Scalable Data Transfers for Web Services

Spyridon Koulouzis

Faculty of Science

Universiteit van Amsterdam

A thesis submitted for the degree of

Master of Science (MSc) in Grid Computing

2009 September

Abstract

The work presented in this thesis address the problems web services face when called to transfer large data sets within the e-Science context. e-Science as a distributed research paradigm requires the sharing and co-ordination of resources scattered around the globe. Web services as an implementation of SOA offer the desired abstraction, interoperability, and flexibility that will allow for the realization of these requirements.

Web services however face problems related to data transports, because SOAP does not offer a scalable and fast way of transporting large data sets, and/or because web services are used in orchestration workflows. Under this type of workflow, SOAP messages passed to the service consumer before delivered to the appropriate service. Furthermore, in cases where web services are used in data-centric workflows, and are developed with data transport capabilities in mind, the most common approach is to provide these services with the means of accessing remote locations (GridFTP, etc.). Although this feature is desirable, it overlooks the fact that many temporary data are first moved to a remote location before delivered to consuming services. This practice may slow down workflow executions, and put unnecessary stress on data resources. Additionally there is an abundance of deployed web services restricted in using only SOAP while service consumers (clients, workflows etc.) are stretching these services to produce and deliver large data sets. This on one hand, results in non-scalable e-Science applications, while on the other hand if these services are to be re-implemented, it will probably make them unusable for exiting workflow implementations. Another reason why these services cannot be re-implemented is simply the fact that the source code is no longer available. Therefore, a significant challenge is to find reliable and efficient methods to transfer large data between web services.

To address these problems we introduce the ProxyWS, a data aware web service that is not only able to *coordinate* the transportation of large data volumes, but also provides support for large data transfers to existing web services. The ProxyWS is implemented as a service itself, so that existing service oriented applications can still use deployed services, with the added functionality of large data transports. Another feature that the ProxyWS is introducing, is direct data streaming between web services, a feature that can speed up workflow execution, by creating a data pipeline between services participating in that workflow.

Thus the goal of the ProxyWS is to provide scalable data transports for both "legacy web services" and new web service implementations alike. In doing so large scale SOA-based applications are promoted, which from the backbone of e-Science applications.

Acknowledgements

I would like to acknowledge the help support and encouragement of many people during the time of completion of this thesis project, as well as the time of my study in the University of Amsterdam. I thank Dr. Adam S.Z. Belloum, my supervisor, for his valuable guidance during the completion of this work, as well his support throughout my study period in the Grid Computing master program. His support also encouraged me in developing a critical mind and a research interest in Gird Computing and e-Science. My co-supervisor Dr. M. Scott Marshall, for his helpful input in identifying the problems and requirements of real-life SOA applications. I would also like to thank all of my teachers in the University of Amsterdam, for providing me with a stimulating study environment that expand my knowledge in the field of computer science. For their help in coping with technical details, providing valuable advice, and use cases presented in this work, I thank Piter de Boer, Edgar Maij, Elena Zudilova-Seinstra, Vladimir Korkhov, Dimitry Vasunin, and Marco Ross. I also would like to thank my family for their support and understanding all these years, I have being abroad. Finally yet importantly, I thank all my friends and people close to me in Amsterdam. Without them Amsterdam would be much different.

Thank you all very much.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

Scientific research, has often being the driving force behind the introduction of new technologies, while it can also being said, that new technologies, have opened new paths for scientific research. An example of this cycle is the introduction of the web, where scientists used it to prompt the collaboration between them, but as scientific research becomes more demanding in terms of collaboration, knowledge, and resource sharing, it has pushed for the introduction and usage of new technologies and models. e-Science is the next step that attempts to support the requirements of scientific research. e-Science depends on Service Oriented Architecture (SOA), and web services as an underling model that will realized goals of large scale coordination and collaboration, through resource and knowledge sharing. For services however to be able to meet these demands, they have to first address some issues, one of which is reliable and scalable data transfers, as this new way of scientific research is producing and analyzing large quantities of data. It is therefore desirable to introduce a mechanism that will be able to address problems of data transports, for e-Science and SOA-applications.

1.1 e-Science

e-Science is the systematic science of collaborative research, employing computational-intensive, data-intensive, and highly distributed methods. It is the evolution of the scientific research, where virtual laboratories provide access to resources around the globe, and knowledge sharing is made possible in a far more distributed, interactive, and massive way. Thus e-Science is the effort of combining, and sharing resources including

computing, network, data, human, etc. so that a research collaborative environment is provided, or in other words (1):
"*e-Science is about global collaboration in key areas of science and the next generation of infrastructure that will enable it.*"

The need for a more efficient, and collaborative environment comes from the challenges presented in modern science research, which has to face problems that are growing in complexity. Moreover, research objectives are not isolated, but are rather cross-disciplinary and large-scaled, causing an overwhelming increase in data complexity and volume, something that makes the need for communication, collaboration and coordination among scientists more apparent than ever. Currently the amount of resources that could tackle these problems is available, but in order for researchers to use them efficiently, a sharing and coordination model is needed. These resources include high-speed networks, an abundance of computational power through HPC (High Performance Computing), Grid, and Cloud Computing, and a large scale of scientific data. Figure 1.1 shows the relationship between resources, applications, and the e-Science environment.

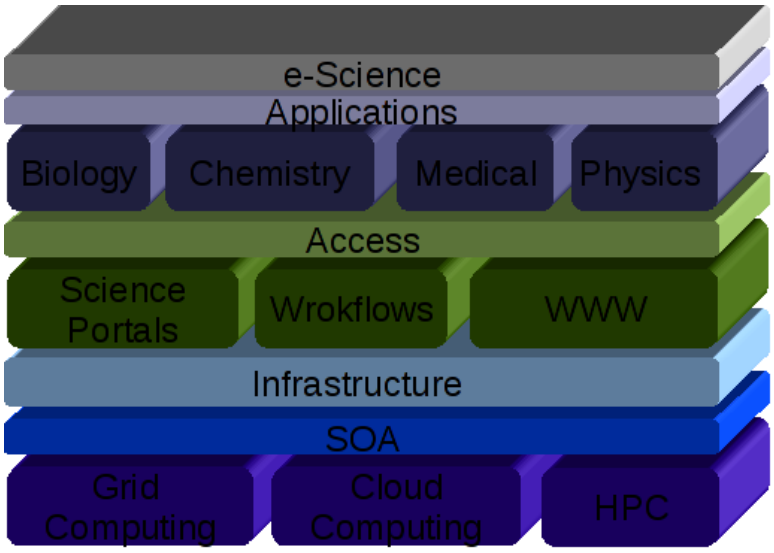

Figure 1.1: e-Science - A layered structure of e-Science and the way applications, and resources are related.

e-Science has emerged as a solution to challenges like cross disciplinary research, coordination and cooperation, while its ultimate goal is to provide scientists with a

more easy-to-use interface and to help them get access of distributed and heterogeneous resources.

1.1.1 Infrastructure

As mentioned in Section 1.1, e-Science is trying to combine and share resources amongst scientists. Although e-Science is about resource sharing, this notion, at least in the context of e-Science describes practices, and frameworks. As seen from Figure 1.1, e-Science uses some form of computing models to deliver a cooperative scientific environment. Very often these models, either depend, or implement Service Orientated Architectures, in order to deliver specific resources to applications. In this Section we will give a brief description of the most common technologies used to-date, in order to provide computational, and storage resources.

Grid Computing One of the most known applications of Grid computing is found in the LHC (2), where the four experiments (ALICE, ATLAS, CMS, LHCb (3; 4; 5; 6)) will produce approximately 15 petabytes of data annually. Apart from the data potentially produced, each of the experiments before the actual collisions, has being performing simulations of collisions with the use of various frameworks and tools (7; 8; 9). The frameworks that will support these experiments are based on the Grid computing paradigm.

Grid computing is the combination and sharing use of hundreds, or thousands, of geographically and organizationally disperse and diverse resources (hardware or software) in order to solve problems that require large computational power (10; 11). The goal of Grid computing, or the "promise", is to be able to provide unlimited computational power to its users regardless of location, or platform used. This "promise" gave this model its name which is compared with the power grid, where users are able to plug in to a socket any kind of device, and imitatively receive power.

One of the key elements of Grid computing is the Virtual Organization (VO), a structure that enables the coordination and sharing of the available resources, whether these are computers, software, or data. VOs are formed by individuals, organizations, institutes, or any other entity that is contributing and/or using resources (11). For example in the Large Hadron Collider (LHC) context each of the four experiments mentioned previously also makes up a VO, which spans many institutes around the

world. Regardless of setting up a VO, Grid Computing still needs to address a number of important issues, some of which include:

- Portability, Interoperability, and Adaptability. In order to make full use of available resources, Grid applications should be able to use heterogeneous resources, across different platforms, and configurations (12).

- Resource Discovery. Since Grid computing is comprised by large numbers of resources scattered across different geographic locations and organizations, resources should be easily and transparently discovered (13).

- Fault Tolerance. The Grid, as a large collection of distributed resources owned by different organizations, is not able to guarantee the stability of individual resources; instead, various mechanisms have to provide alternative resources in order to cope with errors and resource failures.

- Security. The use of different resources, owned by different organizations raises complicated security issues, which span from authorization, and authentication, to data integrity and privacy, on large scale multi-user environments (14; 15).

- Monitoring. Monitoring covers almost every aspect that makes up this model. Among others, it concerns resource monitoring, job execution and error reporting, as well as security and access logs.

Grid Computing is facilitated by grid middlewere, which is a collection software component that provides access to, distributed resources. In recent years grid middlewere is shifting towards Service Oriented Architectures, in order to realize the grid "promise". Under this architecture, services are used for providing resources virtualization, information, etc. Figure 1.2, depicts a simplified view of the Grid, where a hierarchical information service holds information about available resources. End-users and job submission services consume this service for discovering resources (also available as services). End-users can submit and monitor jobs through the job submission service and the notifications it receives from resource services.

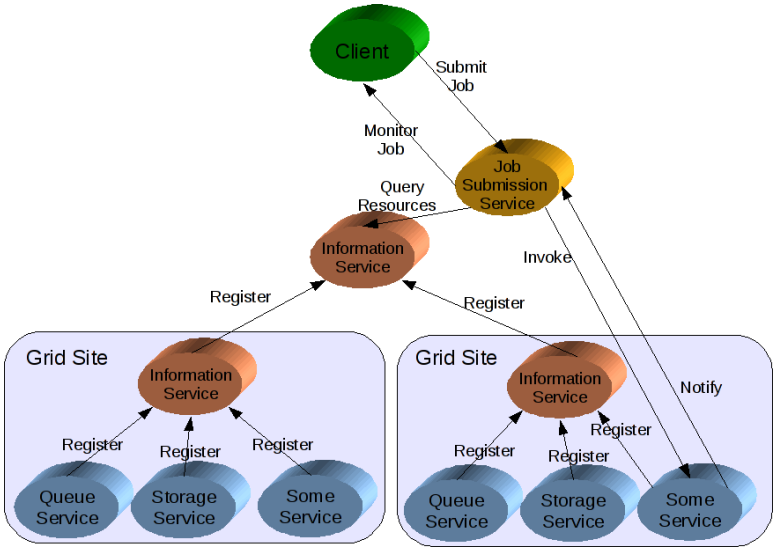

Figure 1.2: The Grid - A simplified view of the Grid. Each grid site registers its resources to a hierarchical information service used by job submission service and end users.

Cloud Computing Cloud computing is emerging as a new paradigm that promises to deliver infinite resources, under an easy access framework, that hides the complexity of the system from the user. This paradigm shifts the location of resources to the network, something that gave this paradigm its name since often the network is depicted as a cloud in design diagrams.

As a new paradigm cloud computing has no commonly accepted definition, although attempts to do so are made (16). In many cases Cloud computing seems to share many features with Grid computing, since both attempt to utilize an abundance of computational and storage resources. Perhaps one of the main differences that separate the two paradigms is their origin and the notion of resource sharing. Cloud computing was initially introduced for commercial usage, while Grid computing was introduced for scientific research. Additionally cloud computing is not providing any form of resource sharing. This does not stop Cloud computing from entering the scientific research domain (17; 18). One of the benefits of using such a model, could be easing the steep learning curve Grid computing seems to have. Apart from easy usage Cloud computing promises "elastic" scaling of resources, by being able to add and remove them depending on requirements, a feature mainly accomplished thought vitalization (19).

The majority of cloud computing infrastructure, consists of services interfacing

different resources hosted in the provider's infrastructure. This allows the Cloud to appear as a single access point for all the computational needs of users. So Cloud computing adopts an "as a service" approach, for offering resources to its users. As such, Cloud computing currently offers the flowing resources as a service:

Infrastructure as a Service (IaaS) . This gives users the ability to request, control and release instances of virtual machines, running on the providers infrastructure (cluster, data centers, etc.). The most popular example of IaaS is Amazon Elastic Computing Cloud (19), where users may request or upload instances of virtual machines.

Platform as a Service (PaaS) . It is the delivery of an application-hosting platform that enables deployment of web applications in the providers infrastructure. An example of PaaS is offered by Google and its Application Engine which enables deployment of scalable web applications (20).

Software as a Service (SaaS) . Is about delivering applications to users, through the web. SaaS software vendors deliver on-demand application releasing users from the need of updating, or purchasing applications. An example of a SaaS is Google documents (21)

As it seems Cloud computing is not clearly defined probably because it is introduced by vendors, each one providing its own implementations. Nevertheless Cloud computing has some features that characterize it. These include:

- Delivery of non-shared resources.
- Aggregation of resources, possibly heterogeneous.
- virtualization of hardware and software platforms.
- Security through isolation of users.
- Scalability of resources (nodes, sites, hardware, software, etc.).
- Centralized control by providers.
- User friendliness.

- Lack of standards.

Although in most definitions the utility aspect of Cloud computing is emphasized, it is felt that it should not characterize this model. Billing for resources or services, is something that is entirely up to providers. Instead, accounting is an aspect that can include billing of usage, without isolating this paradigm to a commercial domain. Despite the lack of an accepted definition, cloud computing is taking advantage of the Service Oriented Architecture in order to deliver scalable, "elastic" and easy to used resources.

1.2 Service Oriented Architecture

In recent years as networks are becoming faster, safer, and more reliable, there is a clear trend for developing distributed applications that depend on standardization for their successful deployment, usage and maintenance (22). Consequently, SOA (Service Oriented Architecture) was emerged as an effort to provide a standardized description for developing loosely coupled distributed applications. Thus, SOA refers to a style of building reliable distributed systems that deliver functionality as services, with the additional emphasis on loose coupling between interacting services. Furthermore SOA refers to the design of a system, not to its implementation, and as an architectural style it emphasizes the description of components as modular services that can be discovered and used by clients (23). Furthermore, clients can use these services individually, or they can integrate them in order to provide higher-level services. This approach promotes re-use of existing functionality and dynamic compilation of distributed applications. In order to do that, services communicate with their clients by exchanging messages, which are usually defined in terms of requests and responses.

As mentioned previously services can be loosely coupled in order to provide a higher-level service. This can be accomplish by organizing them in a workflow, where the order in which individual services are used is controlled by the workflow definition (24). Typically, service providers register their services in service registries where they advertise their capabilities and usage in order for service consumers to use and/or organized them in workflows. Clients can use this registry in order to discover services that meet their demands. Figure 1.3, illustrates this interaction.

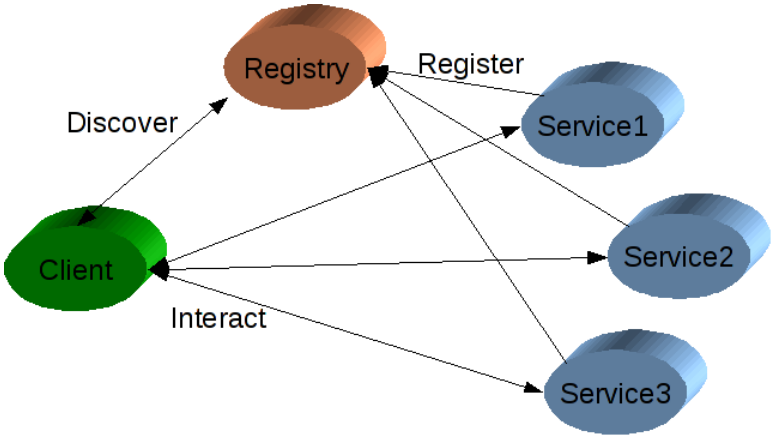

Figure 1.3: SOA - An example of SOA

As mentioned earlier SOA describes loosely coupled services, which in essence means that services participating in a workflow or a SOA application are unaware of each others existence. This provides some benefits namely:

- Flexibility: A service may be deployed in any machine and relocated if necessary. As long as that service updates its registry entry, clients will be able to discover it.
- Scalability: Services can be added and removed in order to meet the demands of the application.
- Replaceability: If services maintain a standard interface, new or updated implementations, can be introduced, without disrupting the functionality of any workflow or application.
- Fault tolerance: If a service becomes unavailable for any other reason, clients can still query the registry for alternate services that offer the required functionality.
- Reusability: A service can be part of more than one workflow, as it can interact with multiple clients.

1.3 Web Services

Web Services are a distributed computing technology, based on Service Oriented Architecture principles that make up the building blocks for developing loosely coupled

distributed applications. W3C defines them as: *"a software system designed to support interoperable machine-to-machine interaction over a network. It (a web service) has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."* (25). Thus, web services interact by exchanging SOAP messages while WSDL provides the description of their functionality.

The Simple Object Access Protocol (SOAP) is an XML-based protocol for exchanging messages over decentralized, distributed environments. Web services can used it over a variety of protocols although the most common transport protocol is HTTP(S). SOAP forms the basic messaging framework upon which web services may communicate, through several different types of messaging patterns. A SOAP message shown in Figure 1.4 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. This message represents remote procedure calls and responses.(26; 27).

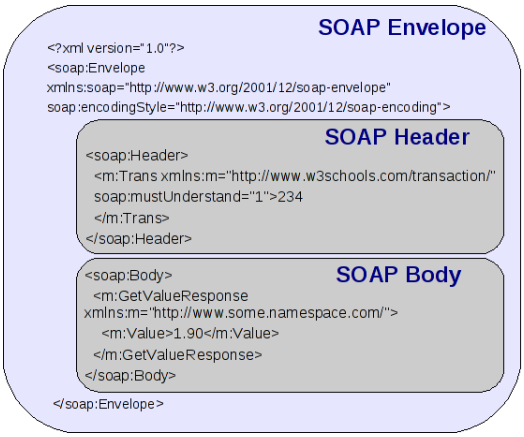

Figure 1.4: SOAP message - An example of a SOAP message, and its three

In order to construct a SOAP message that will invoke the desired web service, a description of the service is needed. This is accomplished through the Web Service Description Language (WSDL), which describes the public interface of a web service. In general, WSDL describes a web service into two levels: an abstract, and a concrete. At

an abstract level, a WSDL document describes a web service in terms of the messages it sends and receives, and the data types used in these messages. The description of these data types found in WSDL, are typically provided by XML Schema (28). Additionally at the abstract level, WSDL also defines the services ports, or a collection of supported operations. At the concrete level, the WSDL defines the protocol and data format specifications for particular port types.

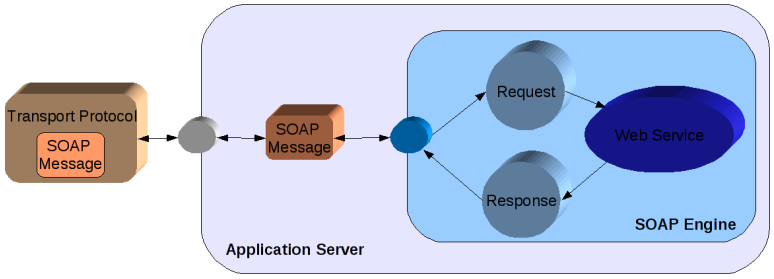

Figure 1.5: Deployed Web service interaction - Typical web services are deployed within a application server

Web services are usually deployed within application servers that use some communication protocol for sending and receiving responses and request. Figure 1.5 shows the architecture of a web service implementation under an application server. On the top level, the application server is responsible for sending and receiving messages with the use of the underling protocol. These message are delivered to the appropriate application, where in the case of web services is an SOAP engine (a SOAP implementation according to the specifications proposed by W3C (26)). The SOAP engine is then responsible for instantiating requests from the SOAP message, and invoking the desired web service. Therefore, when a client wants to invoke a web service, it connects to an application server that provides the WSDL describing the service. The client can then determine which methods are available by the web service. WSDL also describes any special data types in the form of XML Schema. The client can then generate a SOAP message to invoke one of the methods. When a SOAP message arrives at the application server, it is first delivered to the SOAP engine which transforms network level SOAP messages into something more tangible for applications to deal with (such as domain-specific objects. Typically, the service generates some response that is then fed back to the SOAP engine. The SOAP engine creates a SOAP message and passes it to the application server, for delivery back to the client.

The mechanisms described here enable web services to operate in a language independent manner. More importantly the when encapsulating the internal implementation within the service, and providing a standard interface of its methods and data types to consumers, the owners of a service are free to evolve its internal structure over time (for example improve performance, dependability etc.), without making changes to the message exchange patterns used by existing web service clients. This encourages loose coupling between consumers and service providers, which is important for interoperability and language and platform independence, something that provides the basses for large scale distributed applications.

1.4 Scientific Workflows

Researchers can create scientific applications and complex experiments with workflows. Workflows combine and coordinate a set of services so that a more complex goal is met. In other words, a workflow brings together services in a consistent manner in order to provide a description of execution of a higher-level application. 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" (29). Scientific workflows can be broken down in three different steps: design, specification, execution and monitoring (30).

Workflow design determines how workflow components interact. A workflow is composed by connecting multiple tasks according to their dependencies, while the workflow structure indicates the relationship between tasks. In general, a workflow can be represented as a Directed Acyclic Graph (DAG) or a non-DAG. In DAG-based workflows, three structures can be identified: i) sequence, where one task starts after a previous task has completed ii) parallelism, in which tasks are executed in parallel and iii)choice, where a task is selected to execute at run-time if its conditions are satisfied. Non-DAG workflows have the same structures as DAG-based, with the addition of an iteration structure, in which tasks in a workflow are iterated according to some control or conditions.

Workflow Specification defines a workflow including the definition of tasks. There are two types of workflow models namely: abstract and concrete. In the abstract model, a workflow is described in an abstract form, in which the tasks of the workflow are specified without referring to specific resources. On the other hand, the concrete model binds tasks in a workflow to specific resources. Usually the concrete workflow model is generated just before or during workflow execution, according to the status of resources at execution time.

Workflow execution and Monitoring , takes place when the workflow is defined. At this stage, the workflow instance is executed on specified resources. If these resources provide status information, the workflow can monitor the execution of each task.(31).

In the case of web services two approaches exist in workflow implementation: Service Orchestration and Service Choreography. Service Orchestration (shown in Figure 1.6) 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 (32; 33). 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 1.6) is more collaborative, because it describes the message, exchange among interacting — yet independent — web services (34).

Regardless which architecture is chosen, any data driven workflow execution can be effectively reduced into three stages: (i) generating or obtaining data, (ii) processing that data, and (iii) transferring or storing the results. Usually individual services perform these three stages. Thus, it is apparent that data transport in SOA and e-Science applications play an important role.

1.5 Motivation and Goal

SOA has emerged as an appealing paradigm for developing applications by providing interoperability and flexibility in large scale distributed environments which make up the e-Science infrastructure (i.e. clusters, computational grids, clouds, etc)(35; 36). In order to take advantage of these features, e-Science applications tend to adopt this paradigm and more specifically implementing SOA-based applications by developing and using web services that seem to gain acceptance, not only in the scientific domain

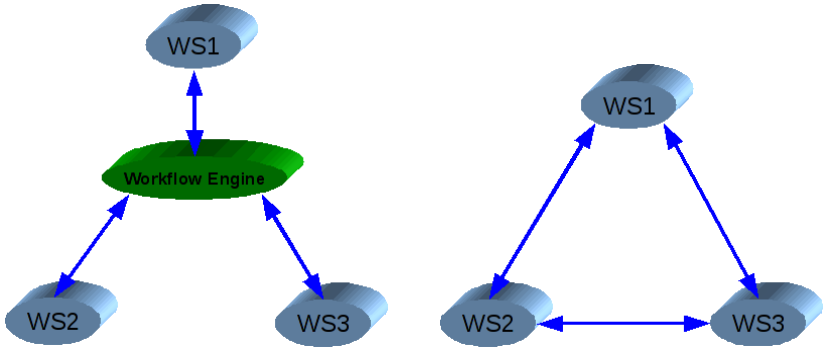

Figure 1.6: Service Orchestration and Choreography. (Left) Service Orchestration — web service calls are always controlled by a workflow engine and (right) Service Choreography — which describes the message exchange among interacting web services.

but also in the commercial(37; 38; 39; 40). Another reason why web services are used in e-Science applications, is the fact that they can be easily coupled in workflows, which forms the bases of the design and execution of scientific experiments (41). Moreover scientific experiments such as medical, and meteorological (39; 40), are data driven, something that shows that Scientific applications require analyses of distributed data from a variety of resources managed in diverse regional, national and global repositories (42; 43; 44).

Although Web services provide a SOA implementation they suffer from "data isolation", since SOAP does not offer a scalable and fast way of transporting large data sets to and from web services (45; 46). Additionally there is an abundance of deployed web services restricted in using only SOAP while service consumers (clients, workflows etc.) are stretching these services to produce and deliver large data sets. This on one hand, results in non-scalable distributed applications, while on the other hand if these services are to be re-implemented, it will probably make them unusable for exiting workflow implementations. Another reason why these services cannot be re-implemented is simply the fact that the source code is no longer available. Furthermore, in orchestration workflows, any call to a data 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.

Under such environments a challenge that arises is how to enable the scientists to harvest on-line data for executing data driven workflows that are made up from

services. To address these problems we introduce the ProxyWS, a web service that is not only able to *coordinate* the transportation of large data volumes, but also provides support for large data transfers to existing web services. The ProxyWS is implemented as a service itself, so that existing service oriented applications can still use deployed services, with the added functionality of large data transports. Therefore, if an existing client wishes to use the functionality of both the ProxyWS and an existing service, the only thing it needs to change is the address of the SOAP call, from the existing service, to the ProxyWS. In doing so, the ProxyWS will make sure that the invocation will be directed to the appropriate service, and data will not be returned via SOAP. Another feature that the ProxyWS is introducing, is direct data streaming between web services, a feature that can speed up workflow execution, by creating a data pipeline between services participating in that workflow (47; 48).

Thus the goal of the ProxyWS is to provide scalable data transports for both "legacy web services" and new web service implementations alike. In doing so large scale SOA-based applications are promoted, which from the backbone of e-Science applications.

1.6 Related Work

Many different approaches have being introduced, in an attempt to address the "data transport problem" faced by web services. One of them is proposed by the Styx Grid Services (SGS) (49), a service implementation based on the Styx protocol (50) initially developed for the Plan 9 (51) distributed OS. In this work, the authors identify the benefits of SOA, as well as the drawbacks of web services when called to transport data. The motivation behind the SGS, is to expose legacy command-line programs as web services, while using a direct streaming approach for data exchange between SGSs. An additional feature of SGS, is the fact that they can operate as a web service or within the Web Service Resource Framework(WSRF) (52), something that means it may send and receive SOAP messages, and expose the service functionality thought WSDL. Although this work is very interesting, it is isolated in using the Styx protocol, and does not provide services with alternative data resources access.

Taverna (53), a well-known workflow management system, faces one of the most common problems known to workflow orchestration. That is large data are returned inside SOAP messages to Taverna, which then has to redirect them to the appropriate

service. Thus, large messages may potentially crash Taverna since it has to cope with both control and data flow. To overcome this problem a Data Proxy web service (54) has been introduced. This service wraps existing web services, and SOAP calls are directed there instead of the target service. When the target service generates a response the Data Proxy web service, references that response, and delivers back an HTTP URL. Although this approach solves the problem of passing large data to consuming services (without having to go through Taverna), it is not addressing the case where data need to be stored or obtained from a remote location (i.e. GridFTP). Additionally if a "legacy web service" is called to produce data that can't be fitted in SOAP, this approach will no prevent that service from crashing because the host run out of memory trying to generate the SOAP massage.

A similar work is presented in (55) where its authors are addressing issues of orchestrating data-centric workflows. Their approach is a hybrid workflow architecture that is based on centralized control flow (orchestration), and distributed data flow. The key component in realizing such a hybrid approach is a set of proxy web services, deployed in the vicinity of the web services participating in a data centric workflow. According to the authors, a centralized control flow has some advantages over the choreography model, mainly in fault detection and error handling. Although this work addresses the issues of orchestrating data-centric workflows, the proxy services exchange data via SOAP. Moreover, services cannot access remote data resources for retrieving or storing data.

Flex-SwA (56) is an implementation that extends the SOAP with Attachments (SwA) specification (57). This work extends the abilities of web services in terms of data transports while it preserves the interoperability of the existing SwA specification. More specifically Flex-SwA attempts to increase the flexibility of bulk data transmissions in service-oriented environments, as it allows the use of different protocols for data transmission including TCP, GridFTP, RFT, and BitTorrent. In this work, the authors take advantage of diverse protocols and do offer a flexible way of delivering data to web services, but this implementation leaves out legacy web services that are no longer able to cope with the data they need to transmit.

The authors of (47), present a framework to support distributed data streams, for web services. Their approach relies on a messaging infrastructure (NaradaBrokering) that adopts a publish/subscribe paradigm. Additionally in this work, the introduction

of a Web Service Proxy (WSProxy) handles data streams on behalf of services, with the use of NaradaBrokering. Although this work is consternated on data streams, and does not provide access to remote data resources for web services (legacy or new), the use of its WSProxy provides a good solution for handling large data that consuming services need.

From the work described above, it seems that an efficient way of addressing the data transport problem of both "legacy" and new web services is to deliver data directly to services, thus tackling the problem orchestration seems to have as well as removing the need for saving temporary data to remote locations. Additionally to overcome the limitations presented by SOAP, as well as to take advantage of legacy services, we need to create a data proxing mechanism that would be responsible for data transfers. In the Sections that follow the specifics for the design and implementation of such a solution will be presented.

2

Architecture for Scalable Data Transfers for Web Services

The next generation of experiments and applications within the e-Science context is expected to generate a "flood" of scientific data (10). Since web services seem to play an important role in this environment, they would have to be able to cope with this expected data explosion. Therefore an architecture that would provide data transfers for web services, would have to scale with potentially enormous data sizes.

One step towards scaling data transfers for webs services, is to enable current implementations to go beyond what they can currently achieve. The main hurdle of "legacy" web services comes from SOAP itself. So by providing alternative data path for existing web services would enable existing implantations to scale with current data demands. As noted from Section 1.6, one way of providing this alternative data path would be to direct SOAP calls to an other services that can take care of data transfers.

Furthermore to be able to meet the data demands of todays and futures applications, web services will also have to take advantage of existing data solutions. Grid Computing has provided some implementations that promise an abundance of data storage. These solutions mainly involve file storage and management, like LFC, SRM etc. It is therefore apparent that a scalable and usable architecture would have to use existing solutions in order to meet the demands of e-Science applications. These demands, mainly involve access to scientific data located in remote data resources.

But to address the problem of enormous data transfers for web services, we should also focus on how data produced by web services can be efficiently handled. An intuitive

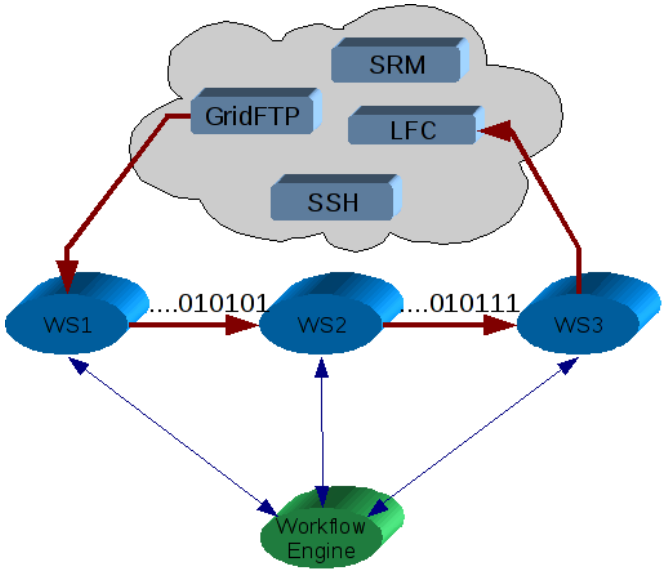

Figure 2.1: Data pipeline for data-centric workflows - In data-centric workflows, creating data streams between web services can provide scalable data transfers.

approach, especially for data-centric workflows, is to deliver data directly to consuming services. If these services are able to process data streams, then streaming can be considered as a scalable solution. A data stream is generally considered as an infinite stream of data, that is channeled in some consuming function or process (**?**). Under this approach, data pipelines are created between web services, as presented in Figure 2.1, and by using the existing data resources, web services can access, process, and store even larger amounts of data, making them a truly scalable solution for e-Science applications.

2.1 Design Requirements

Since e-Science applications and in extension web services operate on diverse data and complex execution environments, it is therefore expected from a data transport mechanism used in these applications, to meet requirements of flexibility, security, reliability, and scalability.

2.1.1 Flexibility

First of all such a mechanism should be adaptive and extensible. As the e-Science infrastructure grows from experimental to mature stable mechanisms, lots of implementations and designs change or even disappear in a short period of time. One day data repositories are located in an SRB server (58), the next they are moved to LFC. Moreover many applications use distributed data sources which might be fragmented and not interoperable. Also datasets may vary in size, type, and format. This fragmentation and data distribution over different resources calls for a mechanism that will provide uniform access to these resources. Furthermore some applications might need specialized mechanisms for accessing specific data resources. If for example a medical application provides data from an MRI scanner, a data transport mechanism for such an applications should be easily extensible to meet these demands. Finally as hardware infrastructures change, new software and services provide access to this sate-of the art infrastructures. For example in the case of StarPlane, a set of services provides access to a high performance phototonic network (59). If a data transport mechanism wishes to take advantage of such infrastructure, it needs to be extensible, and provide a good level of abstraction in order to cope with constantly changing environments.

2.1.2 Security

Another consideration for a data transport mechanism is security. Since e-Science applications grow in sophistication and connectivity their data also becomes vulnerable to tampering. Furthermore some applications, like medical might be using sensitive data that needs to be kept private, or under strict accesses control (60). Security also plays an important role for adopting the e-Science paradigm, because users will need warranties for their data integrity and privacy. For example, although data coming from the LHC will be open to anyone having a suitable Grid certificate, they must keep their integrity in order to ensure sound scientific conclusions. Hence a data transport architecture should consider the following issues:

1. Authentication and Authorization. Authentication refers not only to the identification of users wishing to access a resource, but also to the identification of other resources trying to access one-another. Thus some of the issues authentication mechanisms have to cope with include: i) Interoperability across different systems

and domains, for example when authenticating different resources. ii) Scalability to large communities of users and models of multi-level authentication (14; 15). Authentication mechanisms on the other hand grand access to users or resources based on access control policies.

2.1.3 Reliability and Scalability

When designing a data transport mechanism for e-Science application, one has to ensure that such a mechanism is reliable and can cope with large data sets.

Usually e-Science applications are described and executed through complex workflows which most of the times are characterized by their long execution times. Usually the execution time can be distinguished in two parts: processing and data transport time, since in a typical workflow execution one component processes data produced by another (dataflow paradigm). Thus restarting data transfers due to failures is a luxury such applications cannot afford since they will be incurred with even longer execution times.

2.2 The ProxyWS

The ProxyWS architecture is designed with the requirements mentioned above in mind, as well as the solutions proposed in Section 1.6. The ProxyWS is a web service that focuses on two points: i) Deliver data from remote data resources to web services (existing or new) and ii) Enable these services to deliver larger data-sets. Accessing remote data resources in a uniform way is not mentioned in any of the work presented in Section 1.6, but seems necessary for e-Science applications. On the other hand a number of interesting approaches are presented in the same Section, and have being adopted by the ProxyWS. One of these approached is the notion of proxing SOAP calls, and managing data transport on behalf of web services. Thus by combining the requirements presented in Section 2.1 and related work presented in Section 1.6, the ProxyWS is designed to be a non-intrusive web service that can transport large data sets to and from legacy web services, while taking care that results produced are not returned via SOAP. Additionally by exploiting existing API's the ProxyWS is able to deliver data, from remote data resources (like LFC (61) or GridFTP (62)). This functionality is delivered to existing web services, without having to change the services

them selves, or the underlying environment (application server). The second point is to enable new service implementations to handle and transfer potentially enormous data sets by using direct streaming between web services, thus creating data pipelines. This is achieved by providing a simple API to web services that can either open data streams to or from other web services or by providing a uniform access to remote data resources.

To realize these goals, the ProxyWS is made up from modules, each one playing a different role in the design. These modules are described in the following sections.

2.2.1 Web Service, WSDL interface

The WSDL interface is the front-end of the ProxyWS invoked via SOAP, an interface for making proxy calls to legacy web services while resolving referenced data. Additionally the web service provides the means of getting the transport URIs which the ProxyWS uses to read and write data. This WSDL results from combing the solutions proposed by (47; 55) and (54) (also mentioned in Section 1.6). More specifically the web service provides the following methods:

- `callService`: This method receives as arguments the name of the target service the name of its method and an array containing the necessary arguments. If the call is successful this method will return a URI where the produced data may be obtained.
- `callServiceReturnObject`: Sometimes it isn't necessary to reference data, so this method just returns the result as an object, while taking the same arguments as the `callService`.
- `asyncCallService`: Similar with the `callService` this method makes the proxy call but instead of waiting for the method to complete, it immediately returns the URI back to the caller.
- `asyncCallServiceReturnObject`: When calling this method, the caller will receive a key, which can use to poll the service for the result.
- `getReturnedValue`: When using the `asyncCallServiceReturnObject`, the caller can use this method to poll the service for a result.

- `getReturnedValueRef`: Similar to the `getReturnedValue` this method returns a value generated by a proxy call, but instead of the actual value the caller gets a URI reference.
- `setProxy`: When accessing resources that require authentication and/or authorization a caller must provide the ProxyWS with its proxy credentials.
- `getUloadURI`: Since the goal of the ProxyWS is to enable web services large data transport, it also provides a method for obtaining an upload URI. The ProxyWS may return a number of URI, depending on the implemented protocols, so a client may decides which is more suitable.
- `getResourceURI` method, can reference any resource, local or remote, in any of the available protocols (i.e. a local file or a data stream to an HTTP URL, or a remote GridFTP file to an HTTP URL).

These methods presented here aim in providing clients, the means to invoke existing web services that can now obtain and produce larger datasets, while not having to change their implementations. Furthermore in an effort to keep the WSDL interface simple, while at the same time covering the needs of proxy invocations, the number of methods exposed by the WSDL, is kept low.

2.2.2 Data Transport Context

This component handles the communications and Grid credentials on behalf of the web service. So if the web service has obtained a value from a proxy call, it uses the data transport context to pass that value to the VRSServer or any other data resource. Additionally when streaming is used the web service can obtain input/output streams to any resource, including any VRSServer implementation via the Data Transport Context. So this module can be seen as an effort to provide "data management" services, for the web service, as well as to make a separation between control (legacy service invocations), and data handling.

2.2.3 VRSServer

The VRSServer is a simple interface that enables web services to stream data to other resources. An implementation of that is the HTTPTransport component, which from

the web service's point of view only provides input and output streams to data resources, or other web services. More specifically the HTTPTransport is able to directly stream data produced by the ProxyWS or by a file, local or remote. In general this module enables services to deliver data directly to consuming services, either by direct streaming, or as files. In other words this module makes it possible for the web service, and in extension legacy services, to deliver larger data to other resources (data or services). Hence data pipelines are possible and workflow executions are more efficient, as data don't have to go through the workflow engine or remote storage locations.

2.2.4 Virtual Resource System

The Virtual Resource System (VRS) is a Java API that enables homogeneous access to multiple resources (GridFTP, SRM, data streams, etc) and it is the API used by the VBrowser, an intuitive GUI for browsing Grid resources (63). The API adopts a layered architecture seen in Figure 2.2. According to that, the VRS on the top level defines a resource abstraction which can be seen as a handler for referencing resources like files and services. On the next level the VRS defines a virtual file system, an HTTP, an OGSA and a web service resource. The Virtual file system includes implementations that can handle physical files, like local file systems, SFTP, GridFTP, etc. For abstracting HTTP resources, the VRS assumes that this kind of resources can at least provide input and output streams. This layered implementation adopted by the VRS enables a uniform access to multiple data resources just by providing their URI. As for the service resource implementations, those are mainly used for accessing storage managers and file catalogs. Finally an extension to the VRS is the VRSServer, so that the ProxyWS can provide data streams from proxy calls or web service backend implantations. This part of the implementation is one of the most crucial, as it enables web services to uniformly access scientific data located in a variety of data resources.

2.2.5 Main Component interaction

Having given an overview of the main components that make up the ProxyWS design, we can now describe the way they interact with each other. Figure 2.3 shows the interaction between the main components.

When the web service interacts with a workflow engine or a client, it receives SOAP invocations according to the methods described in Section 2.2.1, while it returns either

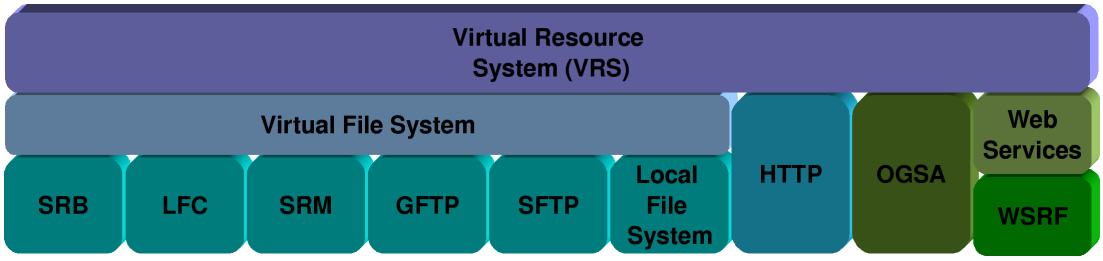

Figure 2.2: VRS Architecture - The VRS architecture is designed in order to abstract any resource

data objects or data references (contained with in SOAP messages). Next, the web service may interact with a target service, where invocations result in data objects that are not yet encased in a SOAP message.

The data context component may receive those data objects as a result of a web service invocation or provide a data object received from the VRSServer to the web service. Furthermore the data context is responsible for providing input and output streams, that are provided by either the VRS, or the VRSServer[1] to the web service.

Next the VRSServer might provide data objects, converted from input data streams or the input streams themselves to the data context. The VRSServer is also serving data requests for resources, whether those are file streams, or data coming from a proxy call.

Finally the VRS is responsible for giving the design access to diverse data resources by providing input, output streams or file handles, from remote locations.

2.3 Design and Implementation Issues

The most common framework for developing web services is Axis (64). It is essentially a SOAP engine, which can be used for constructing SOAP processors such as clients and servers. Nevertheless Axis also includes a simple stand-alone server, which can plug into servlet engines, and Web Service Description Language (WSDL) support. Tomcat on the other hand is a popular Java based web server that supports web applications and servlets. Besides the stand-alone server, Axis may also be deployed under Tomcat.

[1]Currently the VRSServer is not part of the VRS, but it could be integrated so the VRS may also provide stream producing entities.

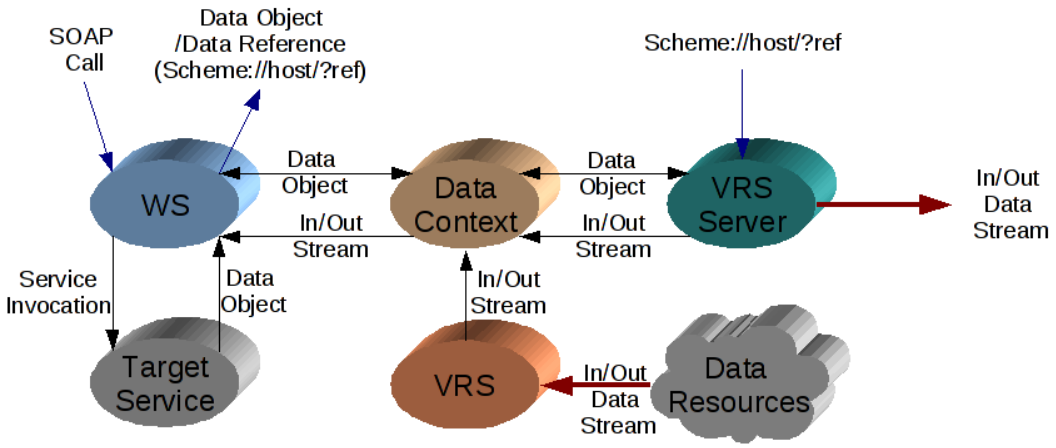

Figure 2.3: Main Component interaction - The web service is responsible for making proxy calls to legacy services, while the data context and the VRSServer take care of data transports

Taking advantage of these tools the ProxyWS implements a VRSServer as an HTTP servlet that runs under Tomcat as its data transport mechanism. This choice was made because developing an HTTP servlet requires minimum effort, while it doesn't put big demands on Tomcat. Additionally Tomcat is optimized for using HTTP, and as such HTTP servlets are fast, light and reliable applications for streaming data via HTTP.

Furthermore the VRSServer is able to serve the flowing request:

- Web Services Streaming. This request calls the VRSServer to copy data coming from the ProxyWS to its output data stream.
- Client Streaming. With the request, an HTTP client may upload data to the VRSServer. This data stream is passed from the data context to the ProxyWS.
- Upload Files. When the client needs to upload files to the ProxyWS, it may use this request. The VRSServer will open an input stream, save the incoming files and pass a code of the transaction to the data context so the ProxyWS is able to obtain the path of the uploaded files.
- Upload arguments. In the case that input arguments are large enough to fit in a SOAP message, the client may use this request for uploading arguments that will be kept in memory, until they are consumed by the ProxyWS through any

proxy calls. Again similarly with the files, arguments also take a reference code, so that they can be retrieved by the ProxyWS.

Security in data transport is enforced by the VRSServer implementations or the data resources. In this case, HTTP was easy converted to HTTPS, to ensure the privacy of data streams. However an issue that needs further attention is the privacy of the SOAP communication, especially when a client or a workflow engine requests streaming URI from the ProxyWS. In this case the response at least should be encrypted to prevent any eavesdroppers from obtaining the URI that will stream sensitive data.

2.3.1 Proxy Call of a legacy web services

Calling a legacy web service requires from the ProxyWS to be deployed in the same container as the target web services. This is because the ProxyWS instantiates any services loaded into Axis classpath. So in order to call a legacy web service via the ProxyWS one may use the methods: `callService`, `callServiceReturnObject`, `asyncCallService` or `asyncCallServiceReturnObject` described in Section 2.2.1. More specifically these methods take as arguments the name of the target service, the method of that service and the service's input arguments. In case that the target service need as input arguments, files or data located in a remote location (file server, or VRSServer), the arguments have to be passed as a URI reference. In this case the ProxyWS will resolve those arguments, fetch the remote data and pass them as arguments to the target service; otherwise the input arguments are normally passed via the SOAP call. These referenced arguments include:

1. Data uploaded via the VRSServer, and kept in memory.
2. Files uploaded in the local file system obtained via the VRSServer.
3. Data objects downloaded in memory from a remote location.
4. Files downloaded in the local file system from a remote location.

The referenced arguments presented here, try to cover the needs of making proxy calls to target services, while the data needed for these calls, might be located anywhere and under different formats.

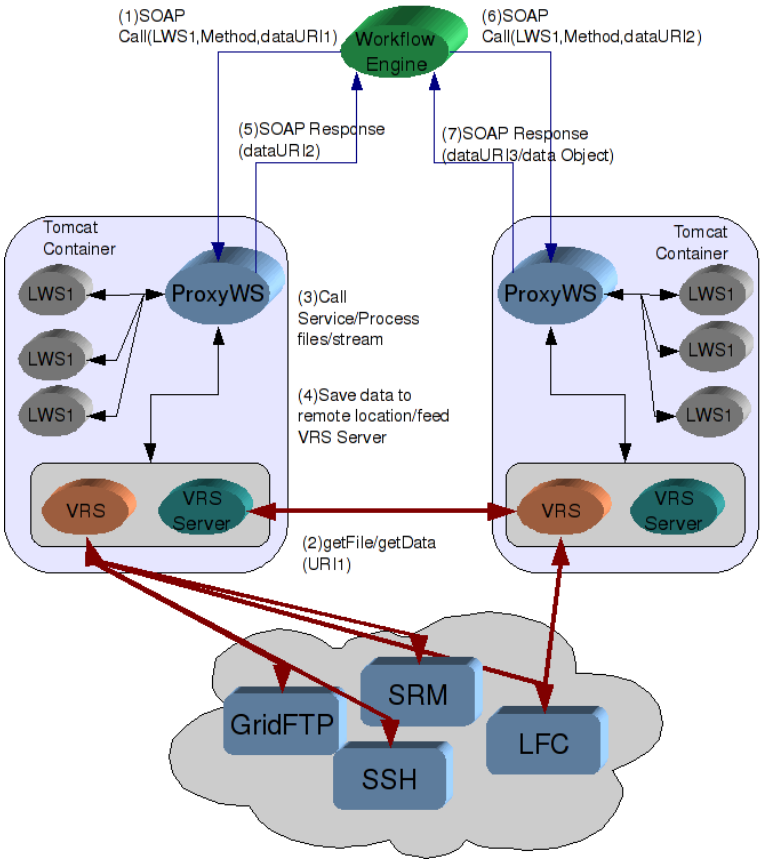

Figure 2.4: The ProxyWS architecture - With the use of the VRS and the VRSServer, web services may access data from remote locations or other web services.

Figure 2.4 shows the architecture of the ProxyWS, as well as the steps required in order to make a proxy call to a legacy web service. Such a call takes the following steps:

1. The client invokes the ProxyWS via a SOAP call that contains the name of the target service, its method and the required arguments. If for example the target method takes two arguments, a file path and a string, the file path argument has to be passed as a URI. This URI can take two forms i) `wsdt://input.remote.files/dummypath?scheme://host:port/path/filename`, in the case where data are located in a remote location, ii) `wsdt://input.local.files/dummypath?key` if files are uploaded to the ProxyWS by the client before the call. If on the other hand the an input argument is a large array unable to fit in a SOAP message the client will have to upload it before hand, using a VRSClient, and passing in its place the argument: `wsdt://input.local.data/dummypath?key`.

2. When receiving the call, the ProxyWS will resolve any referenced data. If these include any remote data, files or data objects, the ProxyWS will use the VRS to retrieve the data from the remote locations. After fetching or locating any referenced data the ProxyWS will dereference them and replace with either local paths, in the case of files, or with the data objects, in the case of uploaded data objects.

3. The ProxyWS will locate the target service, create an instance of that service and invoke its method.

4. As soon as the target service has returned the result, the ProxyWS will pass the data either to a deployed VRSServer or a remote location.

5. The ProxyWS will return the URI referring to the data location.

6. The caller may pass that URI to a next ProxyWS, or fetch the result.

2.3.2 Using the ProxyWS as an API

Besides using the ProxyWS as a service for making proxy calls, one may use the developed code as an API for developing web services capable of exchanging large data sets. The effort required for such a task is minimum as it only involves implementing a method to obtain URI for data transfers, and fetching input and output streams

provided by the data context. Figure 2.5 shows the architecture of such a web service, together with the steps necessary for two services to crate a data pipeline. These steps are:

1. The client creates an asynchronous SOAP call to the producing web service with the remote location of some data resource.
2. The producing web service returns the URI where the data will be available.
3. The producing web service obtains a data stream to the remote data resource with the use of the VRS and starts processing data coming from the remote data resource.
4. The client creates a synchronous SOAP call to the consuming web services, with the location of the produced data (the URI provided by the producing service).
5. The consuming web service obtains a data stream to the producing web service and process it.
6. The consuming web services returns either the result or a URI reference to the data.

When used as an API the ProxyWS, comes much closer in realizing the goal of truly scalable web services, that exchange data through data pipelines as described in Figure 2.1, as this approach, can consume and produce potentially infinite data sizes. Nevertheless this approach might be isolated in specific applications that mainly process audio, video, and images.

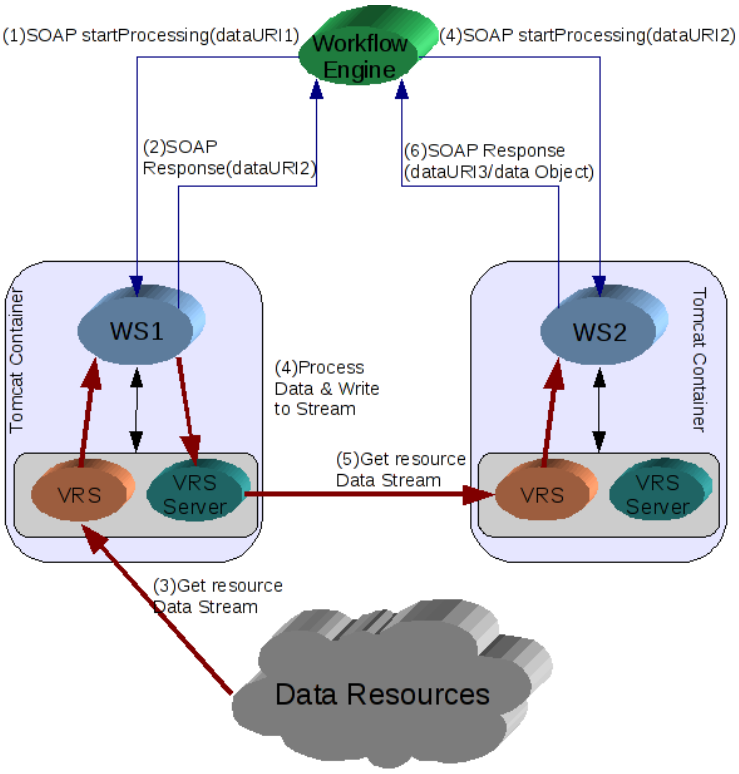

Figure 2.5: The ProxyWS as an API - The ProxyWS and the VRS, can be used as an API in order to develop stream enabled web services

3

Use Cases

Typical e-Science applications developed with the use of web services and workflows, can be effectively broken down into two steps: i) produce or obtain data and ii) process data. This section presents four different use cases that in essence follow these two steps. The first use case is a set of simple benchmarking workflows, aiming to test the scalability of the ProxyWS under increasing data transfers, while the rest are taken from existing e-Science applications. For these use cases, a comparison is made between their original implementations and the ProxyWS approach. In all the use cases the advantages of using the ProxyWS is examined in terms of enabling "legacy web services" to scale the data produced and consumed. Furthermore the advantage of using data streaming between web services is measured mainly in terms of execution time, and data volume. Additionally two of the use cases come from the information retrieval domain. The first is used in order to determine if the ProxyWS can offer large data transfers solutions for web services, while the second is there in order to investigate how the ProxyWS can scale data transfers for legacy web services, as well as to determine the usability and benefits of direct data streaming. For the last use case (visualization pipe line), we primarily investigate the benefits of delivering data directly to consuming web services. Table 3.1, provides a summary of each use case and their variations in terms of data transfer characteristics.

Usecase	Workflow Type	Protocol Used for Data Transfer	Data Flow
Benchmark	SOAP	SOAP	Orchestration
	Proxy	HTTP	Distributed Massage Based
	Stream/API	HTTP	Distributed Stream Based
Indexing	Proxy	HTTP	Distributed File Based
Search & NER	SOAP	SOAP	Orchestration
	Proxy	HTTP	Distributed Massage Based
	Stream/API	HTTP	Distributed Stream Based
VTK	SOAP	GridFTP	Centralized File Based
	Proxy	GridFTP, HTTP	Distributed File Based

Table 3.1: Use Cases. - Summery of the use cases, in terms of data transfer characteristics

3.1 Benchmarking

Since the ProxyWS is trying to address data transport issues for web services, it should be able to scale with the underling applications and also provide stability and scalability. To ensure these properties, as well as to investigate any advantages over SOAP for data transports, the ProxyWS has being tested against a set of simple but increasing data intensive workflows, that demonstrate a common workflow setup for e-Science applications. Under this set-up, a workflow engine invokes a data producing web service, passes the data to a consuming web service that will process that data, and return its results for archiving. Thus the workflows have three distinct modules: i) a workflow engine that makes RPC calls to two web services, ii) a producing and ii)a consuming web service, that generate and consume data respectively.

More specifically the producing web service generates a random string of increasing size, that needs to be encoded by the consuming web service and finally return the result back to the workflow engine for archiving. The first workflow, presented in Figure 3.1, uses only SOAP, and goes through the flowing steps:

1. The workflow engine invokes the producing web service, to generate a random string of a specified size.
2. The producing web service returns that string back to the workflow engine.
3. The workflow engine invokes the consuming web service to process the random string.
4. The consuming web service, encodes the string (in base64), and returns it back to the workflow engine.
5. Finally the workflow engine passes the data to a local module (a simple file writer), for archiving.

Next the same process is repeated, but instead of making calls directly to the producing and consuming web services, the calls are now directed to the ProxyWS, of each container. This is implemented by the ProxyWS workflow (a) presented in Figure 3.2, and it includes the following steps:

1. The workflow engine, sends a proxy call to the ProxyWS of the producing side.

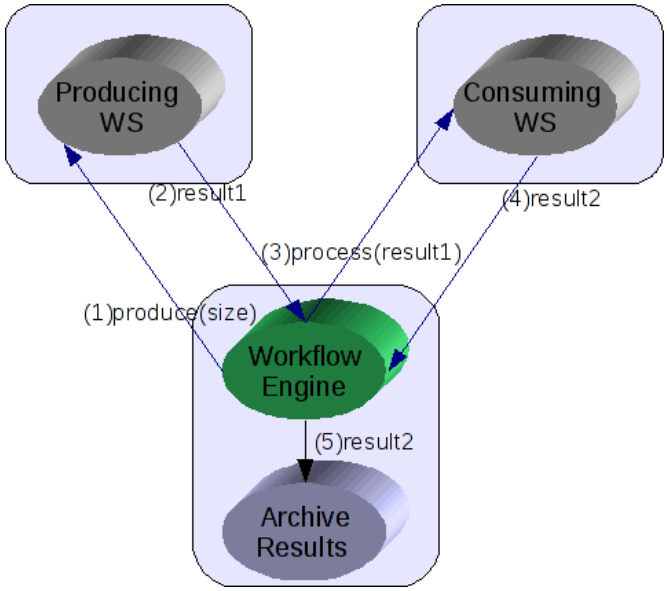

Figure 3.1: SOAP workflow - In this all communications are made through SOAP

2. The ProxyWS (of the producing side) instantiates, and invokes the producing web service, passing as arguments the size of the string to be generated. After the target service is done the ProxyWS obtains the result and references it.

3. The same ProxyWS passes the actual data to the VRSServer, and the URI reference to the workflow engine. In this case the reference is an HTTP URL in the form of http://host:port?key.

4. The workflow engine invokes the ProxyWS of the consuming web service, passing as argument the referenced data, a URI in the form of wsdt://input.remote.data/dummypath?http://host:port?key.

5. The data URI is resolved, and passed to the VRS (of the consuming side).

6. The VRS obtains the data from the VRSServer of the producing side, and the actual data are passed to the ProxyWS of the consuming side.

7. The ProxyWS, invokes the consuming web service, with data obtained directly from the producing side.

8. The result is passed to the ProxyWS, which finally returns to the workflow engine via SOAP.

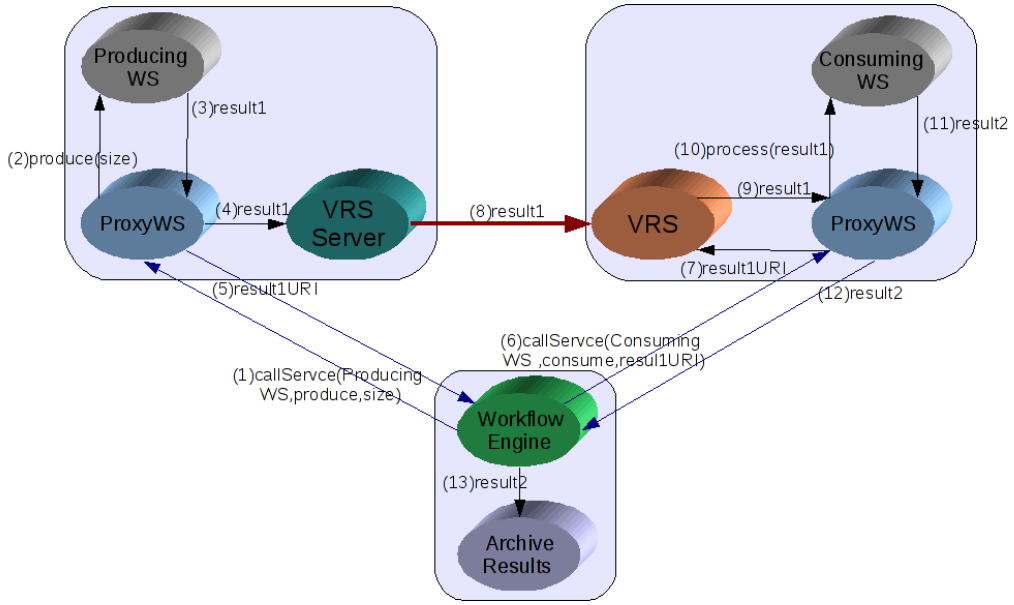

Figure 3.2: ProxyWS workflow (a) - In this workflow calls are directed to the ProxyWS, while results are returned to the workflow engine via SOAP.

The workflow described above amuses that the workflow engine, has no other means of communication, other than SOAP. In the case where the workflow engine can obtain data with the use of the VRS, the ProxyWS workflows (b, c) presented in Figure 3.3, where used. These workflows follow the same steps as the previous workflows, except that the ProxyWS of the consuming side returns a URL that refers to the result. Form that point two variations were considered. In the first (ProxyWS workflow (b)), the workflow engine passes the result URL to the archiving module, which first obtains the data, and then writes them to a local file, while in the second (ProxyWS workflow (c)), a stream is opened to a local file, and data are written to that file, as they are streamed from the consuming side.

Finally direct streaming is implemented with the Streaming workflow presented in Figure 3.4. In this workflow both the producing and consuming web services have being implemented with the ProxyWS API. This provides both services with streaming capabilities. The execution of that workflow is as follows:

1. The workflow engine makes an asynchronous call to the producing web service.

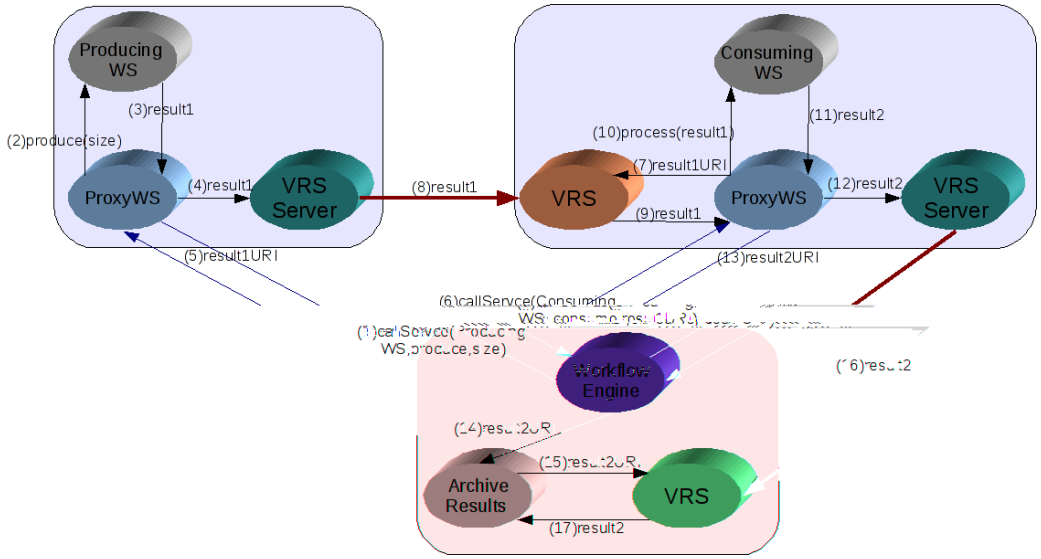

Figure 3.3: ProxyWS workflow (b, c) - In these workflows, calls are directed to the ProxyWS, while results are returned to the workflow engine via a URL. The data archiving is made by either obtaining the entire data and then saving it(b), or by opening a stream to a file and copying the data from the consuming service (c)

2. The producing web service starts generating data, while passing them directly to the VRSServer.

3. The producing web service returns the URL where the data may be streamed from to the workflow engine.

4. The workflow engine, invokes the consuming web service, passing the URL as the argument.

5. The consuming web service now needs to open a stream directly to the producing side. This is done by passing the URL to the VRS.

6. The VRS contacts the VRSServer, on the consuming side and opens a stream that provides the data.

7. This stream is passed to the consuming web service for processing and each processed chunk of the data is re-streamed to the VRSServer.

8. At this point the consuming web service returns the URL back to the workflow engine.

9. Finally the workflow engine passes the URL to the archiving module, where it opens a stream directly to a local file, copying incoming data coming from the VRSServer.

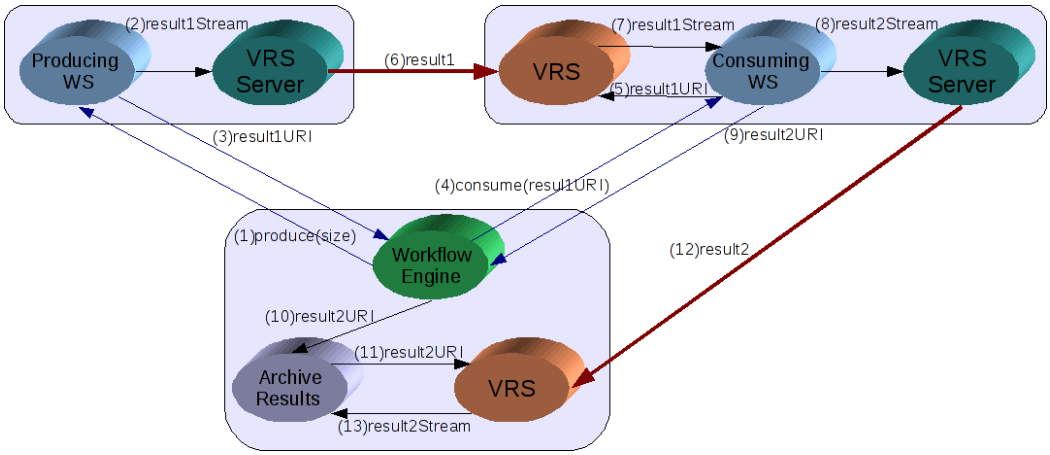

Figure 3.4: Streaming workflow - In this workflow both the producing and consuming services have being implemented with the ProxyWS API, which help in the creation of a data pipeline

For all the workflows described above, the process is repeated for an increasing data size, while measuring their execution time. Table 3.2 shows the data sizes returned from the producing and consuming web services.

The top part of Figure 3.5 shows the results, for data sizes from 1 to 6 MB, and the time needed to execute each workflow, thus the x-axis represents the data sizes returned by the producing web service in MB, while in the y-axis represents time required to execute each workflow, measured in msec. The first column for each experiment represents the SOAP workflow (Figure 3.1), the second the ProxyWS workflow (a) (Figure 3.14), the third and forth ProxyWS workflows (b, c) (Figure 3.3), and the last the Streaming workflow (Figure 3.4).

As shown from these results, the SOAP workflow, failed to transfer data larger than 2 MB. This is attributed to the workflow engine running out of memory while trying to obtain the SOAP response from the consuming web service (that is 4.01 MB). The same happed with the ProxyWS workflow (a), while returning data to the workflow via SOAP, but instead the workflow engine run out of memory while obtaining 5.34 MB from the consuming web service. Although this looks like an increase in memory of

Random string size (MB)	**Encoded (base64) string size (MB)**
1	1.34
2	2.67
3	4.01
4	5.34
5	6.68
6	8.02
7	9.35
9	12.02
11	14.7
13	17.37
15	20.04
17	22.71
19	25.38
600	801.62
800	1068.83
1000	1336.1

Table 3.2: Data sizes. - Data sizes returned by the producing and consuming web services

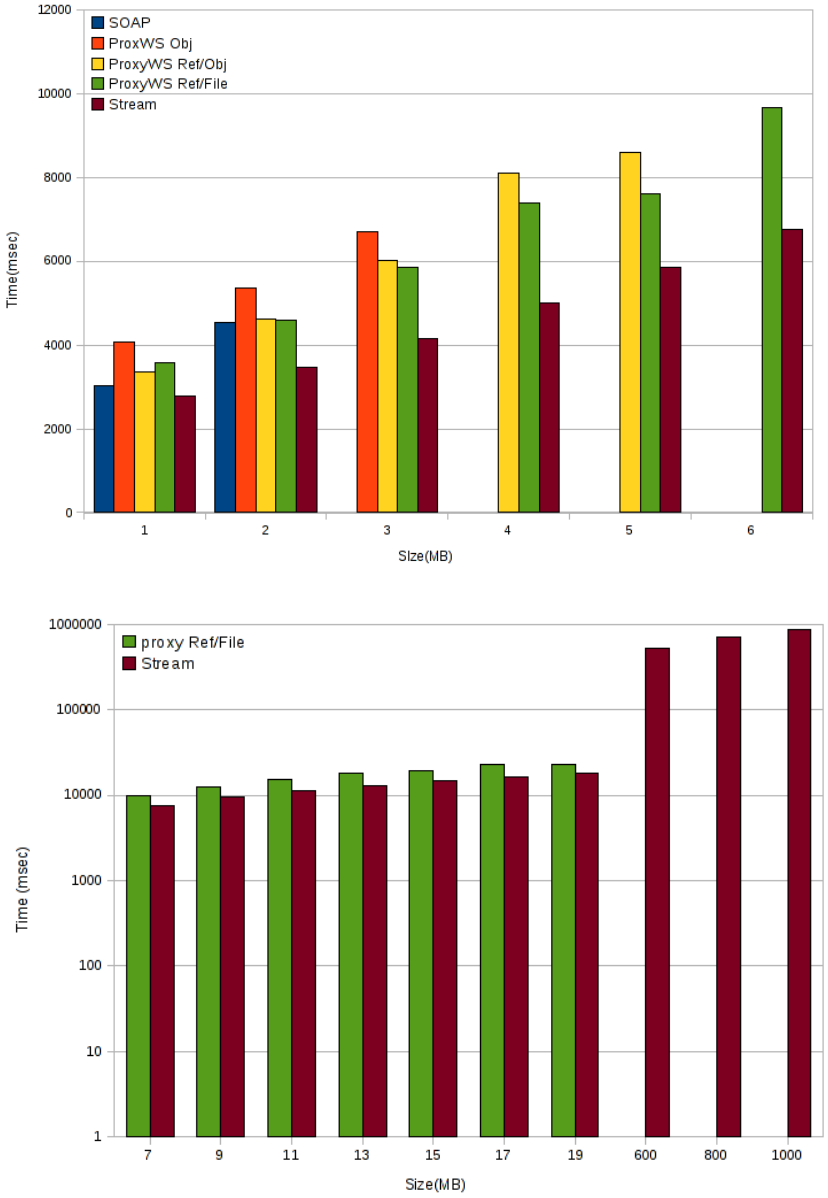

Figure 3.5: Benchmarking results. The graphs represent time measures for five different workflow implementations. In the x-axis a measure of the data produced by the producing web service is given, while the y-axis measures execution time of each workflow. In the top graph the results are given for data up to 6MB, while in the bottom the measures where made for up to 1GB.

around 1.3 MB in the workflow engine, in reality in both cases the workflow reaches the limit of its available memory, which in the particular setup is below 5.34 MB since in the first case the workflow engine tried to obtain 7.01 MB (3 + 4.01) while in the second 5.34 MB, which is the data returned from the consuming service. So this increase in performance is made possible by just passing a data reference to the workflow engine, instead of the data itself.

On the remaining data sizes, 4 to 6 MB, only the last two workflows (ProxyWS workflow (b, c) and Streaming workflow) managed to cope with the data returned. ProxyWS workflow (b) managed to get up to 6.68 MB, a fact that shows the overhead introduced by SOAP. The reaming workflows managed to process and archive up to 8.02 MB.

The bottom part of Figure 3.5 shows time measures for data from 7MB up to 1GB. From these results it can be seen that only direct streaming was able to cope with data sizes from 600MB up to 1GB. This is because even when passing references from the producing to the consuming web services, the services themselves are not able to hold in memory the results generated. Thus when opening a stream from the producing to the consuming web service, and from that to the workflow engine the application is able to cope with potentiality infinite data sizes.

When looking at the execution times of each workflow, it is apparent that direct streaming is faster than any other method. This is because data are processed as they are generated from one module to the next, creating a data pipeline. For the rest of the implementations SOAP was the second faster, while the slowest implementation was the ProxyWS workflow (a), at least for sized up to 2MB. These time results are expected, when taking into consideration the fact that when making proxy calls, the ProxyWS has to instantiate the target service, as well as to communicate with the VRSServer. So in terms of scalability, the ProxyWS can provide to legacy web services same advantage, that although is not up to 600MB, it is considerable gain against SOAP, where it reached it's limit in 2MB, while the best performing ProxyWS scaled up to 19MB.

3.2 Parallel Indexing

After verifying that the ProxyWS is able to provide scalable data transfers for web services, we consider an indexing use case. Indexing is the process of analyzing and extracting content from a document, and it is applied in search engines. The contents are stored in an index in order to optimize the speed and performance of finding documents relevant to a search query. Without an index, a search engine would have to scan every document available to it, which would require considerable time and computing power. Thus before any document set can be made searchable, it needs to be indexed. For this use case, an indexing Java application was obtained from the Adaptive Information Disclosure project. The Adaptive Information Disclosure project is part of the VL-e project(65), that aims to create a Virtual Laboratory environment for e-science for several application domains. The indexing applications is part of the Adaptive Information Disclosure Application (AIDA) Toolkit, 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 also provides a set of components which enable the indexing of medline documents, obtained from the Medline database (66). Medline provides a bibliographic database produced and maintained by the U.S. National Library of Medicine (NLM), which contains a collection of 17 million plus citations of bio-medical literature. These documents encode richly structured data, in XML formats, about bio-medical publications which include authors, affiliations, titles, abstracts, grants, medical subject headings etc. Medline releases approximately 5000 new citations per day, while each year users can obtain the annual data set of documents. The 2009 medline, baseline database contains 17,764,826 records and requires 63.9GB disk space (8.4 GB compressed).

Since AIDA provides search and learning tools, it needs to provide an index where learning and search applications can look for bio-medical information. For that reason an indexing application was developed in Java, able to index medline files. In principle this application is given a local path which contains medline files, it indexes these documents, and reruns the path where the index may be found.

Considering the advantages of SOA, the indexing application provided by AIDA, was service enabled to form the IndexerWS. In doing so, we provide the means for

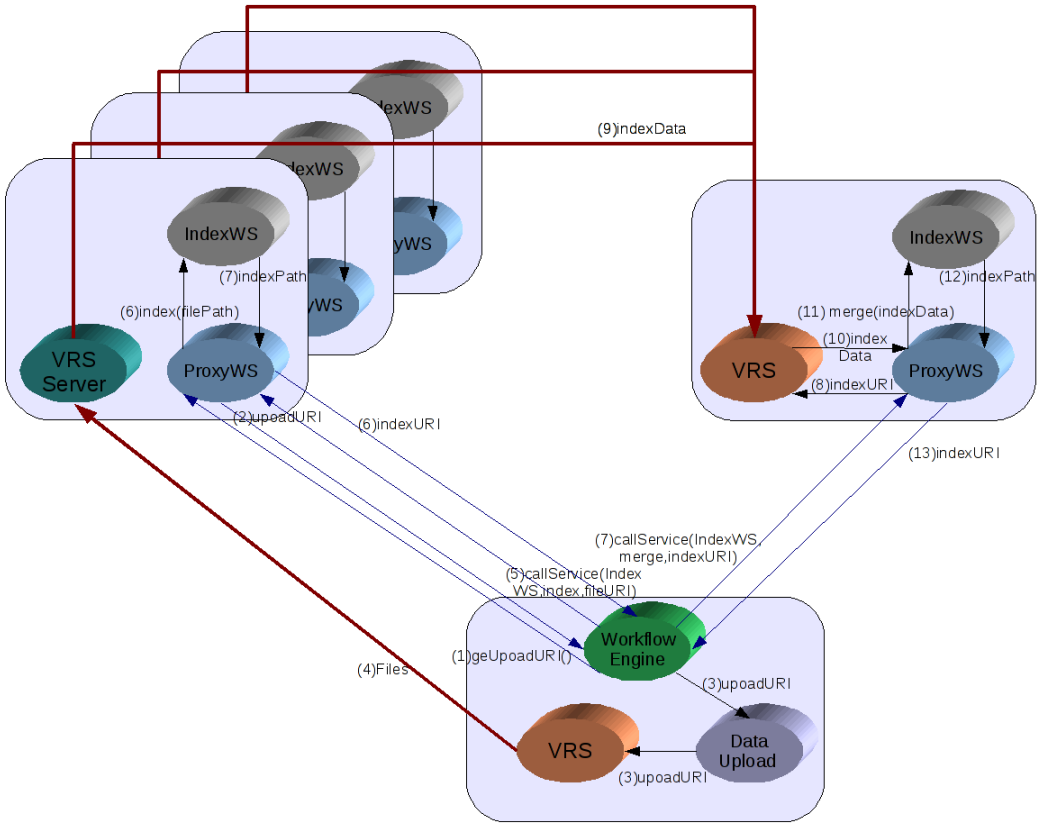

Figure 3.6: Parallel Indexing workflow - In this workflow several web services are used to speed up the indexing process of medline documents.

flexible construction of new workflows that may add, delete, and modify documents. But before this web service can index any documents, we need to transfer the documents to the web service. While it takes minimum effort to turn a Java application into a web service, this web service may be seen as a computational island, unable to get data in or out. Although the IndexerWS could be implemented with the ProxyWS API, it is kept as a separate web service allowing the distinction of the actual indexing and data transport operations. Additionally, the indexing operation of the IndexerWS is computationally and data intensive, e.g., in the case where it needs to index the entire 2009 Medline data set. Keeping the two operations separate allows for more modular approach, where the IndexerWS may be a front-end of a computing cluster or Grid application that will allow for even grater scaling. In this use case to be able to index large numbers of documents, the indexer web service together with the ProxyWS is deployed on multiple nodes, thus enabling a parallel indexing operation. Figure 3.6 shows this parallel indexing operation, which although assumes that the workflow engine has the entire set of documents for indexing, they can be located in any remote location. So to index this data set, the workflow engine sends to each node a subset of the documents location. Each node brings the data to the indexer web service location, and starts indexing. When a node is finished indexing its subset of documents, it sends its index to a master node that will merge all the sub indexes into the universal index. More specifically the steps of this workflow are:

1. The workflow engine gets the URI where files can be uploaded on each node from the ProxyWS. This URI may be an FTP server, or in this case an HTTP servlet able to upload files.

2. The response is returned to an uploading module, which with the use of the VRS will upload a subset of the files on each node.

3. The workflow engine will make a proxy call to the ProxyWS to start the indexing operation, passing as argument the path the documents are located in.

4. The ProxyWS will resolve this path, and pass it to the indexer web service.

5. After the indexing operation is done the indexer web service will return the local path where the index is located.

6. The ProxyWS will reference this local path into a URI (in this case a URL), and rerun it to the workflow engine.

7. The workflow engine will now call the indexer web service to merge the sub index into a universal index, with the use of the ProxyWS.

8. When the ProxyWS receives the merging proxy call, it will resolve the URI, download the sub index from each node, and dereference the data into a local path.

9. After the merging operation is over, the index web service will return the local path where the universal index is located.

10. Finally the ProxyWS will reference and return the index location.

This workflow was measured in terms of execution time, speedup[1] , and efficiency[2], while using document sets of 29, 119, 545 MB and finally the entire data set of 8.4 GB.

Figures 3.7, and 3.8, show the relevant results. When considering the speedup results for 8 and 16 nodes, one can see that for 29 MB and 119 MB the performance drops significantly, something that can be also verified by the efficiency results seen in Figure 3.8. To identify the source of the bottleneck, a break down of the total execution time for indexing a data set of 29MB, is presented in the top part of Figure 3.9. In this graph, the total execution time is broken down in pre-stage time, the time needed to upload the subset of files in the individual nodes, execution time, the time need to do the actual indexing, and the post-stage time, the time needed to merge the sub-indexes in to the universal index. These results clearly indicate the source of the bottleneck, is the post stage time, which increases significantly for 8 and 16 hosts. The execution time on the other hand is decreasing linearly, while the pre-stage time is almost negligible. Next, in the bottom part Figure 3.9 shows the total execution time broken down in pre-staging time, execution time and post-stage time while indexing the 8.4 GB data set. From this graph it is obvious that the most time consuming step of the indexing workflow is the post-stage part, which although less than the execution time when

[1]Speedup refers to how faster a parallel implementation is when compared with a sequential one, and it is given by : $S_p = T_1/T_p$, where T_1 is the execution time of the sequential implementation, while T_p is the execution time of the parallel implementation with p processors.

[2]Efficiency measures the utilization of processors, indicating the time spent on computation and communication, and it is given by: $E_p = S_p/p$

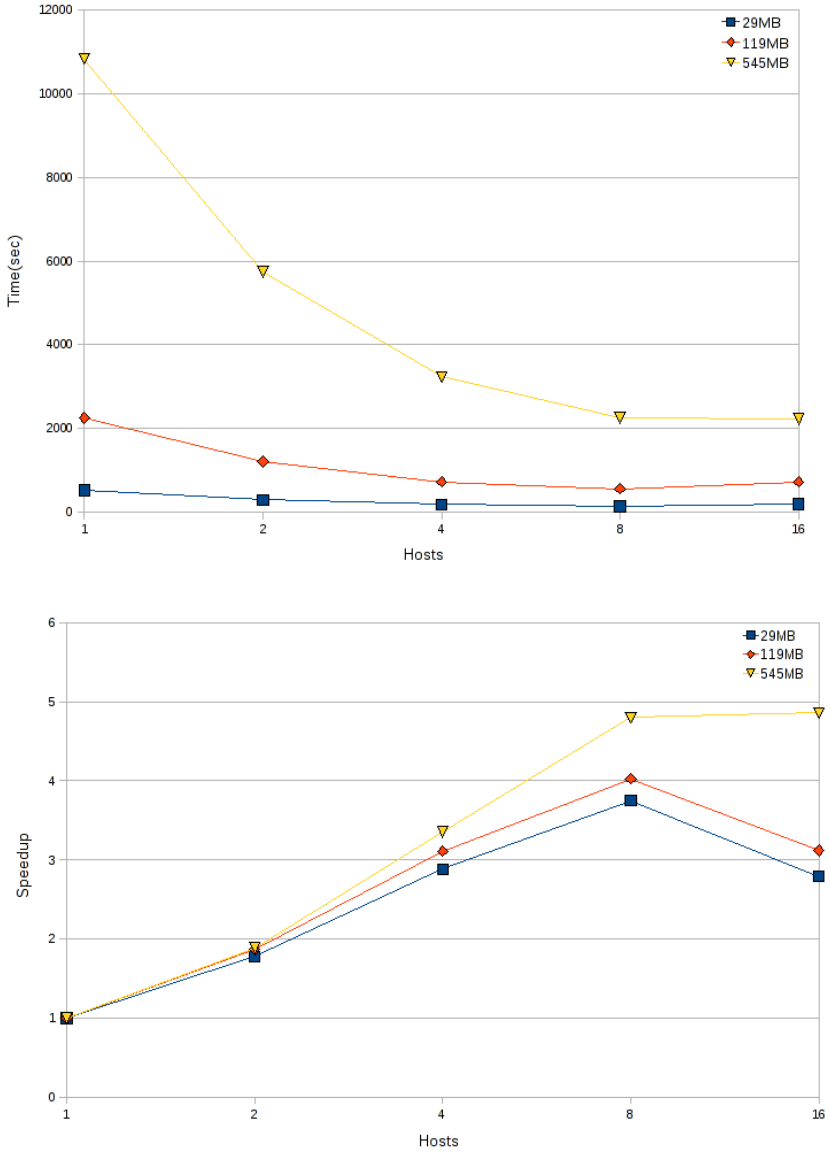

Figure 3.7: Execution time and Speedup of Indexing workflow. Execution time and speedup for 1, 2, 4, 8, 16 indexing web services. For indexing data sets of 29 and 119 MB and 8 hosts or more the execution time increases, while speedup drops

using 4 nodes, it almost doubles each time we add more nodes. This dramatic drop of performance is more evident when 32 nodes are used, where the pre-stage time makes for almost the entire execution time.

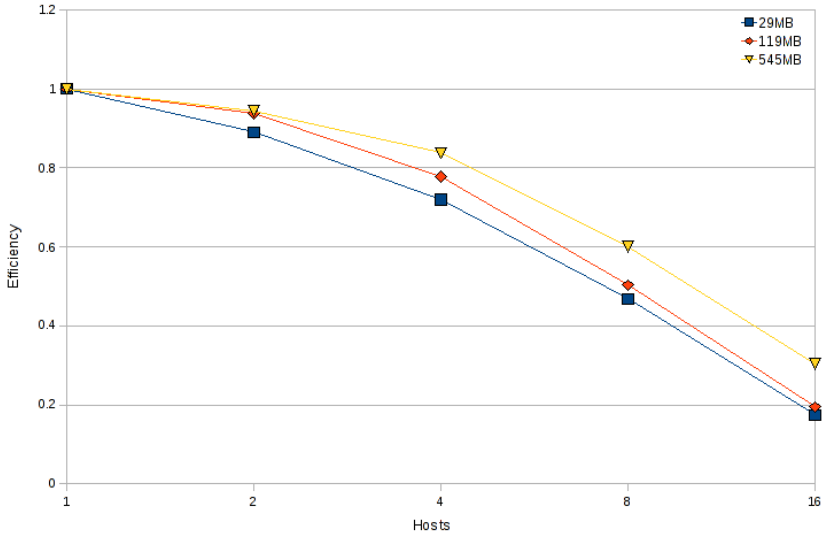

Figure 3.8: Efficiency of Indexing workflow - Efficiency for 1, 2, 4, 8, 16 indexing web services. From this graph it may be seen that the actual computation drops when using 8, or more hosts, for data sets of 119 and 29 MB

3.3 Search & Name Entry Recognition

Having obtained an index representing a large set of documents is the first step towards obtaining some useful knowledge from this set of documents. The index generated from the workflow described in Section 3.2, can be searched, with the use of the searcher web service, given a query, the name of the index to be searched, and the relevant filed (abstract, main text body, etc.) one wishes to get results from. The results returned, are passed for Named Entity Recognition (NER) using the NERcognizer service, in order to support knowledge extraction. NER aims to extract and classify information units in text such as names, biological species, temporal expressions, organizations, numeric expressions including time, date, money, and percentages (67). So to support knowledge extraction the workflows shown in Figures 3.10, 3.11, 3.12 where evaluated in terms of execution time and scalability. These workflow where based in the work presented in (68), and also in relevant workflows located in myexperiment web site (http://www.myexperiment.org/workflows/72 and http://www.myexperiment.org/workflows/74). The steps for the first workflow are:

1. The workflow engine invokes the searcher web service, passing as arguments the

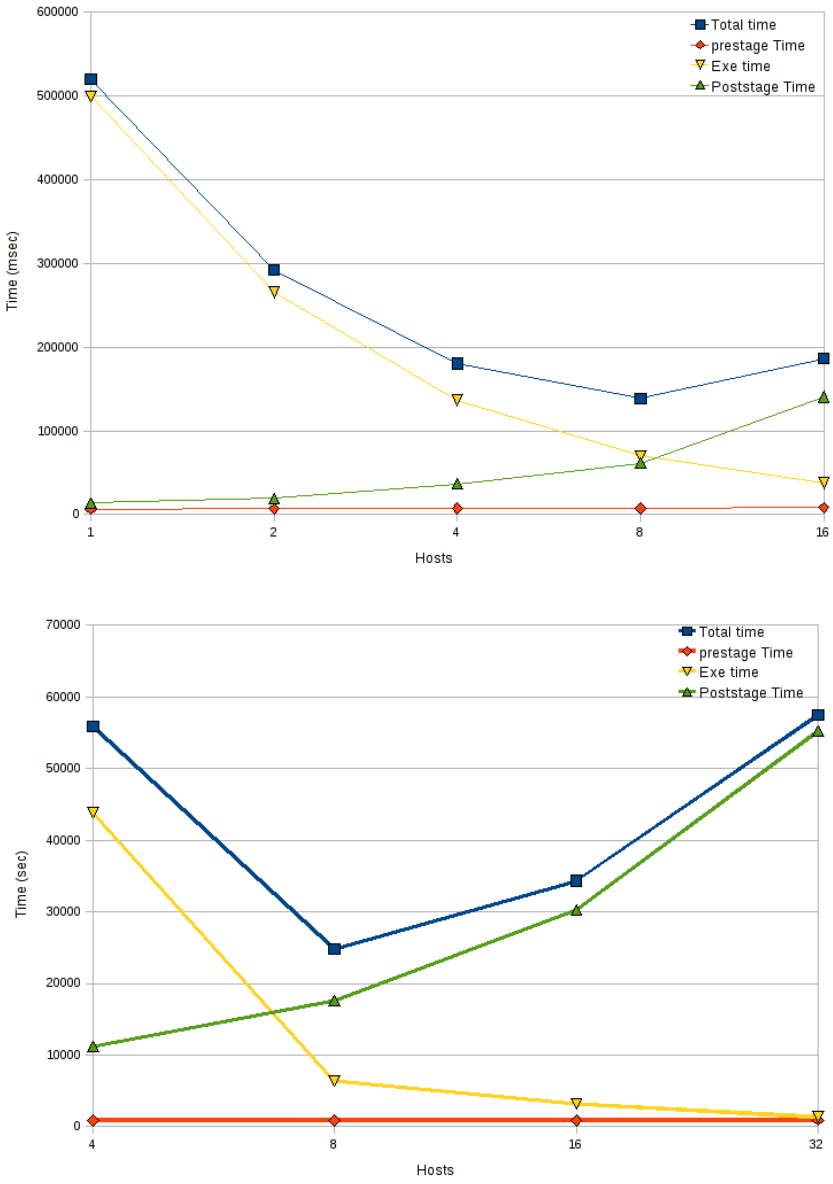

Figure 3.9: Execution time break down. Execution time for indexing a data set of 29MB (top) and 8.4GB (bottom), broken down in pre-stage, execution, and post-stage time. It is obvious that the bottleneck is the coming from the post-stage time.

query, and the maximum number of documents returned.

2. The result is returned to the workflow engine, and from there to the NERcognizer web service.

3. The NERcognizer web service identifies the entities, and returns the results for archiving or further processing.

The next workflow shown in Figure 3.11, executes the same steps, but instead of invoking the searcher and NERcognizer web services directly, the call is passed to the ProxyWS, which provides the query results directly to the NERcognizer, instead of passing them through the workflow engine. A third workflow, shown in Figure 3.12, is implemented and it follows the some procedure as the one described previously, but this time the two services are using the ProxyWS's API to transfer data between them. This workflow goes through the following steps:

1. The workflow engine invokes the searcher web service.
2. The searcher returns the data reference to the workflow engine.
3. The workflow engine invokes the NERcognizer service passing as argument the data reference.
4. The NERcognizer service connects to the searcher web service, and starts streaming the query results.
5. The identified entries are send back to the workflow engine via SOAP.

The process of the workflows described above, is repeated for 100 up to 8300 documents (that is the number of documents returned by the searcher web service), using the same query. The relation of the number of documents and the size of the annotated text returned by the NERcognizer web service may be seen in Table 3.3, while the results obtained by the three workflows can be seen in Figures 3.13.

As seen from the results of these three workflows, SOAP failed to scale for more than 1100 documents. This is because the workflow engine, could cope with the size of the two returned SOAP messages (the query results and the NER). In Figure 3.13 (bottom) the results of the Streaming and ProxyWS workflows are presented. These workflows managed to cope with the data requirements of 8300 documents. In terms of execution time the ProxyWS workflow has clearly the advantage when retuning up to 1100 documents, as the results don't have to go through the workflow engine. When comparing the performance between SOAP and Streaming workflows, one can see that

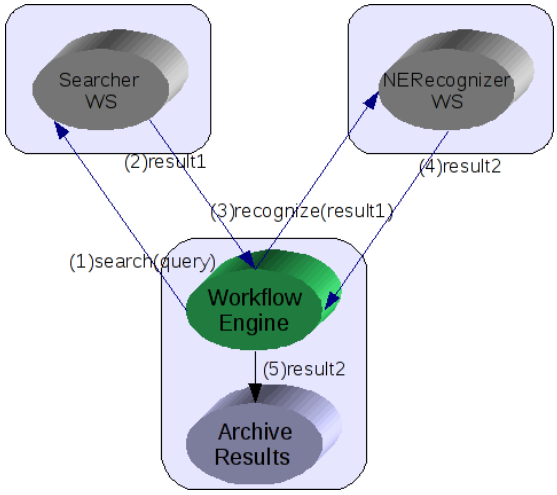

Figure 3.10: Search and NER SOAP workflow - In this workflow a query is passed to the searcher web service, while the query result is passed to the NERcognizer All the communications are over SOAP

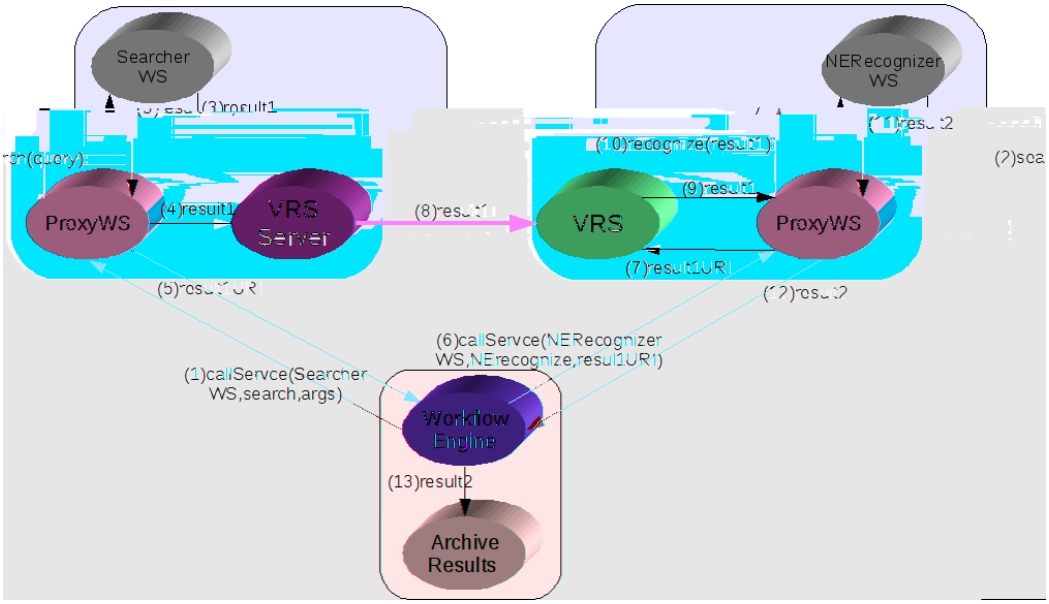

Figure 3.11: Search and NER ProxyWS workflow - In this workflow the ProxyWS is used to pass data references to the NERcognizer web service

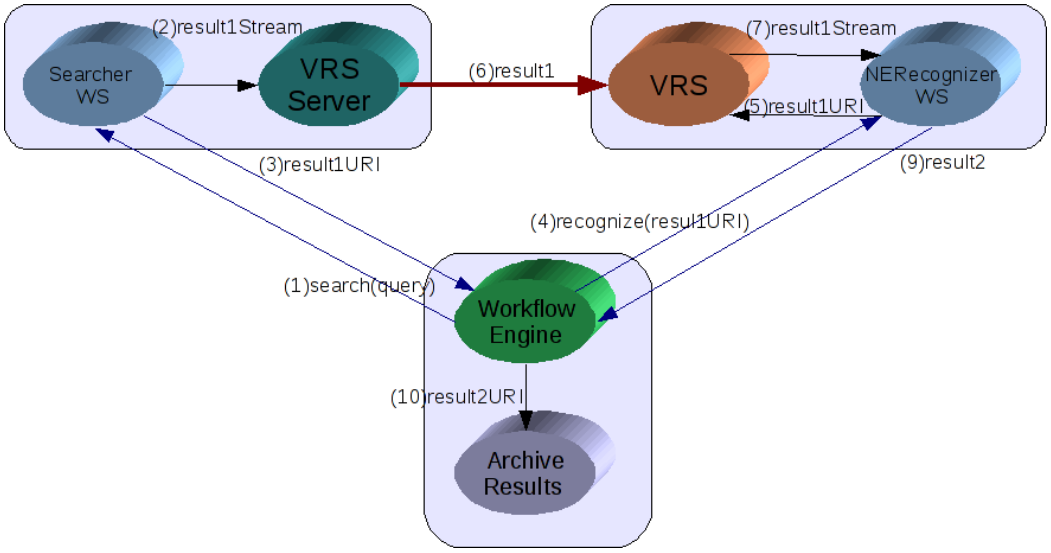

Figure 3.12: Search and NER Streaming workflow - In this workflow the two web services are streaming data directly to each other

Number of Documents	NER text (kbyte)
100	9.46
300	27.54
500	45.58
700	63.61
900	81.65
1100	99.68
1900	171.84
2700	244.07
3500	316.24
4300	388.35
5100	460.5
5900	532.62
6700	604.69
7500	676.65
8300	748.83

Table 3.3: Data sizes for number of documents returned. - The first column represents the number of documents returned by the searcher web service, while the second the size of the NER text returned by the NERcognizer

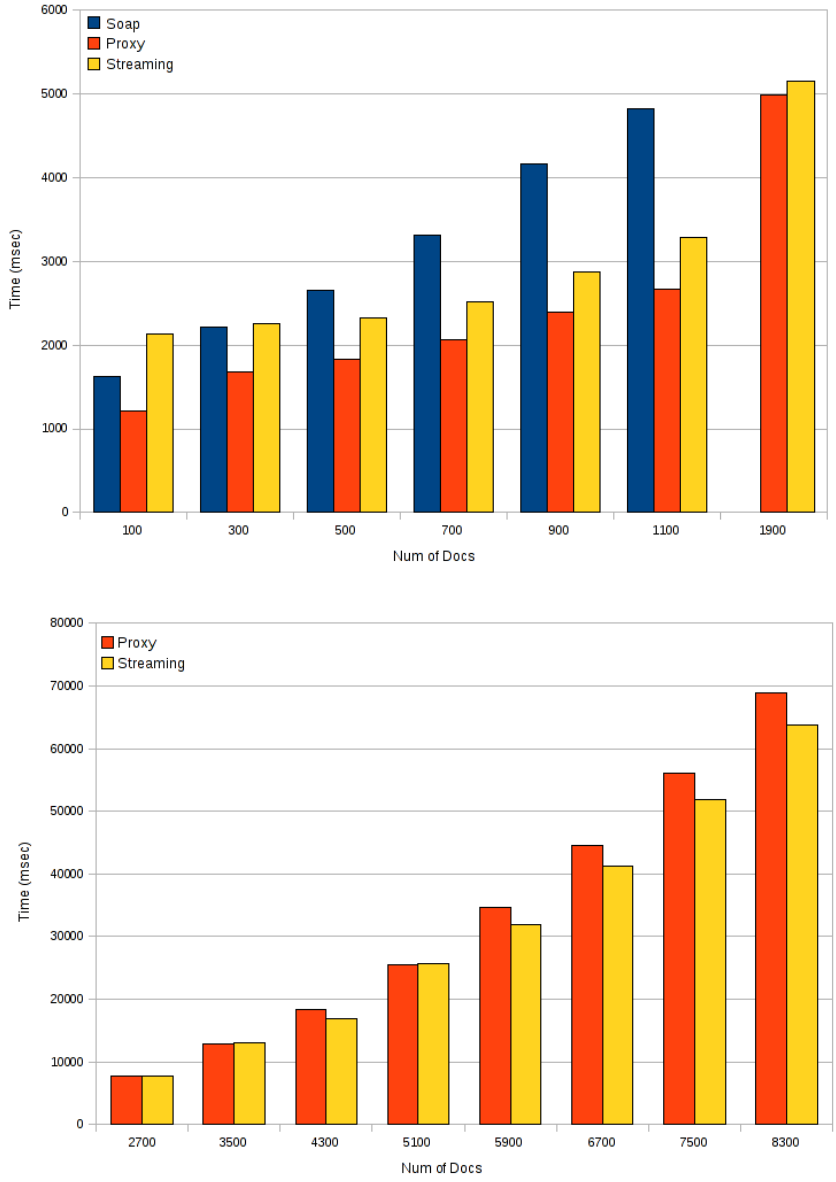

Figure 3.13: Execution times for the Search and NER workflows. The first column for each experiment is the execution time for the first workflow, the second for the second workflow, and so on. The top graph represents results for 100 to 1900 documents, and the bottom 2700 to 8300 documents

the overhead of stating a connection between the two services, is grater than passing SOAP messages, at least when returning 100-300 documents. This advantage no longer

exists when processing more than 500 documents. When trying to process more than 4300 documents it is clear that Streaming workflow scales better than the ProxyWS workflow, although the advantage gained, over the effort of developing the services with streaming capabilities, is not significant. Nevertheless if the searcher returned more than 8300 documents the execution time difference between the two methods would be grater.

3.4 Visualization WS

Up to now all use cases presented, involve web services that had no means of transferring data from a producing to a consuming web service at least in their original implementation. This use case is an implementation of a visualization pipeline based on the work presented in (69) the workflow of which may be seen in Figure 3.14. This workflow is compared against the one seen in Figure 3.15, to investigate time execution advantages, when files are not stored in remote locations, but instead are passed directly from one service to the next.

A visualization pipeline, may be described as the process of turning data into a visual representation. This process is divided into distinct steps, where every step depends on the result of the previous. These steps are described below, while a pipeline seen in Figure 3.16

1. Data acquisition. Input data are gathered from some source, like MRI-scanners, some simulation, experiments, or some form of measurement.

2. Filtering. In this stage of the pipeline, data are filtered to more relevant, sets for the problem at hand, for example extracting a scull from an MRI scan. In other words in this stage one chooses what it wants to visualize.

3. Mapping. The filtered data are mapped into geometric primitives, like points, lines, etc. At this point there are a number of primitives suitable for visualizing the focused data. This poses a trade-off between expressiveness and effectiveness.

4. Rendering. Having all the necessary primitives it is now time to transform them into images. Before turning the geometric primitives into images, they are assigned with visual properties, like color, opacity, or shadow. The goal here is not photorealism, but rather information content.

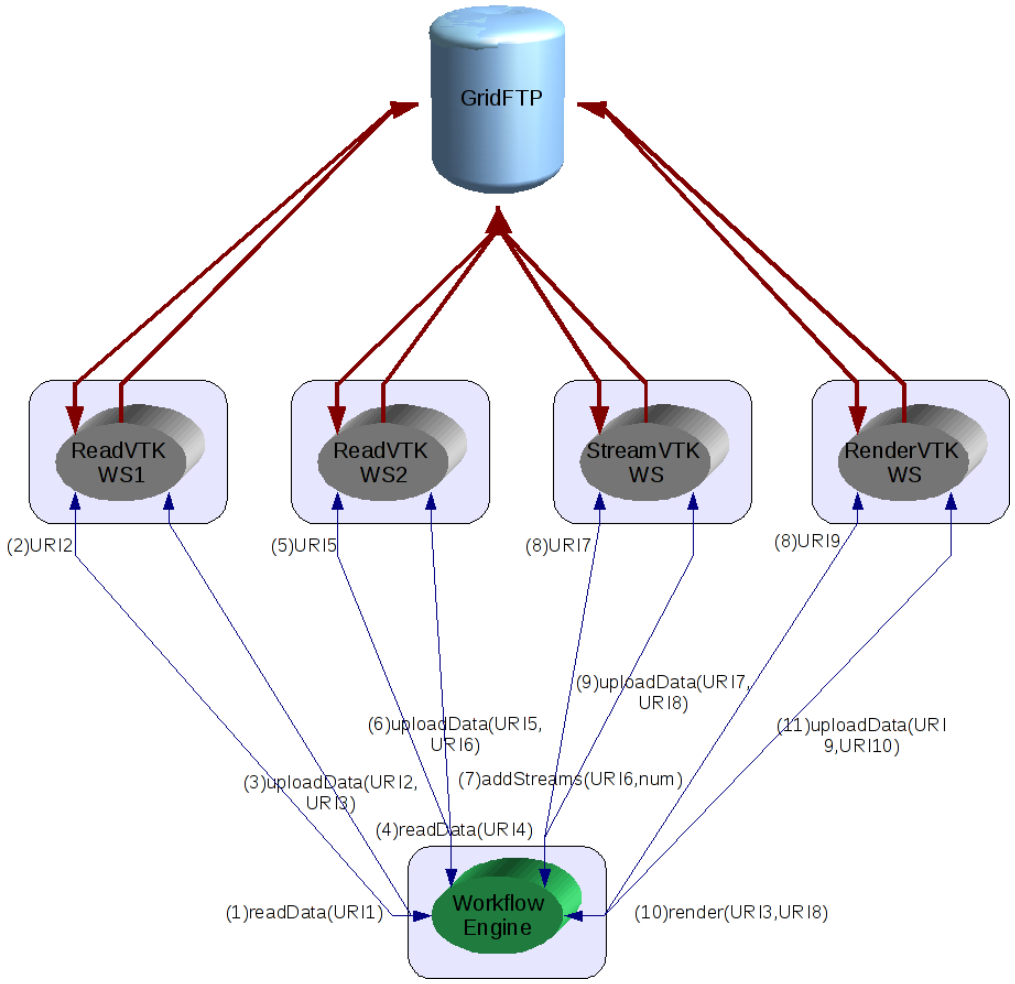

Figure 3.14: Visualization Pipeline workflow (a) - In this workflow intermediate data are stored in a GridFTP location

The two workflows shown in Figures 3.14 3.15, are used for visualizing Lattice Boltzmann data sets, with the use of VTK (70), a visualization toolkit that implements visualization pipelines. The lattice Boltzmann method (LBM) is a powerful technique used for simulating fluid flow problems in complex geometries. It is a discrete computational method based on the Boltzmann equation. According to that, fluid is modeled by particles moving on a regular lattice, where each particle at a specific lattice point is assigned with a velocity according to a distribution function. The original Boltzmann equation describes the dynamics of the one-particle distribution function, $f(r, v, t)$. This gives the probability of finding a particle at the position r

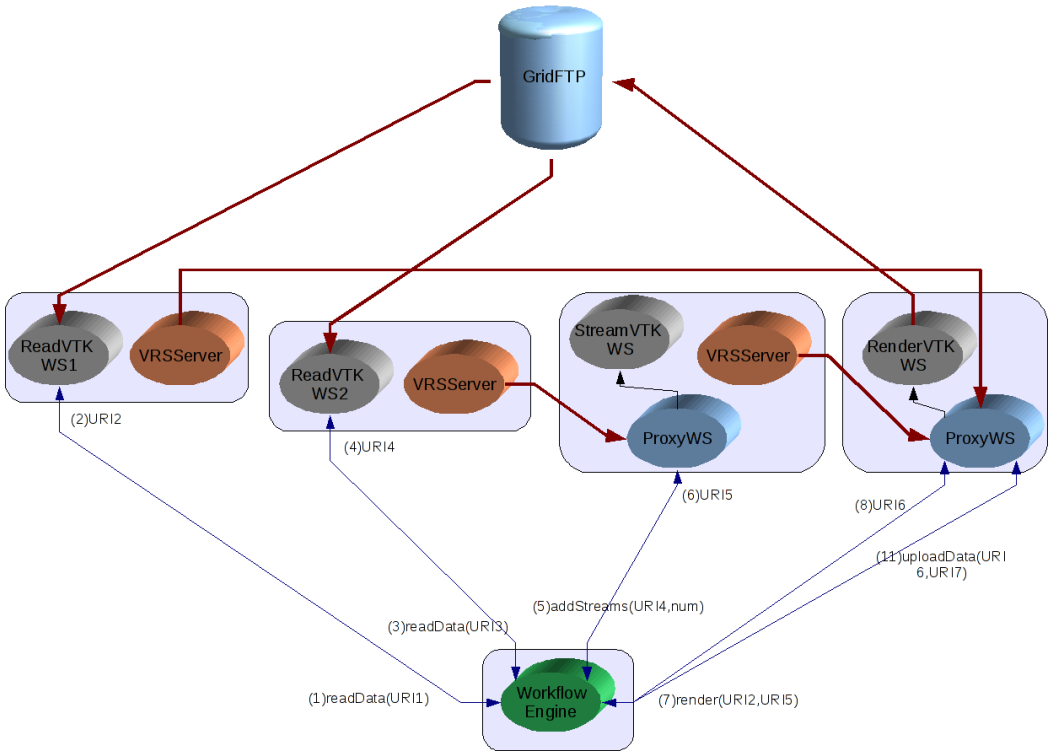

Figure 3.15: Visualization Pipeline workflow (b) - This workflow uses the ProxyWS to pass data directly to the consuming services

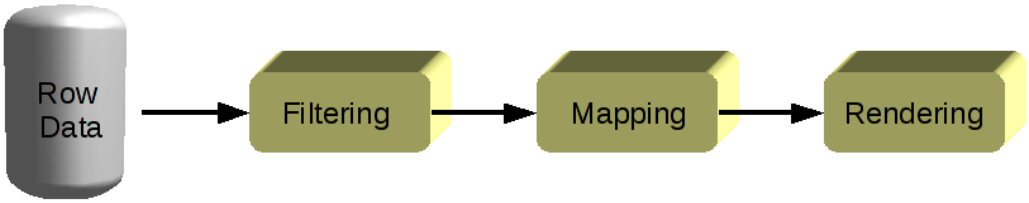

Figure 3.16: Execution time for the Search and NER workflow, for 2700 to 8300 documents - The first column for each experiment

with velocity v at time t. Discretizing the previous function one gets: $f_i(r,t)$, where the velocity vector is mapped onto a discrete set of velocity vectors indexed by i. In other words this function is the probability of finding a particle at the discrete lattice site r at the discrete time t. So by using the discrete Boltzmann function, particles are moved to their next location that is in a neighboring node in the lattice, also known as the propagation phase. Next the particles are assigned with their new velocity by also considering collisions. The results of simulating a fluid flow with LBM need to be visualized, as this might provide a better insight for the given problem than simply numerically analyzing the data. Another advantage provided by visualization is that one may concentrate on a specific part of the simulation without loosing the general perspective.

the data sets used for these workflows concern the blood flow in the common carotid artery. This artery shown in Figure 3.17 is an artery that supplies oxygenated blood to the head and neck. This artery divides in the neck ("bifurcates") to form the external and internal carotid arteries. At this location, a buildup of "atheroma" can occur that obstructs the bloodstream to the brain. Visualizing the blood flow at the bifurcation point, should reveal any anomalies in the course of one hart beat. This is accomplished with the use of the services presented in (69). Using these services the workflow of Figure 3.14 is created, and compared against the one presented in Figure 3.15, to investigate time execution advantages, when files are not stored in remote storage systems, but instead are passed directly from one service to the next. The services that make up these workflows are:

ReadVTK is responsible for reading and filtering raw data-sets.

StreamVTK maps filtered data-sets, so it may add streamlines to the flow fields.

RenderVTK renders the mapped and filtered data sets, producing the final image.

ProxyWS is present in the second workflow (ProxyWS workflow), and it is used to deliver data directly to the VTK services.

To provide to the user a better control over the visualization process, the StreamVTK service, enables the selection of the ratio of the stream line coverage of the flow field [1].

[1]This ratio controls is the number of stream lines added to the flow failed. So a ratio of 1 means that the entire flow filed will be visualized by the added stream lines

This however creates a trade-off between speed and accuracy.

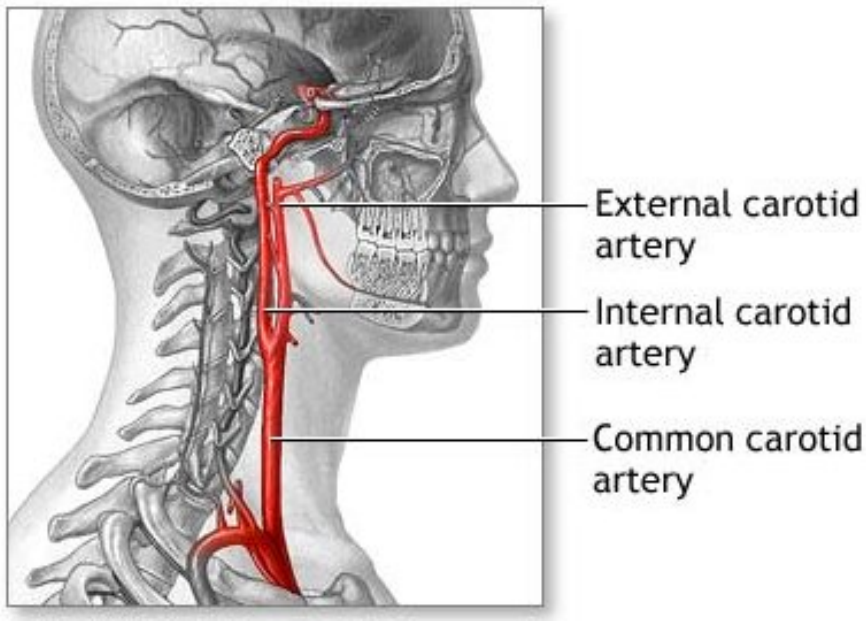

Figure 3.17: Carotid artery - The carotid artery supplies oxygenated blood to the head and neck.

In order to execute this visualization pipeline, the workflow of Figure 3.14, executes the steps given below:

1. The workflow engine invokes the first ReadVTK service to read the artery data set from a remote location (a GridFTP server).

2. The first ReadVTK service, downloads filters and maps the artery, the location of which is returned back to the workflow engine.

3. The workflow engine provides to the first ReadVTK service the remote URI (the GridFTP server) where the results are uploaded.

4. The same procedure is flowed for the second ReadVTK service.

5. The workflow engine calls the StreamVTK web service, to add stream lines to the flow data, by providing the URI of the remote server, where the flow data are located.

6. The result of the previous step is upload in the remote server.

7. Finally the workflow engine calls the rendering service by providing the URIs of the artery, and the flow field. After the data are rendered the result image is uploaded again in the remote server.

Alternatively, the workflow presented in Figure 3.15, uses the ProxyWS and the VRS to deliver data directly to the consuming services, avoiding unnecessary data hops to the remote server. More specifically this is done by flowing steps:

1. The workflow engine invokes the first ReadVTK service to read the artery data set from a remote location (a GridFTP server).

2. The first ReadVTK service reruns the URI (local file system) where the result is saved.

3. The same procedure is flowed for the second ReadVTK service.

4. Now we need to add streamlines to the flow filed data set. The workflow engine Invokes the ProxyWS in StreamVTK service to obtain the data from the second ReadVTK service, so the streamlines will be added.

5. After both data sets where read, filtered, and mapped, the workflow engine calls ProxyWS of the RenderVTK service to obtain both data sets.

6. The ProxyWS now calls the RenderVTK to generate the image. After the image is rendered, the workflow engine calls the RenderVTK service to upload the image back to the remote server.

These two workflows where measured in terms total execution time, under different configurations, and data sets. Figure 3.18, shows execution times under different configurations. Starting from the top line, the set up of each experiment is as follows: i) 66 MB flow vector data set, transferred via GridFTP from a remote location. ii) 66 MB flow vector data set, transferred via the ProxyWS to the consuming services. iii) 30.1 MB flow vector data set, transferred via GridFTP from a remote location. vi) 30.1 MB flow vector data set, transferred via the ProxyWS to the consuming services, and so on. For each experiment the ratio of the stream lines to be rendered (x-axis) is

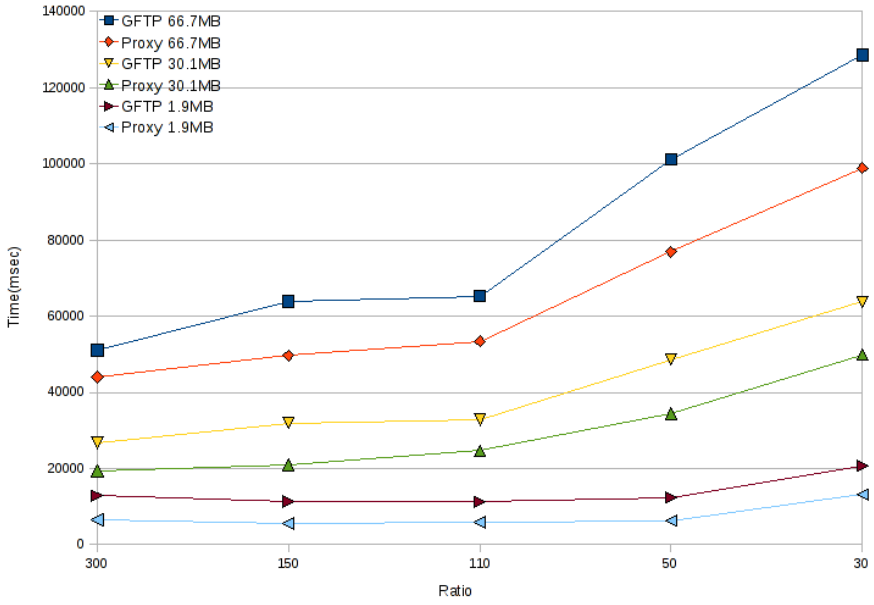

Figure 3.18: Execution time for the visualization workflows - Execution times under different configurations, for the two visualization workflows

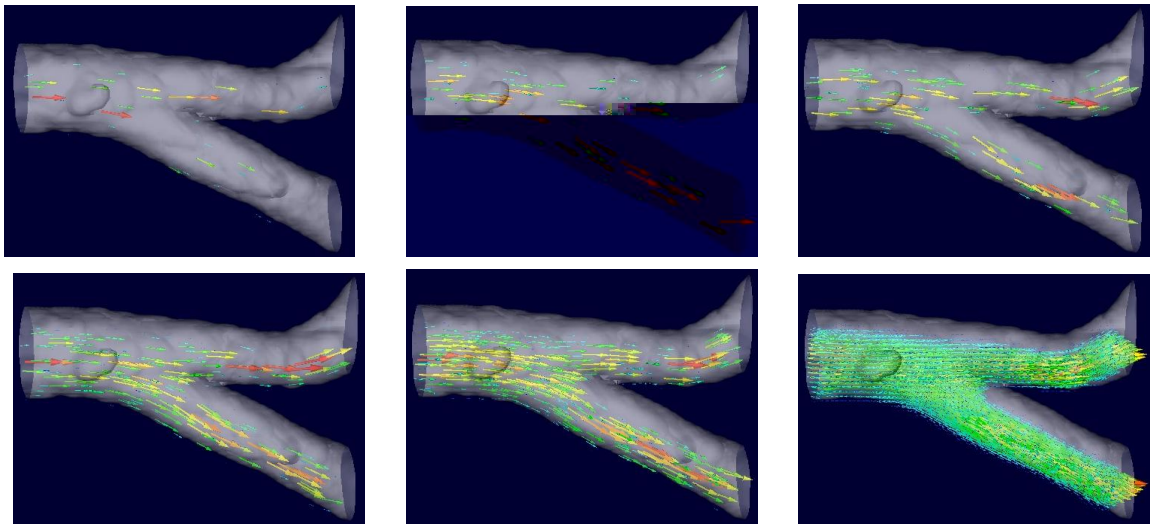

Figure 3.19: Rendered artery and flow field. Starting from the top left, the stream line ratios used are 300, 150, 110, 50, and 30, while the last image (bottom right), shows the artery with a ratio of 2. In this case it becomes vary difficult to examine the diffraction point.

set to 300, 150, 110, 50 and 30. Figure 3.19 show the different rendered images while using the ratios mentioned.

When looking at the resulted obtained, from the different workflow setups, it is clear that by transferring data directly to the consuming service, execution time can be shorted. This is especially true, as data and computation become more intensive. So in the case where the flow filed data set is 66MB and the stream line ratio is 30, the time difference between the two methods (GridFTP, and ProxyWS) is in the order of 35.14 sec.

3.5 The ProxyWS in Practice

In the use cases presented here the ProxyWS has tested in benchmarking and real life applications, in terms of data scaling, usability of the architecture, and workflow execution time.

For the first use case it is clear that SOAP falls short when it's called to transfer data, while streaming proves to be the fastest and most scalable solution, as it manages to transport 1GB of data. Proxing on the other hand falls somewhere in the middle. It can scale more than SOAP, but it can't reach streaming performance. Nevertheless it is a solution for legacy web services, where with minimum effort it provides larger data sets to these services.

The next use case (indexing) although not compared with any other variations, it did prove that with a design like the ProxyWS, SOA based applications, can handle data and computation intensive applications. The indexing application, required in total the transfer of more than 19 GB (8.4GB the document set and approximately 11GB the index generated) of data, which where all handled by the ProxyWS, while the indexing and merging operations where performed with minimum modification of the original application. Moreover the performance of the workflow was improved when we used 16 nodes for the indexing, but serious overhead problems appeared when attempting to add more nodes. Nevertheless the total data moved by the ProxyWS proves that such an approach can be used for SOA-based applications.

The search and NER application showed that the ProxyWS scaled exiting services by just deploying it in the same container. When used as an API the ProxyWS didn't

exactly follow the behavior of the benchmark tests. This can be attributed to the longer processing times of NER, as well as to the smaller data sizes generated.

For the last use case, the problem was not data size scaling, but data delivery efficiency. As the results show execution times can improve when data are delivered directly to consuming services. This approach is more meaningful in the case where lots of intermediate temporary data are created, as these temporary data would put unnecessary stress to storage resources.

Overall the use cases presented here show that the ProxyWS can be used with legacy services, in order to provide them with larger data consumption and/or production. Additionally uniform access to data resources is one more contribution. Another feature that the ProxyWS offers is direct data streaming. This feature however is not always applicable. For example in the case of the indexing applications, direct streaming could not have being used as the particular applications are file oriented. This seems to be also the case for the visualization workflow, but conceptually a visualization pipeline could be ideally fit into the creation of data pipelines. This however would require adapting the VTK toolkit accordingly.

4

Discussion

4.1 Discussion

The next generation of scientific research puts forward demands for grater collaboration, and coordination. These demands are presented by the fact that research can no longer be isolated in one particular field. Instead it must cross and combine many disciplines, in order to address and solve more complex problems.

In order to provide the tools and models that will aid scientific research, e-Science is primarily focused in creating a collaborative research environment where resources can be shared at a global scale. The infrastructure (Grids, HPC, Clouds, etc) e-Science depends on is primarily made available through SOA. Thus it seems that SOA as architecture is playing an important role in e-Science. Web services are a successful implementation of SOA, and have being used to provide access, and virtualization to resources. Web services however seem to have some shortcomings when used within the e-Science context and more specifically in data centric workflows. These shortcomings may be summarized in the following:

- SOAP is inefficient for data transports.
- Orchestration workflows move data through workflow engines.
- Third party data transfers to remote locations are not efficient, especially when lots of temporary data are created.
- Direct data streaming is not fully exploited to increase data scalability.

- Web services don't include mechanisms for accessing diverse data resources.

Previous work was being done to address most of the points mentioned, but each suggestion seems to only address part of the problem. Leveraging existing work, we have tried to address these problems with the introduction of a data-aware web service, the ProxyWS. The ProxyWS distinguishes web service's communication between control and data flow. Control messages (SOAP), are allowed to be delivered to workflow engines, in order to have better control over the execution, while data flow is redirected, and through referencing, and alternative transport channels, is delivered directly to consuming web services, whether those are legacy or new implementations. Additionally the ProxyWS, with the help of the VRS API, is able to provide uniform access to diverse data resources.

The proposed ProxyWS has being tested against real life applications, and proved that it can offer a solution to the problems mentioned earlier. First of all the life of existing web services can be extended, as they can now access, process, produce and consume larger data sets. Another contribution is that any web service is able to get access to data resources, in a uniform way. Finally data flow can be targeted directly to consuming web services, either through the use of massage based data delivery or with the use of data streams. Another feature of the ProxyWS is that service providers don't have to change, or replace their environment in order to use the ProxyWS, since it can be deployed as normal web service.

Considering those features, the ProxyWS supports scalable data transports for web services. In doing so, e-Science applications can work with larger data sets, thus meet the demands of the expected "data explosion".

4.2 Future work

Although the ProxyWS was tested in a variety of applications, there might be some scenarios that can't be covered by the current design. One of them includes the need for delivering data from one or more producing services to one or more consuming services. Such a feature would make it easier for web services to consume and combine data from different sources. If for example the visualization services could directly stream data from one service to the next, and the images produced by RenderVTK service had

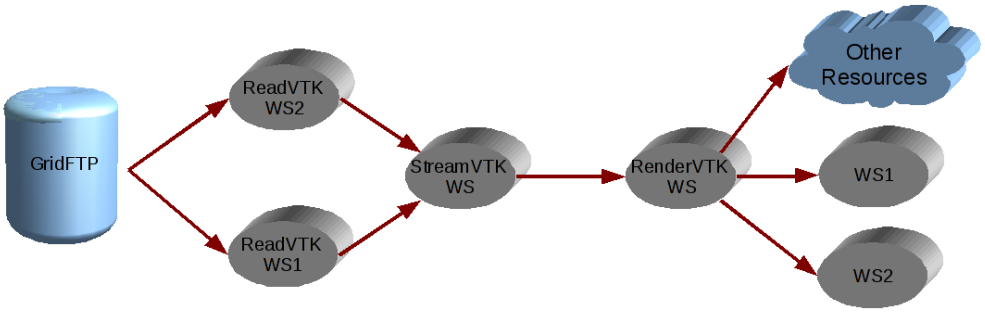

Figure 4.1: Multiple Data Streams Scenario - According to this scenario, a mechanism is needed if data streams need to be delivered on multiple resources at the same time.

to be delivered to many resources, as presented in Figure 4.1 then there has to be a mechanism that can make copies of data streams.

Furthermore, as seen from the results presented in Chapter 3, the size of the results generated from proxy calls, although lager than SOAP, are limited to available memory of the particular service container. This is because before the ProxyWS can obtain the data from the proxy call the target service has to generate the entire result. So in other words results have to stay in memory. A more efficient, and scalable technique, would be to get access to memory, and stream the results, in a file or directly to the consuming service, so that legacy web services can use the benefits of direct data streaming.

Bibliography

[1] J. Taylor. (2009, May) Defining e-science. [Online]. Available: http://www.nesc.ac.uk/nesc/define.html 2

[2] (2009, May) LHC web site. [Online]. Available: http://lhc.web.cern.ch/lhc/ 3

[3] (2009, May) ALICE web site. [Online]. Available: http://aliceinfo.cern.ch/Collaboration/ 3

[4] (2009, May) ATLAS web site. [Online]. Available: http://atlas.ch/ 3

[5] (2009, May) CMS web site. [Online]. Available: http://cms.web.cern.ch/cms/index.html 3

[6] (2009, May) LHCb web site. [Online]. Available: http://lhcb-public.web.cern.ch/lhcb-public/ 3

[7] R. Brun, P. Buncic, F. Carminati, A. Morsch, F. Rademakers, and K. Safarik, "Computing in alice," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 502, no. 2-3, pp. 339 – 346, 2003, proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research. [Online]. Available: http://www.sciencedirect.com/science/article/B6TJM-4805T3S-1/2/edefbd51675bf95d6d9f346cb56a3425 3

[8] P. Saiz, L. Aphecetche, P. Buncic, R. Piskac, J. E. Revsbech, and V. Sego,

Physics Research. [Online]. Available: http://www.sciencedirect.com/science/article/B6TJM-480C1TX-13/2/5eb76d3bf340c5df764790cf56e96c38 3

[9] A. Fanfani, J. Andreeva, A. Anjum, T. Barrass, D. Bonacorsi, J. Bunn, M. Corvo, N. Darmenov, N. De Filippis, F. Donno, G. Donvito, G. Eulisse, F. Fanzago, A. Filine, C. Grandi, J. M. Hernndez, V. Innocente, A. Jan, S. Lacaprara, I. Legrand, S. Metson, H. Newman, A. Pierro, L. Silvestris, C. Steenberg, H. Stockinger, L. Taylor, M. Thomas, L. Tuura, F. Van Lingen, and T. Wildish, "Use of grid tools to support cms distributed analysis," CERN, Geneva, Tech. Rep. CMS-CR-2004-057. CERN-CMS-CR-2004-057, Oct 2004. 3

[10] F. Berman, G. Fox, and A. J. G. Hey, *Grid Computing: Making the Global Infrastructure a Reality*. Wiley, 2003. 3, 17

[11] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Jounral of Supercomputer Applications*, vol. 15, no. 3, 2001. [Online]. Available: http://citeseer.ist.psu.edu/foster01anatomy.html 3

[12] P. Z. Kolano, "Facilitating the portability of user applications in grid environments," in *DAIS*, 2003, pp. 73–85. 4

[13] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, "Grid information services for distributed resource sharing," 2001. [Online]. Available: http://citeseer.ist.psu.edu/czajkowski01grid.html 4

[14] I. F. Carl, C. Kesselman, G. Tsudik, and S. Tuecke, "A security architecture for computational grids," 1998. 4, 20

[15] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman, "A multi-policy authorization framework for grid security," *Network Computing and Applications, IEEE International Symposium on*, vol. 0, pp. 269–272, 2006. 4, 20

[16] J. Geelan. (2009, May) Twenty one experts define cloud computing. [Online]. Available: http://www.sys-con.com/node/612375 5

[17] K. A. Delic and M. A. Walker, "Emergence of the academic computing cloud," *ACM Ubiquity*, vol. 9, no. 31, August 2008. [Online]. Available: http://www.acm.org/ubiquity/volume_9/v9i31_delic.html 5

[18] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *Proceedings of Cloud Computing and Its Applications*, October 2008. [Online]. Available: http://eucalyptus.cs.ucsb.edu/wiki/Presentations 5

[19] (2009, May) Amazon Elastic Compute Cloud (Amazon EC2). [Online]. Available: http://aws.amazon.com/ec2/ 5, 6

[20] (2009, May) Google App Engine. [Online]. Available: http://code.google.com/appengine/ 6

[21] (2009, May) Google Docs. [Online]. Available: http://docs.google.com/ 6

[22] A. Tsalgatidou and T. Pilioura, "An overview of standards and related technology in web services," *Distrib. Parallel Databases*, vol. 12, no. 2-3, pp. 135–162, 2002. 7

[23] K. Channabasavaiah, K. Holley, and J. Edward M. Tuggle, "Migrating to a service-oriented architecture," December 2003. [Online]. Available: http://www.ibm.com/developerworks/library/ws-migratesoa/ 7

[24] H. He, "What is service-oriented architecture," September 2003. [Online]. Available: http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html 7

[25] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, and D. O. C. Ferris, "Web Services Architecture," February 2004. [Online]. Available: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/ 9

[26] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," April 2007. [Online]. Available: http://www.w3.org/TR/soap12-part1/ 9, 10

[27] A. Akram, R. Allan, and D. 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. [Online]. Available: http://pubs.doc.ic.ac.uk/web-services-binding-data-styles/ 9

[28] D. C. Fallside and P. Walmsley, "Xml schema part 0: Primer second edition," October 2004. [Online]. Available: http://www.w3.org/TR/xmlschema-0/ 10

[29] 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 11

[30] D. Hollingsworth, "Workflow management coalition the workflow reference model," January 1995. [Online]. Available: http://www.wfmc.org/standards/docs/tc003v11.pdf 11

[31] H. Li, Y. Yang, and T. Y. Chen, "Resource constraints analysis of workflow specifications," *J. Syst. Softw.*, vol. 73, no. 2, pp. 271–285, 2004. 12

[32] A. Barker and J. van Hemert, "Scientific workflow: A survey and research directions," in *Proceedings of the Third Grid Applications and Middleware Workshop (GAMW'2007)*, ser. LNCS, 2007, p. In press. 12

[33] C. Pelz, "Web Services Orchestration and Choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003. 12

[34] (2009, May) Web Service Choreography Interface (WSCI) 1.0. [Online]. Available: http://www.w3.org/TR/wsci/ 12

[35] M. A. Vouk, "Cloud computing: Issues, research and implementations," in *Information Technology Interfaces, 2008. ITI 2008. 30th International Conference on*, 2008, pp. 31–40. [Online]. Available: http://dx.doi.org/10.1109/ITI.2008.4588381 12

[36] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid services for distributed system integration," *Computer*, vol. 35, no. 6, pp. 37–46, 2002. 12

[37] N. J. Brauer, B., "Extending Your SOA and Internal Applications Utilizing Commercial Web Services: Integrating Externally Available On-Demand Data and Functionality for Greater Productivity." [Online]. Available: http://www.strikeiron.com 13

[38] J. R. Hill, "A management platform for commercial web services," *BT Technology Journal*, vol. 22, no. 1, pp. 52–62, 2004. 13

[39] S. Krishnan, K. K. Baldridge, J. P. Greenberg, B. Stearn, and K. Bhatia, "An end-to-end web services-based infrastructure for biomedical applications," in *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 77–84. 13

[40] G. K. Slominski, D. Gannon, S. Krishnan, A. Slominski, and L. Fang, "Building applications from a web service based component architecture," 2005. 13

[41] S. Miles, P. Groth, M. Branco, and L. Moreau, "The requirements of using provenance in e-science experiments," *Journal of Grid Computing*, 2006. [Online]. Available: http://eprints.ecs.soton.ac.uk/13242/ 13

[42] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, September 2005. [Online]. Available: http://dx.doi.org/10.1145/1084805.1084812 13

[43] S. Vazhkudai, S. Tuecke, and I. Foster, "Replica selection in the globus data grid." IEEE Computer Society Press, 2001, pp. 106–113. 13

[44] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23, no. 3, pp. 187 – 200, 2000. [Online]. Available: http://www.sciencedirect.com/science/article/B6WKB-45F4XDK-5/2/977c18d8d008c500ebd0414d5f9d6a27 13

[45] R. van Engelen, "Pushing the soap envelope with web services for scientific computing," in *ICWS*, 2003. 13

[46] (2009, May) Three web services recommendations. [Online]. Available: http://www.w3.org/2005/01/xmlp-pressrelease.html 13

[47] G. Fox, G. Aydin, H. Bulut, H. Gadgil, S. Pallickara, M. Pierce, and W. Wu, "Management of real-time streaming data grid services: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 7, pp. 983–998, 2007. 14, 15, 21

[48] S. Koulouzis, E. Meij, M. S. Marshall, and A. Belloum, "Enabling data transport between web services through alternative protocols and streaming," *eScience, IEEE International Conference on e-Science*, vol. 0, pp. 400–401, 2008. 14

[49] J. D. Blower, A. B. Harrison, and K. Haines, "Styx grid services: Lightweight middleware for efficient scientific workflows," *Sci. Program.*, vol. 14, no. 3,4, pp. 209–216, 2006. 14

[50] R. Pike and D. M. Ritchie, "The styx architecture for distributed systems," *Bell Labs Technical Journal*, vol. 4, no. 2, pp. 146–152, June 1999. [Online]. Available: http://www.vitanuova.com/inferno/papers/styx.html 14

[51] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from bell labs," in *In Proceedings of the Summer 1990 UKUUG Conference*, 1990, pp. 1–9. 14

[52] "OASIS Web Services Resource Framework (WSRF)," April 2006. [Online]. Available: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf 14

[53] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services." *Nucleic Acids Res*, vol. 34, no. Web Server issue, July 2006. [Online]. Available: http://dx.doi.org/10.1093/nar/gkl320 14

[54] "Webservice data proxy," May 2009. [Online]. Available: http://www.cs.man.ac.uk/~sowen/data-proxy/guide.html 15, 21

[55] A. Barker, J. B. Weissman, and J. van Hemert, "Orchestrating data-centric workflows," in *CCGRID*, 2008, pp. 210–217. [Online]. Available: http://www.adambarker.org/techreport_barker.pdf 15, 21

[56] S. Heinzl, M. Mathes, T. Friese, M. Smith, and B. Freisleben, "Flex-swa: Flexible exchange of binary data based on soap messages with attachments," *Web Services, IEEE International Conference on*, vol. 0, pp. 3–10, 2006. 15

[57] "SOAP Messages with Attachments," May 2009. [Online]. Available: http://www.w3.org/TR/SOAP-attachments 15

[58] (2009, May) SDSC website. [Online]. Available: http://www.sdsc.edu/srb/index.php/What_is_the_SRB 19

[59] P. Grosso, D. Marchal, J. Maassen, E. Bernier, L. Xu, and C. de Laat, "Dynamic photonic lightpaths in the starplane network," *Future Gener. Comput. Syst.*, vol. 25, no. 2, pp. 132–136, 2009. 19

[60] G. van 't Noordende, S. D. Olabarriaga, M. R. Koot, and C. T. A. M. de Laat, "A trusted data storage infrastructure for grid-based medical applications." in *CCGRID*. IEEE Computer Society, 2008, pp. 627–632. [Online]. Available: http://dblp.uni-trier.de/db/conf/ccgrid/ccgrid2008.html 19

[61] J.-P. Baud, J. Casey, S. Lemaitre, and C. Nicholson, "Performance analysis of a file catalog for the lhc computing grid," *High-Performance Distributed Computing, International Symposium on*, vol. 0, pp. 91–99, 2005. 20

[62] J. Bresnahan, M. Link, G. Khanna, Z. Imani, R. Kettimuthu, and I. Foster, "Globus GridFTP: what's new in 2007," in *GridNets '07*, 2007. 20

[63] (2009, May) VBrowser web site. [Online]. Available: http://www.vl-e.nl/vbrowser 23

[64] (2009, May) Apache Axis web site. [Online]. Available: http://ws.apache.org/axis/java/index.html 24

[65] (2009, May) VL-e web site. [Online]. Available: http://www.vl-e.nl 41

[66] May 2009. [Online]. Available: http://medline.cos.com/ 41

[67] Nadeau, David, Sekine, and Satoshi, "A survey of named entity recognition and classification," *Linguisticae Investigationes*, vol. 30, no. 1, pp. 3–26, January 2007. [Online]. Available: http://www.ingentaconnect.com/content/jbp/li/2007/00000030/00000001/art00002 46

[68] M. Roos, M. S. Marshall, A. Gibson, and P. W. Adriaans, "Structuring mined knowledge for the support of hypothesis generation in molecular biology," *Semantic Web Applications and Tools for Life Sciences*, 2008. 46

[69] E. Zudilova-Seinstra, N. Yang, L. Axner, A. Wibisono, and D. Vasunin, "Service-oriented visualization applied to medical data analysis," *Service Oriented Computing and Applications*. [Online]. Available: http://dx.doi.org/10.1007/s11761-008-0031-6 52, 55

[70] (2009, May) VTK website. [Online]. Available: http://www.vtk.org 53