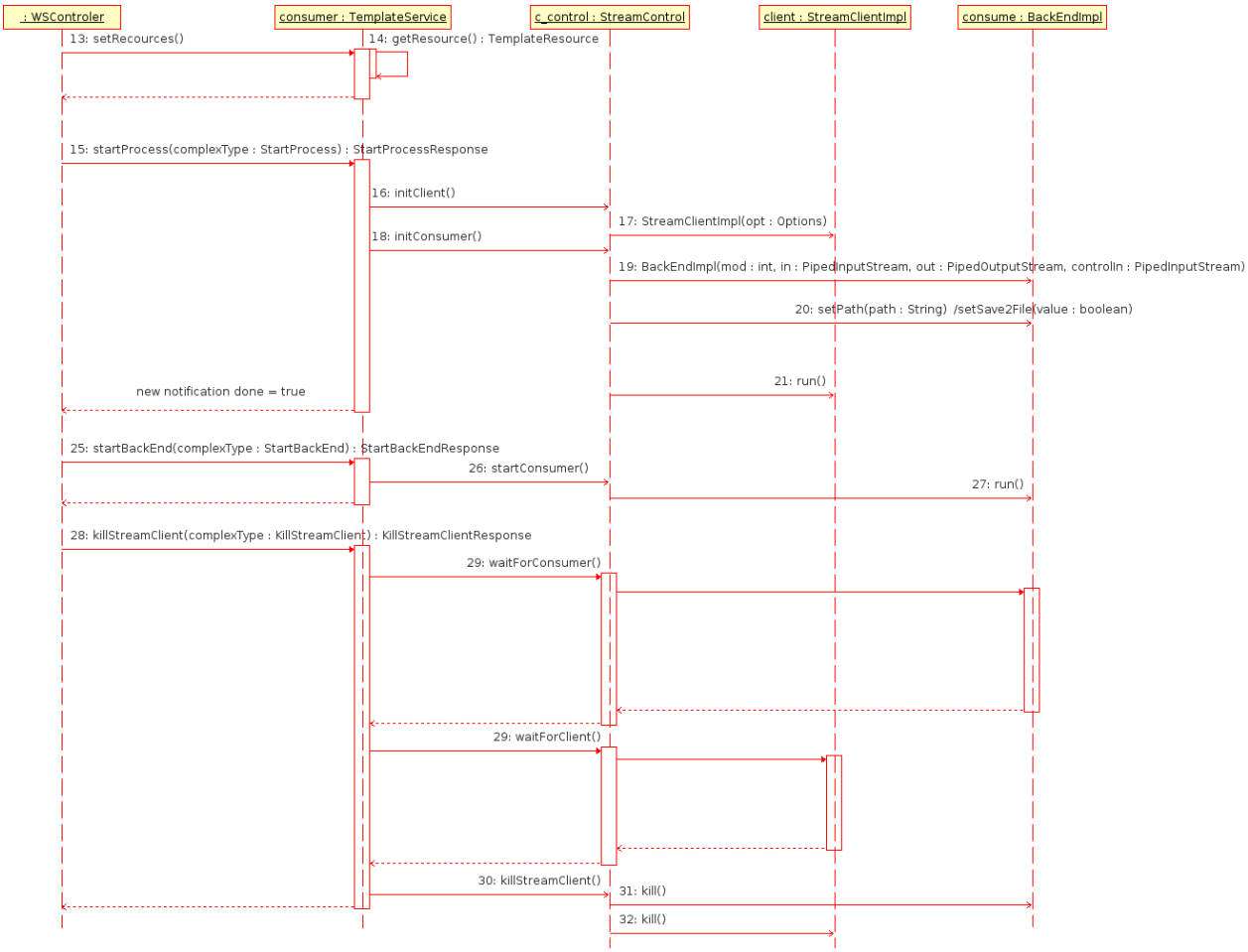

Figure 5. The sequence diagram on the consuming side.

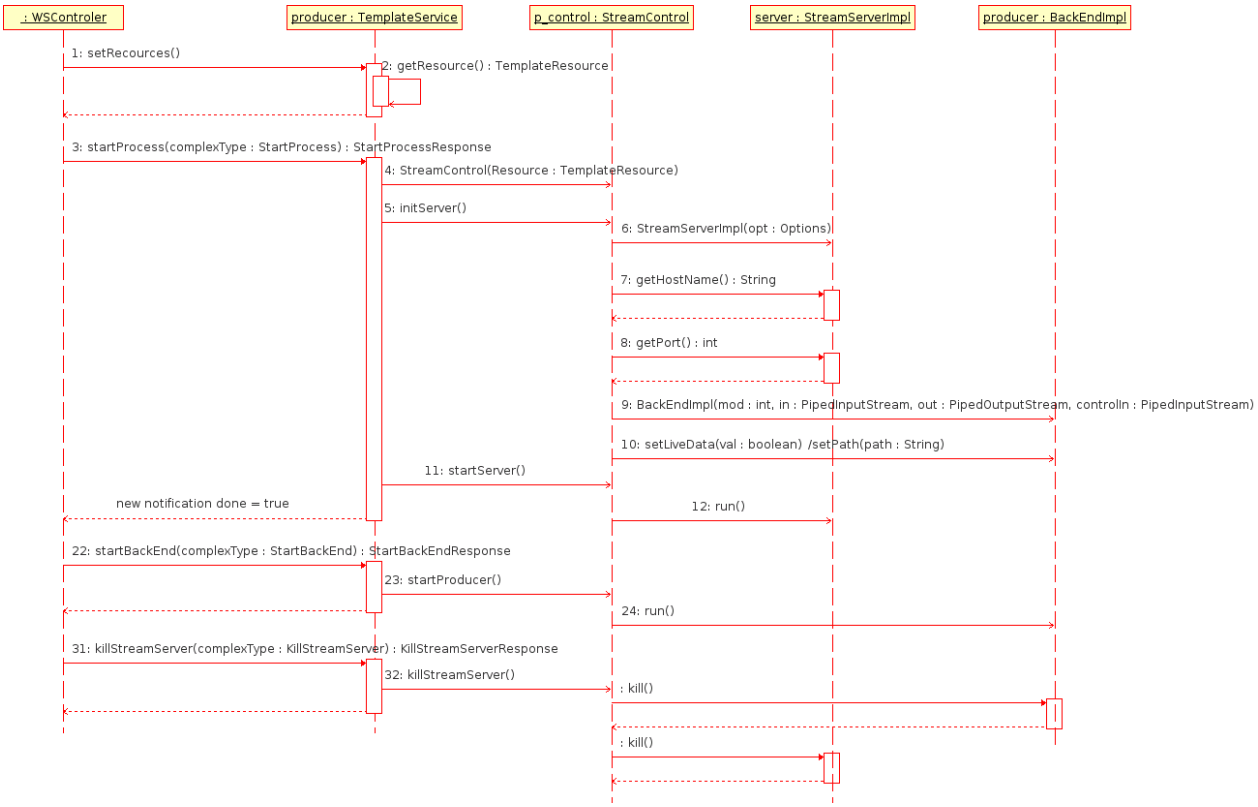

Figure 6. The sequence diagram on the producing side.

server through a special Java stream class that is provided in Java CoG. Thus a connection can be made by using the same interface that all streaming protocols use in our design. This adds to our library functionality, but we have refrained from testing our simple GridFTP client since the results in any case should be comparable to the standard GridFTP client. Thus the results outlined in the Results Section are those obtained using the standard GridFTP server and client.

UDT is a protocol that should be ideal for streaming on the Grid. It is designed for high bandwidth connections with equally high delay, the normal configuration in Grid environments that usually span geographically distant institutions that use connections of several GB/s. However, since it is implemented in C, it was impossible to use the default driver provided in our design, although it should be easy to implement it in a C version of our design as its API is sufficiently close to the standard sockets API and there is even an XIO driver that can be used to load UDT on the standard Globus stack. We have considered re-implementing basic parts of it in Java, but since its power rests upon its complex congestion control algorithm it would have been pointless. For this, UDT was left out of our design, although it must be emphasized that it is a very interesting protocol for streaming on the Grid whose potential should be examined.

4. Testing Methodologies

In order to assess the suitability of each protocol thorough tests have been performed on the DAS3 Distributed Supercomputer [25]. The DAS3 consists of 272 dual AMD Opteron compute nodes, spread out over five clusters, located at the four universities, UvA, TU Delft, Leiden University and Vrije University. Measurements were done using both available interfaces, namely Ethernet and Myrinet, which produced similar results.

Since our main goal was to measure the suitability of each protocol as to the specific network interconnect, an attempt was made to keep the tests as neutral as possible with regards to the underlying hardware. In that sense, we are not measuring disk-to-disk performance but rather "interface-to-interface" since the data are fetched from /dev/zero and end up in /dev/null. Therefore, the added delay of disk I/O is negligible. Finally minimal operations are performed on the stream's data, to minimize delay from the CPU other than necessary for the streaming functions.

The tests comprised of:

1. Node to node test on the same location (namely on the UvA cluster, from node232 to fs2)
2. Direct transfer tests between the DAS fileservers fs2-fs0 (UvA - VU, with a RTT of about 1 ms) and fs3-fs1 (TU Delft - Leiden, RTT $\simeq$ 3 ms [3]).
3. An example workflow application, in which a producer WS kicks-off a transfer from fs3 (TU Delft) through an application in fs2 (UvA) that manipulates the data and immediately streams them to fs1 (Leiden University) where they are stored.[4]

In addition to the implemented protocols in our WS (as described in Section 2), GridFTP and FDT transfer times were also examined, but only in the test cases where direct transfers between 2 nodes were taking place. GridFTP is the standard way of transferring data between Grid applications at the moment (by usually invoking it through RFT) and thus an alternative transfer scheme must at least match its speed. FDT [13] is in essence a fast parallel TCP implementation. It must be noted however that GridFTP and FDT tests are quite out of scope, since they don't support streaming of data from WS to WS. Still, their results can be used for performance comparisons over streaming, and that is why we have decided to include them in our measurements.

All of the tests were performed through a WS-Core [5] container, set up with user privileges in the target sites. In that respect, they are realistic as to the performance penalty imposed by streaming through an active WS container, that could be serving a multitude of different requests.

The most directly perceivable metric in data transfer is of course the transfer speed, that is the amount of transferred data over the unit of time. It is also the most interesting metric in this case, since Grid applications typically are dealing with high bandwidth networks and require fast and reliable transfer of terabytes of data. Thus, a better utilization of these pipes, a larger transfer speed, will significantly affect the performance of complex and intensive Grid workflows.

Transfer speed was measured in two ways. First, directly through the backend application, by measuring the transferred/received bytes over the traversed time, and second through the use of an external bandwidth monitoring application.

[3] All the RTT were measured by ping

[4] The third test case was meant to represent an actual use case scenario, but should be taken only as indicative since certain performance aspects (disk, CPU etc.) have been negated as described above.

[5] The WS-Core is the basic java based container offered by globus.

TCP	115 MB/s
UDP	115 MB/s
GridFTP	115 MB/s
FDT	115 MB/s
RTSP	5 MB/s

Table 1. Transfer speed measured in local cluster.

TCP	45 MB/s
UDP	80 MB/s
GridFTP	75 MB/s
FDT	78 MB/s
RTSP	1.5 MB/s

Table 2. Remote transfer speed UvA-VU.

The reasons for taking measures from two separate sources are first of all that although taken them directly from the backend application proved to be the most suitable method to visualize the results, it didn't quite give a convenient real-time representation of the data transfer. Additionally two more protocols that are not integrated in the streaming library needed to be measured for reference, and despite the fact that both of them provided an estimate for their transfer speed, we opted for the external bandwidth monitoring application for verification: bmon.

bmon [26] is a bandwidth monitoring application, that provides live data on the transfer speed over the interfaces available on a workstation. When in doubt about the speeds observed (making sure that our way of measuring was no different that what was happening in reality), we have used it to verify application measurements and to monitor the overall transfer process.

5. Results

As mentioned in section 4, results were achieved for 3 test cases: i) a local 1 Gb/s Ethernet cluster interconnect between 2 nodes in the local UvA cluster, ii)a remote interconnect from TU Delft to LU and iii) from UvA to VU and a 2 step connection from TU Delft to LU through UvA. The first two test cases were performed for all protocols outlined in 4, whilst the final example of a Grid workflow was tested only for TCP, UDP and RTSP.

Node to node transfer In this case, the transfer speed was measured from fs2, the UvA DAS3 head node, to node232 in the same cluster. The average speed in each case can be seen in Table 1. The results are to be expected since we are measuring over a clear channel with low RTT, so all protocols achieve maximum transfer rate. The only surprise is RTSP, since it basically uses a UDP channel for transfer of more elaborate packets, so it can be attributed to implementation specifics.

Some remarks should also be noted in this case on the way each protocol behaves. UDP achieves the max speed a bit faster than TCP, even with similar buffer size. GridFTP under any configuration achieves a suboptimal yet satisfactory performance (this may be verified by Tables 1-3), which could be attributed to specific reliability checks, before trusting the channel. Probably due to this also, FDT is a bit faster than GridFTP in fully utilizing the channel's bandwidth.

Remote transfer, medium RTT Medium RTT means in this case $\simeq$ 1 ms, which is the RTT for the UvA - VU file servers channel (fs2 - fs0). This is where we start seeing the problems of TCP, since it barely averages half the total bandwidth, as seen in Table 2. UDP gets the full bandwidth, while FDT's implementation of parallel TCP streams seems to be a bit faster (0.2 MB/s) than the mechanism used in GridFTP. RTSP is also unreasonably slow in this test.

Remote transfer, large RTT Typically, Grid workflows spanning multiple institutions have to deal with large RTT's, sometimes in the order of 100's of milliseconds when in different continents. However, in our case "large" RTT means $\simeq$ 3 ms, which is the RTT between the 2 more distant file servers on the DAS 3, namely the TU Delft - LU file servers (fs3 - fs1). The results obtained can be seen in Table 3, and a comparison of the stability of the connection between TCP and RTSP

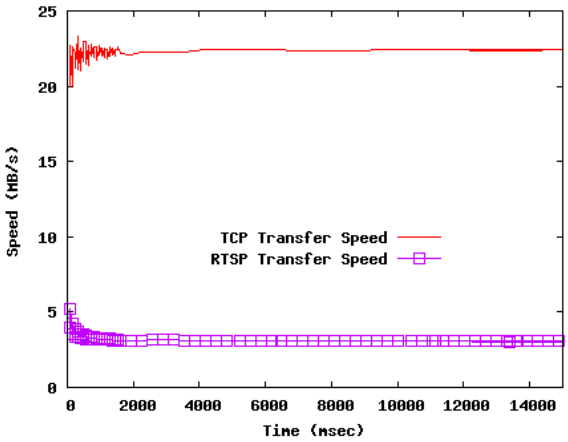

Figure 7. TCP and RTSP speed.

TCP	22.5 MB/s
UDP	32 MB/s
GridFTP	11.5 MB/s
FDT	11.7 MB/s
RTSP	1.5 MB/s

Table 3. Remote transfer speed TU Delft - LU.

in Figure 7. We must also note that the UDP transfer speed is a bit unreliable, as it is not achieved uniformly, and therefore UDP does not appear in Figure 7. It also comes as a surprise that GridFTP and FDT both achieve sub-TCP performance, which could indicate an overestimation of TCP speed, although it is quite improbable since the application measurements were verified using bmon, as detailed in Section 4.

Example scenario: 3 site transfer A real example of streaming data through 3 different sites will give better insight into the performance of each protocol. We only have to deal with the 3 different streaming protocols in this comparison, that perform as seen in Table 4. In Figure 8 one can more closely examine how fast each protocol achieves its average speed. UDP again is unreliable, yet at least 20 MB/s faster than any other tested protocol, in contrast to RTSP that never exceeds 1.5 MB/s. TCP achieves a steady rate of transfer of about 11 MB/s, which is roughly half of its performance when making the direct transfer.

It is interesting to note that a GridFTP transfer in those same channels (fs3-fs2 and fs2-fs1) achieves in the first transfer a speed of about 90 MB/s and in the second 11.3 MB/s. This will come in handy later on, when we will make a comparison between streaming and the traditional way of moving data with GridFTP.

6. Conclusion

It is easy to see why streaming data between WS is desirable. In most cases, a transfer of data over multiple sites is needed, where in each site data processing occurs. If the WS does not need the whole dataset for its processing but can start with the

TCP	11 MB/s
UDP	32 MB/s
RTSP	1.5 MB/s

Table 4. Three party remote transfer speed TU Delft - UvA - LU.

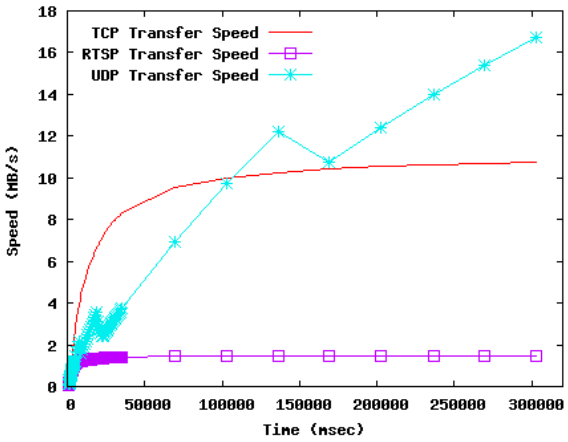

Figure 8. Speed of the total transfer through UvA.

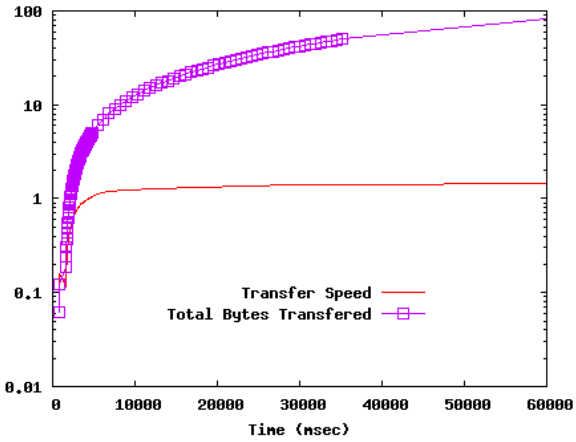

Figure 9. RTSP speed as measured at the UvA.

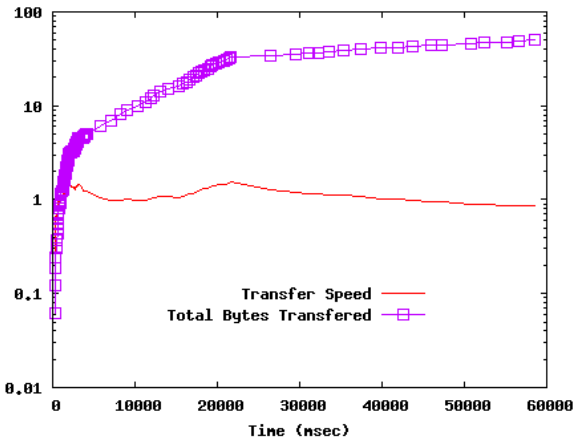

Figure 10. UDP speed as measured at LU.

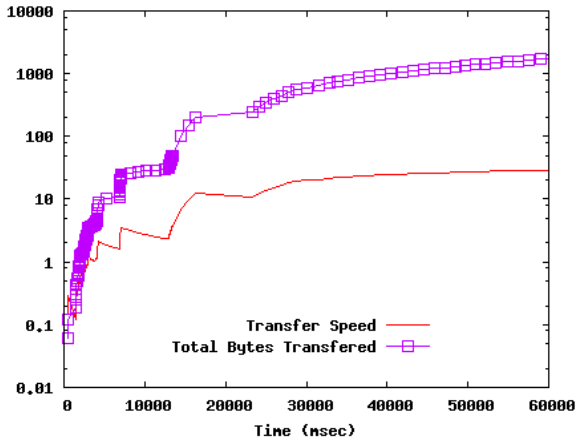

Figure 11. UDP speed as measured at the UvA.

available subset of data from the stream, then the whole process can be sped up significantly.

Let us take a hypothetical example of an application running on DAS3 (Figure 12). A WS on the Leiden University cluster orders data from TU Delft that have to be first processed by an application running on the UvA cluster, before finally storing the results in the LU. Let us now take a reasonable amount of data for such an operation, something like 1GB, and assume that the application can operate on live data, as in our test. We obviously have 2 choices to perform this operation:

1. Transfer the 1GB chunk of data from TU to UvA using GridFTP/RFT, perform the operation and then transfer the 1GB of results to LU.

2. Open a stream through UvA, which includes the application that manipulates the data as they become available on the stream.

Using the rates of transfer measured (5), a 1GB file transfer using GridFTP would take

$$\frac{1000MB}{90MB/s} + \frac{1000MB}{11.3MB/s} \simeq 99.5s$$

The same transfer if executed over a TCP stream would take

$$\frac{1000MB}{11MB/s} \simeq 90s$$

Just by removing the need to "stop" data at the processing site we get a speedup of about 10 seconds!

If we also take into account the need for at least 2GB of storage space at the processing site, which have to be reserved before hand, we can eliminate a part of the Grid administrator's problems by familiarizing the users with streaming. This is not an unreasonable consideration, since for example on the DAS3 cluster every user gets immediately access to Teraflops of processing power, yet the default quota is under 1GB. Therefore, although CPU power is in abundance, storage capacity is disproportionally scarce and streaming can alleviate this problem.

It is clear to us that streaming is a solution to the problem of coordinating data transfer over a grid workflow. It is however quite difficult to get the kind of guarantee that everything will go well. In short, there is no small number of things that can go wrong. Even a relatively simple design like the one described in section 3 can be quite cumbersome to implement.

Many of the pitfalls have to do with putting either too much or too few trust on the underlying protocol implementations. For example, our RTSP implementation places so many constraints on the UDP transfer layer that becomes amazingly slow as seen from Section 4. On the other hand, the UDP implementation is extremely fast yet unreliable, as we have entrusted too many things on the protocol's designer. One can spot proof of sheer "UDP madness" in Figures 11 and 10 and compare it with the sanity, at a cost, of RTSP in Figure 9.

Having tested the TCP protocol with encryption over the communicating channel, tests revealed a significant loss of speed. While unencrypted TCP achieved speeds of $110MB/s$ the encrypted version in the same configuration and channel achieved

Appendices

A Full WS Sequence Diagram

1. The WS controller setts the RP of the producing WS (protocol, port,etc.). Updating RP of a WS is described in [22]
2. The producing WS creates a Template resource (an XML file) according to the WS factory pattern
3. The WS controller invokes the startProcess method that acts as a initialization frontend for all the modules involved (server, backend application etc.).
4. The producing WS created a new StreamControl object, that is there to provide to the WS a uniform access to the streaming library components. It takes as an argument the TemplateResource in order to be able to initialize the components of the streaming library
5. As soon as the producing WS was the StreamControl object it calls the initServer method
6. This creates a new Options object, a component that holds all the data types necessary for initializing the server client etc. Although this could be the TemplateResource, it would stop the streaming library from being used outside a WS framework. Next a new StreamServerImpl is created with Options as arguments.
7. The StreamControl component is asking for the hostname for the server. This is necessary for the consuming client, in order to know where the server is.
8. The some as above is happening here but with the port.
9. At this point the StreamControl component creates a new BackEndImpl instance, with the use of the initProducer method. The BackEndImpl instance takes as arguments the mode (producing/consuming), the data piped streams and a control stream, that enables the application to synchronize with the server.
10. The StreamControl set to the BackEndImpl the path of the file to be streamed, or setts it in a live data producing mode. This means that data produced in the produce method of the BackEndImpl will be passed to the server, and then on to the client, that will deliver it to the consumer application.
11. The StreamControl start the StreamServerImpl instance.
12. The StreamServerImpl goes in a blocking mode waiting for connections.
13. Science the WS controller receives a notification from the producing WS that everything is ready, it goes on updating the RP of the consuming WS.
14. The consuming WS gets the RP again according to the WS factory pattern[22]
15. Now the WS controller calls the startProcess of the consuming WS.
16. The consuming WS uses its own StreamControl to initialize the client.
17. The StreamControl creates a new StreamClientImpl that takes a component Options as an argument.
18. Next the StreamControl initializes the consuming BackEndImpl.
19. A new BackEndImpl. instance is created
20. The StreamControl setts to the BackEndImpl the path were to save the consumed data.
21. The StreamControl starts the StreamClientImpl that will connect with the StreamServerImpl.
22. After the WS controller receives the notification from the consuming WS the it is ready, it invokes the startBackEnd method of the consuming WS.

23. Now the StreamControl starts the producing BackEndImpl.
24. The producing BackEndImpl starts generating data, either form a file, or by its produce method.
25. At this point it is the tern of the consuming BackEndImpl to start.
26. The StreamControl starts the consuming BackEndImpl.
27. The BackEndImpl. starts and receives any data passed from the StreamClientImpl.
28. The WS controller invokes the killStreamClient method provided from the consuming WS.This method blocks until the streaming is done.
29. In order for the killStreamClient to block the StreamControl is waiting until the consuming BackEndImpl terminates.
30. The same happens with the StreamClientImpl.
31. Finally when the streaming is complete, that is when the producing BackEndImpl sends a termination signal to the server, the client is terminated, releasing any piped streams with the server and the application. From this point the server may terminate, or stream to an other client.

Figure 13. The full sequence diagram.

B How to use the Streaming Library with the Template WS

B.1 Overview

The ability to stream data between web-services is vital for the implementation of complex workflows on the Grid. This streaming library implements various protocols, taking into consideration issues such as security and speed. This streaming library provides a simple API through which Java applications can stream data. Also, this architecture can be reused by programmers who want to stream data without having to deal with protocol or web service issues.

B.2 Prerequisites

1. java 1.6 or higher
2. globus toolkit 4.0.4 or higher
3. The streaming library with the Template WS

B.3 The Template WS

The template WS provides a way to manipulate a backend application, and the streaming server. The WS should implement the following methods:

```
public StartProcessResponse startProcess(StartProcess complexType);
```

This method is intended to initialize the backend application and the stream server. Ideally it dose the flowing:

1. Creates a new stream server, passes the necessary arguments to it.
2. Starts this server thread.
3. creates a new backend application instance, together with its arguments (the backend will be examined in a next section)
4. set the ready resource property (RP) to true. Clients calling this WS may subscribe for notifications on this RP.

Note all the arguments are coming from the resource. Also for convenience it would be good for a control class to be crated so that specific initializations of the server and the backend application can be handled form there. In this implementation that "control" class is the StreamControl

```
public StartBackEndResponse startBackEnd(StartBackEnd complexType);
```

The role of this method is quite simple. It starts the backend application. This means either that the producing application stats generating data, or that the consuming application starts receiving data. So it takes the following steps:

1. sets the RP done to false (indicating that the application is not done yet)
2. starts the application thread

```
public KillStreamServerResponse killStreamServer(KillStreamServer complexType);
```

Having this method gives the control to force the server to shutdown. Ideally it should call the kill method the stream server provides. There should be some mechanism to make sure that the method doesn't kill an already dead server. Additionally science the server is no longer there the application could be terminated as well. This is not a constrain, but assuming that the application's (producing) job is to provide data to the other end, it has no longer a purpose.

```
public KillStreamClientResponse killStreamClient(KillStreamClient complexType);
```

This method is here to kill the client, and possibly the application. Unless a forced termination is interned, this method should block until the consumer is done. In this case the method's job is:

1. wait for the consumer to terminate
2. wait for the stream client to terminate
3. kill the stream client by calling its kill method. (this is not so necessary, but it makes sure that the certificate and the stream objects connecting it with the application are released.

As a frontend the Template WS should make sure that Constants class is initialized properly (e.g. pass the RP maxbuffersizeRP to the Constants.setMaxBufferSize(**int** maxbuffersize)).

B.4 The Template Resource

The TemplateResource is there to provide the arguments to the streaming server, streaming client, and backend application. So the RP are:

`flielocationsRP` . A string array (unbounded) that can be used by the backend application. If the backend application is in a producing mode and it is meant to stream a local file (possibly after some process), this RP specifies the file locations. If the backend application is in a consuming mode it specifies the (local) path were the file is going to be saved. Only one value is set in this case, as the consumer can't distinguish between 2 separate stream files, so while using multiple files, it should be made sure that the client side is teared down, and start receiving the next file with a fresh client.

`requestfileRP` . A string the is used only in the case of GridFTP or Styx. Science there is no server implementation for these two protocols, this path is used to request from the server (GridFTP,Styx) for a specific file.

`portRP` . This RP is an integer, that is used by both the stream server and the client. In the stream server case it sets the port that the server will be listening to (assuming that the underling protocol has this property), while in the stream client case it informs it were the server is listening to (ideally after the stream server is waiting for connections, the port should be verified that it is the same as the one returned to the stream client).

`modeRP` . An integer specifying the backend application's mode producer/consumer. The class Constants provides the two modes as a static final integer. It is better to use that science some important checks are made based on that constant.

`protocolRP` . An integer specifying the protocol to be used for the streaming. Use the Constants class to set/get the protocol from/to the resource, as the library relays on that to return the appropriate protocol implementation.

`hostnameRP` . A string specifying the string servers location. This is returned to the streaming client after possibly verified from the stream server.

`readyRP` . A boolean used for coordination purposes. It notifies the WS client (if registered to that RP) that the initializations are done (stream server/client, backend application).

`doneRP` . A boolean used for coordination purposes. Notifies the WS client (if registered to that RP) that the backend application are done.

`encryptdataRP` . This boolean is used in order to encrypt the streaming channel (option passed to the protocol thorough the stream server)

`save2FileRP` . A boolean specifying to the consuming backend application whether or not it should save the data it receives from the stream.

`livedataRP` . A boolean specifying to the producing backend application whether or not it should produce data from its produce method, or a file should be streamed.

`maxbuffersizeRP` . An integer, setting the buffer size to this limit. This integer affects the backend application by setting the produced byte array size to that limit and the stream server/client by sting the socket's buffer size. This is done to make sure that there is a consistency throughout the classes.

B.5 The Backend Application

The aim of this class is to provide an abstraction for a backend application. It implements a Runnable interface, so that it may run on parallel with the stream server or client. The BackEnd class has two roles: producing and consuming. Starting with the consuming role ,extending this class, one should implement the following methods:

```
public abstract void consume(byte[] data);
```

This method is there to provide data to the backend application. This method is called until the stream client has no more data to offer. The stream client notifies the application that there are no more data with the string BACKEND_EOD. This happened with by the conumeBytes() method.

When in a producing role extending the BackEnd class requires the implementation of:

```
public abstract byte[] produce();
```

Intuitively this method provide data to the stream server so the may be passed to the consuming application. This method should return **null** when the are no more data to stream.

```
public abstract byte[] consumeAndProduce(byte[] data);
```

This method takes data provided from the client, dose some process on them and passes them to the server.

The following methods don't need to be implemented, they just provide an easy way to handle different scenarios while producing, or consuming.

```
protected void streamBytes() throws IOException
```

This method calls the produce() method, until it returns **null**, when it dose, it throws an IO exception, that should be caught be the the run method. When this happens, the run method calls endOfTrasminiton();

```
protected void streamFile(String path) throws IOException
```

Although this API doesn't aim to stream files, it provides this method for continence, and testing purposes. It simply opens a stream to a local file specified by the path argument, and writes the bytes read to the server.

```
protected void conumeBytes()
```

This method reads data from the client and passes them to the consume method. If the client returns the BACKEND_EOD string, then a wile loop breaks

```
protected void conumeAndPushBytes()
```

Assuming that a backend application is in the middle, that is it will consume data, process them, and pass them on to the next application, then it would use this method. A while loop reads data from the client, passes it to the consumeAndProduce method. Data returned from that method are written to the server.

A class that extends the BackEnd class is the BackEndImpl class. Apart from implementing the classes methods above, as a Runnable it implements a run method.

```
public void run()
```

This method runs until a kill is given to the application. Depending on the mode given at the constructor, it adopts a producing, consuming, or a middle role.

```
private void producingRole() throws IOException
```

if the RP livedataRP of the template WS has being set to true, then the application calls the streamBytes() method resulting in pushing to the server what ever the produce method returns. In the case where the livedataRP is set to false, this means that the application will stream a file. This is done by calling the streamFile(path) method

```
private void consumingRole() throws IOException
```

Similarly this method depends on the save2FileRP RP of the WS. If set to true, the consuming application will save the streamed data to file, otherwise it will just run the conumeBytes method.

```
private void midleManRole()
```

This method just calls the conumeAndPushBytes method.

B.6 Developing and Testing a streaming enabled WS in 11 steps

At this point, an example will be given on how to develop an application that will use the Template WS as a frontend, and the streaming library, so that data may be streamed from one WS to the next. More specifically this application will have three parts. i) A producing backend will generate a byte array that will represent pixels. ii) An intermediate backend, will receive that byte array and place a random byte, in every even byte of that array, and pass it on to the next backend. iii)The final backend will take the processed data and place an other 100 random bytes to the byte array and save it as a jpg format in the local disk. To archive this the following steps are taken.

1. In the folder nl/uva/science/grid/data/stream/impl/application, create the class that will do the job. Say the class's name is PhotoApplication. This class extends the BackEndImpl. The constructor should provide the flowing arguments, needed by the parent class BackEndImpl:**int** mode,PipedInputStream in, PipedOutputStream out,PipedInputStream controlIn. So the constructor looks like this:

```
public PhotoApplication(int mode,PipedInputStream in,PipedOutputStream out,
        PipedInputStream controlIn){
        super(mode, in, out, controlIn);
}
```

   - Why these arguments? As mentioned above they are needed by the parent class BackEndImpl, but they will not be used in this class, as the parent class takes care of the data transfers and synchronization.
   - What do they do?

     mode . This integer specifies the role that the application will assume. Later on when using this constructor, the argument may be either Constants.CONSUME, or Constants.PRODUCE, or Constants.MIDLEMAN

     in It is where the application will get its data from. This is a PipedInputStream that connects the stream client with the consuming application

     out . When producing the application will use this PipedOutputStream to send data to the stream server. controlIn. This enables the stream server to synchronize with the producing application.

2. Next we need to override the produce method provided by the BackEndImpl class. Doing so the data returned from this method will go to the server for streaming. So we create a simple random byte array resulting to the method looking like this:

```
public byte[] produce(){
        counter++;
        if(counter>=10){
                counter = 0;
                return null;
        }
        pixel = new byte[1024];
        random.nextBytes(pixel);
        return pixel;
}
```

   - why the counter and the **return null**? The BackEndImpl class will keep calling this method until it gets null from it. So if we are planing to stop it eventually we need to return **null**. In this case 10 kB are enough. That's it the producing part is ready.

3. Now we need to process the byte array produced. This is done by overriding the consumeAndProduce method. This step is quite straightforward. Just take the data from the methods argument, process them, and return them. This method would look like this:

```
public byte[] consumeAndProduce(byte[] data){
        for(int i=0;i<data.length;i++){
                if( (data[i]%2)==0 ){
                        random.nextBytes(processByte);
                        data[i] = processByte[0];
                }
        }
        return data;
}
```

The incoming data are received from the streaming client processed and passed to the streaming server.

4. Finally we need to consume the byte array and save the final product. Overriding the consume method will do just that, with the addition of a save method. So the consume method is :

```
public void consume(byte[] data) {
        processByte = new byte[100];
        random.nextBytes(processByte);
        for(int i=0;i<processByte.length;i++){
                data[i] = processByte[i];
        }
        image = Toolkit.getDefaultToolkit().createImage(new MemoryImageSource(300, 300, co
        saveImage();
}
```

while the saveImage(); is created specifically for saving jpg fomats, so that would be:

```
private void saveImage(){
        int w = image.getWidth(null);
        int h = image.getHeight(null);
        BufferedImage bi = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
        Graphics2D g2 = bi.createGraphics();
        g2.drawImage(image, 0, 0, null);
        g2.dispose();
        try{
            ImageIO.write(bi, "jpg", new File(path+"/testImage.jpg"));

        }catch(IOException ex){
            ex.printStackTrace();
        }
}
```

5. Now that the backend application is ready, it's time to put it in the template WS. In the nl/uva/science/grid/data/stream/impl/, there is the StreamControl class. This class is there to provide an easy manipulation and initialization of the backend application and the stream server, and client. So at this point we need to create the producing instance of the PhotoApplication. Locating the initProducer mathod in the StreamControl class we create the instance. So the method is:

```
public void initProducer(){
        app = new PhotoApplication(Constants.PRODUCE, pinc, poutc, pinsControl);
        app.setLiveData(true);
        threadProducer = new Thread(app);
        threadProducer.setName("Producer");
}
```

- Why the live data is set to true? Since this application will produce a byte array from the produce method, we need to specify that to the parent class, so that will call the produce() method and not the streamFile(String path) method. Alternatively this could be set from the RP. For example: app.setLiveData(resource.getLivedata());. Doing so we need to set the RP from the WS's client.
- What are the inc, poutc, pinsControl and where did they come from? When looking the constructor of the StreamControl class we see that these piped streams are created to connect the stream server and client with the application without them there is no communication between them.
- What is this threadProducer? Since we will run the server and the application in two separate threads, it is necessary to start the application as a thread.

6. Now it's time to crate the processing instance of the PhotoApplication. This is done in the initMidleMan method, of the StreamControl class. To do so we write the flowing of code in that method:

```
public void initMidleMan(){
        app = new PhotoApplication(Constants.MIDLEMAN, pinc, poutc, pinsControl);
        app.setSave2File(false);
        threadMidleMan = new Thread(app);
        threadMidleMan.setName("MidleMan");
}
```

   As we see from this method it's the same thing as before, with the difference that the mode is set to Constants.MIDLEMAN, and that the Save2File is set to **false**. This means that the application will not save the incoming data to a file but rather it will pass them on to the next backend. This attribute may also be set from the WS's RP, by calling app.setSave2File(resource.getSave2File());

7. The same process will be flowed in setting the consumer instance of the PhotoApplication. Locating the initConsumer() method in the StreamControl class. Adding this code to the method would crate the appropriate instance:

```
public void initConsumer(){
        app = new PhotoApplication(Constants.CONSUME, pinc, poutc, pinsControl);
        app.setSve2File(false);
        app.setPath(resource.getFlielocations(0));
        threadConsumer = new Thread(app);
        threadConsumer.setName("Consumer");
}
```

   - why the setSave2File is set to false, while the application saves data in the disk? Although the parent class provides a method that saves data to a file, in this case we crated an other method suitable for saving bytes in the form of jpeg
   - What is the app.setPath(resource.getFlielocations(0))? In order for the application to know where the final product will be saved, it needs to get it from the WS's RP and in extension from the WS's client.

   That's it! The PhotoApplication is ready to be compiled and deployed to the container.

8. Locating the globus-build-service.sh script in the top folder, in a console run [6]

```
./globus-build-service.sh template
```

9. Now just deploy the service by running

```
globus-deploy-gar nl_uva_science_grid_data_stream.gar
```

   The service should be deployed in the container

[6] this script is taken from the tutorial found in http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html

10. If we want to test the service locally we open three terminals and with separate containers on different ports. This is done be calling:

```
globus-start-container -nosec -p 8082
globus-start-container -nosec -p 8081
globus-start-container -nosec -p 8080
```

The ports should be these exactly, as the WS's client is set to call the WS on these ports.

11. To start the PhotoApplication we use the StreamControler class, by ruining:

```
java -DGLOBUS_LOCATION=$GLOBUS_LOCATION \
-cp $CLASSPATH:build/classes/:build/stubs/classes/ \
nl.uva.science.grid.data.stream.control.StreamControler \
-example -local 192.168.1.10
```

This should result in creating an image in /tmp directory.

References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in highperformance computational grid environments," 2001. [Online]. Available: citeseer.ist.psu.edu/article/allcock01data.html

[2] 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*.

[3] J. G. F. Hernandez, P. Bangalore, and K. Reilly, "A graphical modeling environment for the generation of workows for the globus toolkit," 2004.

[4] G. von Laszewski, K. Amin, M. Hategan, N. Zaluzec, S. Hampton, and A. Rossi, "Gridant: A client-controllable grid workflow system," 2004. [Online]. Available: citeseer.ist.psu.edu/vonlaszewski04gridant.html

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

[6] B. Tierney, "Tcp tuning guide for distributed applications on wide area networks," *http://www.didc.lbl.gov/papers/usenix-login.pdf*, 2001.

[7] M. K. Gardner, S. Thulasidasan, and W. C. Feng, "User-space auto-tuning for tcp ow control in computational grids," *Computer Communications*, vol. 27, no. 14, p. 13641374, 2004.

[8] Advanced Networking Pittsburgh Supercomputing Center. Enabling high performance data transfers.

[9] B. Tierney, "Tcp tuning guide for distributed applications on wide area networks," pp. 33–39, 2001. [Online]. Available: http://www-didc.lbl.gov/papers/usenix-login.pdf

[10] M. K. Gardner, S. Thulasidasan, and W. chun Feng, "User-space auto-tuning for tcp flow control in computational grids." *Computer Communications*, vol. 27, no. 14, pp. 1364–1374, 2004.

[11] Y. Gu and R. L. Grossman, "Udt: A transport protocol for data intensive applications," *Internet Engineering Task Force Internet Draft, draft-gg-udt-01.txt*, August 2004.

[12] Y. Gu and R. Grossman, "Using udp for reliable data transfer over high bandwidth-delay product networks," manuscript submitted to Computer Communication Review.

[13] Fast data transfer website. [Online]. Available: http://monalisa.cern.ch/FDT/

[14] Real. (2000) Realsystem iq whitepaper: Rtsp interoperability with realsystem server 8. [Online]. Available: http://www.real.com

[15] D. Abramson and J. Kommineni, "A flexible io scheme for grid workflows." in *IPDPS*. IEEE Computer Society, 2004. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2004-c.html#AbramsonK04

[16] D. Abramson and J. Komineni, "Interprocess communication in griddles: Grid enabling legacy software," School of Computer Science and Software Engineering, Monash University, Tech. Rep., 2003.

[17] J. Kommineni, D. Abramson, and J. Tan, "Communication over a secured heterogeneous grid with the griddles runtime environment," in *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2006, p. 21.

[18] R. Pike and D. M. Ritchie, "The styx architecture for distributed systems," *http://www.vitanuova.com/inferno/papers/styx.html*, 1999.

[19] J. Blower, K. Haines, and E. Llewellin, "Data streaming, workow and rewall-friendly grid services with styx," *http://www.resc.rdg.ac.uk/jstyx/sgs/*, 2005.

[20] R. Kettimuthu, W. E. Allcock, L. Liming, J.-P. Navarro, and I. T. Foster, "Gridcopy: Moving data fast on the grid." in *IPDPS*. IEEE, 2007, pp. 1–6. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2007.html#KettimuthuALNF07

[21] Humphrey, Wasson, Jackson, Boverhof, Rodriguez, Gawor, F. Bester, Lang, Meder, Pickles, and M. Keown, "State and events for web services: A comparison of five ws-resource framework and ws-notification implementations," pp. 24–27, July 2005.

[22] B. Sotomayor. (2005) The globus toolkit 4 programmer's tutorial. [Online]. Available: http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html

[23] (2007) Cog jglobus. [Online]. Available: http://dev.globus.org/wiki/CoG_jglobus

[24] (2007) Jstyx website. Reading e-Science Center. [Online]. Available: http://www.resc.rdg.ac.uk/jstyx/

[25] The das 3 website. [Online]. Available: http://www.cs.vu.nl/das3/

[26] bmon website. [Online]. Available: http://people.suug.ch/ tgr/bmon/