University of Southampton
Faculty of Engineering and Applied Science
Department of Electronics and Computer Science

A project report submitted for the award of
Software Engineering

Project supervisor: Zaluska, E. J.
Second examiner: Nunn, Dr. D.

Scientific Data Access and Integration into the Grid using Grid Technology

by Anthony Relle
08/05/2003

Image courtesy of http://web.datagrid.cnr.it/

Abstract

The Grid is a multinational effort to develop systems which will enable distributed computing on a global scale. Standards are currently being formalised to create a globally interoperable network built on a common architecture. There are currently no standards for providing access to databases through interfaces on the Grid. As organisations adapt Grid technology, interfaces for data access and their integration into the Grid must be made.

This project developed an existing Open Grid Services Architecture (OGSA) component produced under the E-science programme by the University of Edinburgh and IBM to expand on the ability to access data over the Grid. The ability to deliver data using new mechanisms was added, along with access to Binary Data, through the redevelopment of BinX. Mechanisms for security were introduced to the results obtainable from this OGSA component, adding integrity, authentication, and privacy to data accessed over the Grid.

It was found there are many ways to implement these mechanisms, as many different standards are being created for Grid interfaces. The BinX implementation required concessions to be made that did not fit into a fully networked model. Implementation of security found further work would be required as new security standards are produced.

Table of Contents

1 Introduction

1.1 The Grid

The Grid is a multinational effort of using distributed computing on a global scale. It is designed around the concept of a defined set of interfaces for sharing computer resources across a large scale interlinked network. It is on this architecture that Grid Computing is based.

The Grid currently exists as a large network between many organisations, with many research based such as CERN and PPARC. Each community that is linked into the grid is known as a *Virtual Organisation*. These organisations can share resources such as data and computational time, in order to achieve common goals. Grid technologies are the key in connecting each VO over the Grid, providing a secure and automated way of sharing these resources.

A successful implementation of the Grid is one where a network is provided where a user submitted request, such as a query to retrieve a set of scientific data from a known location, is allocated suitable resources by the Grid itself. Request processing is then monitored, and notification is given on results being returned for the request. The choice of what systems or storage in use do not have to be made with the request, the Grid can deal with this, allowing users to concentrate on what they want done rather than how they do it.

Many groups are currently working to standardise the interfaces of the Grid, breaking the Grid down into components and working towards standards for each. The architecture of these components provide a global computing infrastructure, allowing the large scale applications aimed for by the Grid, thus described in the paper 'Physiology of the Grid' [1].

Research groups are currently working on the integration of data access over the grid, with many of these co-ordinated by the European Union funded DataGrid Project. The DataGrid Project aims to enhance scientific research by using distributed data access along with computation and analysis across a wide range of scientific communities. In 2002 the EU DataGrid project showed success by demonstrating the first datagrid testbed. Their demonstration involved computational jobs submitted to the Grid and their return successfully to an audience of 60 scientists.

This project deals with Data Access and Integration into the grid, based on the OGSA-DAI Release 1 package, developed by IBM and the University of Edinburgh. The OGSA-DAI package is designed for providing a standard interface over the Grid, using Grid technologies, for data access.

With Grid infrastructure in place and having been demonstrated as successful, the need for standards of data access and integration into the Grid are required. For the purpose of creating standards in this project, first the scenarios of usage of data over the grid will be examined, and then compared to the current standards available in Grid Computing. By using the combination of the scenarios of use and the available standards, an implementation of scientific data access and integration into the Grid is designed and created.

For the design of OGSA-DAI, many scenarios must be catered for [4] before it can be adopted for use. OGSA-DAI is a Grid Data Service, which is the service that provides the interface for the data access, returning results in a standard way using XML. The Grid Data Service is responsible for handling data requests on the grid. XML is used as it a recognised standard that can easily be adapted for the purpose of representing data in a predefined format.

The OGSA-DAI is being developed for the Open Grid Services Architecture, which uses the Open Grid Services Infrastructure, the framework of the Grid. First the structure of the Grid architecture, OGSI is explained in Chapter 1.2 , followed by the implementation of Grid Services, OGSA in Chapter 1.3 .

1.2 Infrastructure of the Grid

In order to have a globally interoperable grid network, a set of conventions have been drafted to attempt to standardise the services that run on the grid. The Open Grid Services Infrastructure, OGSI, is the framework on which Grid Services can be used. It does not define what they do and how they do it, but rather conventions for lifetime management of these services, their discovery, creation, and other attributes they may have. The implementation is left open ended, only standard interfaces to these services are specified.

A worldwide collaboration of over 400 organisations participate in regular meetings with attendance from all over the world, in an effort to produce specifications for services on the grid with one global standard. Documents are drafted on all aspects of services for the grid, ranging from the core framework of the OGSI to aspects such as grid computing, or how services will handle access to backend databases. This paper is concerned with

Database Access and Integration Service (DAIS) access on the grid. The OGSI provides the framework for the 'Grid Services' to be accessed on. Currently, a Grid Service is defined by WSDL, the Web Service Description Language. WSDL is a language of definition only; it uses XML to define information about a Web Service.

Currently, WSDL allows several properties to be defined for a Grid Service. Summed up briefly, the ones listed below are those defined in the OGSI draft document [2].

- *Service Data* – The standard method for representing and querying metadata and state data from a Grid Service instance.
- *Grid Service Description* and *Grid Service Instance* – a description of each Grid Service for their extension and use.
- *Grid Service Handle* and *Grid Service Reference* properties, which are used to refer to instances of the Grid Service.
- Definition of a common approach for returning fault information from operations.
- Definition of the lifecycle existence of a Grid Service.

There are also extensions to WSDL 1.1, involving portTypes for the definition of interfaces; however this paper focuses on the implementation of a Grid Service, and not these underlying Grid Service definitions. An example WSDL document is shown in Appendix 8.1 and contains an example WSDL definition of a web service.

1.3 Services on the Grid

The OGSI is the base for the Open Grid Services Architecture, concerning the actual implementation of the Grid Services. 'Physiology of the Grid' [1] describes the grid as being a collection of services provided by 'virtual organisations'. A Grid Service provided by a virtual organisation would be shared to others over the grid network, and provide the ability for others to use their resources. The main aim of the grid is full interoperability of the services it provides. For the definition of the interfaces, section 1.2 contains information on OGSI, currently built on the existing web standard of WSDL.

The OGSA makes standard definitions for these services that run on the grid, a Grid Service, as 'a Web service that provides a set of well-defined interfaces and that follows specific conventions' [1]. The definition caters for interfaces such as address discovery, dynamic service creation, lifetime management, etc. as standard for each Grid Service. Table 1 from 'Physiology of the Grid'

[1] shows a proposed table of interfaces for an OGSA Grid Service. The operations shown cater for discovery of data, termination of the service, notification for events such as errors, registration and creation of a Grid Service.

PortType	Operation	Description
GridService	FindServiceData	Query a variety of information about the Grid service instance, including basic introspection information (handle, reference, primary key, home handleMap: terms to be defined), richer per-interface information, and service-specific information (e.g., service instances known to a registry). Extensible support for various query languages.
	SetTerminationTime	Set (and get) termination time for Grid service instance
	Destroy	Terminate Grid service instance
Notification-Source	SubscribeTo-NotificationTopic	Subscribe to notifications of service-related events, based on message type and interest statement. Allows for delivery via third party messaging services.
Notification-Sink	DeliverNotification	Carry out asynchronous delivery of notification messages
Registry	RegisterService	Conduct soft-state registration of Grid service handles
	UnregisterService	Deregister a Grid service handle
Factory	CreateService	Create new Grid service instance
HandleMap	FindByHandle	Return Grid Service Reference currently associated with supplied Grid Service Handle

Table 1

The actual implementation of a Grid Service is left up to the Hosting Environment. Any language such as Java, C, .NET may be used. One implementation of a Grid Service for access to relational databases is Spitfire [6], implemented as a Java Servlet through AXIS, developed as part of the European DataGrid project. SpitFire provides uniform access to databases for Java/C++/SOAP enabled frontends. By defining its own standard of ways of accessing databases, the SpitFire code can handle the access to different databases through a single interface, and thereby be able to be defined as a Grid Service. It is currently in development, as are many other Grid Services.

The actual implementation of a Grid Service is described later, in section 2.1 .

1.4 Scientific Data and the Grid

There are many requirements for data access over the grid, resulting in many possibilities of how to integrate the data interface into Grid Services. One example of a scientific application of databases being accessed over the grid is AstroGrid. The AstroGrid project was proposed in April 2001 [13], with a submission for a system integrating multiple databases into a collaboration using Grid technology.

Astronomy research can result in Gbytes of data produced daily as measurements and analyses are taken of data recorded from Astronomical sources. There are many different laboratories around the world recording using various different methods, resulting in many different databases, in many different locations, with these huge volumes of data. Integrating all this data together is a problem. The AstroGrid proposal involves using Grid Technology to bring all the databases together, to make it easier to query, analyse, and explore all the available data, to help and accelerate the advancement of scientific knowledge through collaboration.

This project takes this requirement for scientific data to be implemented into the Grid, along with other usage scenarios identified for data access and integration. The next section deals with the background research performed on a prototype designed for providing an interface to databases over the Grid.

2 Background Research

At the beginning of this project, the IBM Matrix prototype delivery system was a publicly available system with a sample implementation of Data Access and Integration into the grid, using Java and JDBC to access a relational database. Before providing interfaces to integrate data into the grid, the Matrix prototype and its aims were examined.

There are many areas to consider when deciding on an interface for data for integration into the Grid. Papers such as "Grid Database Access and Integration: Requirements and Functionalities" [8] are produced by Working Groups on Grid Database Access and Integration, and list requirements and functionalities required for a grid implementation of a Grid Service for Data Access. The paper describes areas that must be considered when defining the interface to such a Grid Database Access and Integration service, as listed below

- Publishing and discovery of data sources
- Statements, how to query the data
- Structured Data Transport, how to return the data
- Data Transformation, operations on the data
- Transactions, robustness of operations
- Authentication, Access control and Accounting
- Metadata, to provide more descriptive information
- Management, to control the operations and performance
- Data Replication
- Connections and Sessions
- Integration

For the initial research of this project the Matrix Prototype was used as a base for any existing code to be added to. The Matrix Prototype implemented only some of the interfaces in the above list. For example, there is no way to use the initial release and control transactions, or to maintain persistent connections, which are employed in normal usage of a database. Although this release was limited in these ways, it was used as the basis of background research with alteration for this project.

Before the implementation of the Matrix prototype is described, first the design of a Grid Service is examined.

2.1 Grid Services

A Grid Service is created for the purpose of taking queries from other services or users of the Grid. There are standard ways of defining the interface to a Grid Service which are described in the introduction section 1.3 . These mostly involve defining a method that is used to perform an action. Through the use of these methods it is possible to implement data queries and data returns.

To actually instantiate a Grid Service, OGSA defines the *Factory*, *Registry*, *GridService*, and *HandleMap* interfaces for Grid Service objects. The standard approach to create a new instance of a Grid Service is shown below in Figure 1.

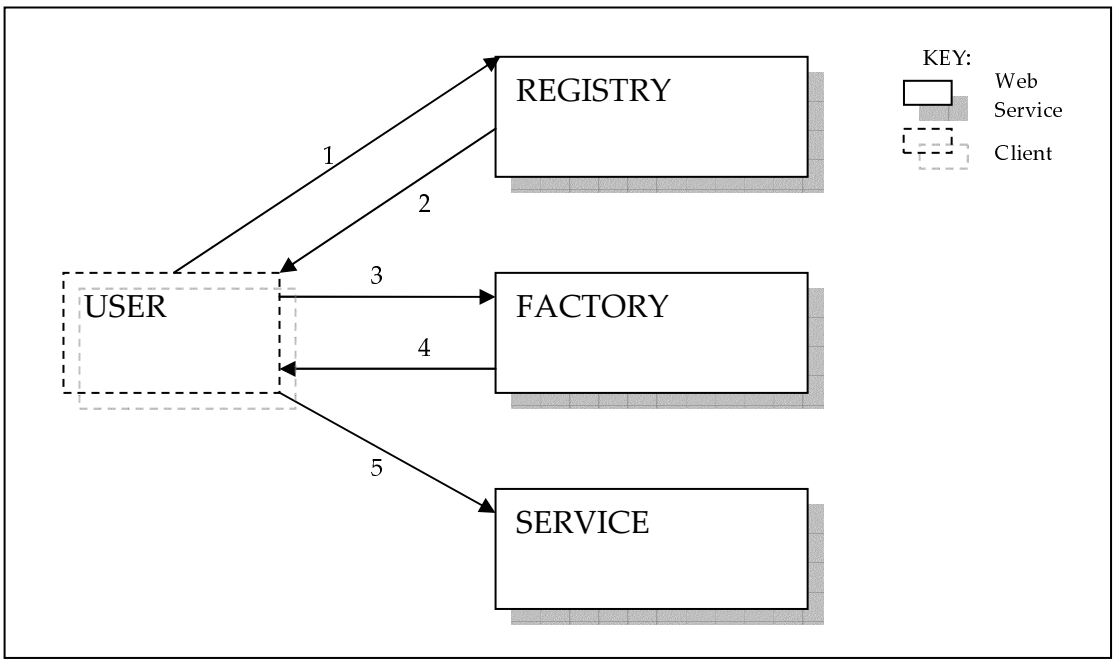

Figure 1

In a simple hosting environment, a *Registry* is used to store the availability of any factories available for the creation of Grid Services. A *Registry* can store multiple factories, for creating different kinds of Grid Services. As shown in the above diagram, clients can discover a *Factory*, and call them using OGSA interfaces to create a new instance of a *GridService*. On creation, the instance can be assigned an identifier and make that available to *HandleMap* service. The client calling for a Grid Service can now access this service directly and perform queries to it. This is a standard mechanism of discovery and creation of Grid Services in the OGSA.

2.2 Initial prototype Matrix of a Grid Data Service for Database Access

The Matrix package is a prototype for an OGSA Grid Data Service, written in the Java programming language. The Matrix implements Grid Data Access to a Relational Database Management System. It is designed to provide the necessary functionality for access to a RDBMS, using the OGSA interfaces as a Grid Service, and was designed around the database use scenarios identified [8] for a data resource in the Grid. The Matrix delivery system was a prototype attempting to provide the interface for as many identified scenarios as possible, a few example ones are listed below in Table 2.

Patterns	Description
R1	Select with small number of rows each with a small volume of data returned directly to the calling process.
R2	Select from a database table that contains references to external information. This is often the case in scientific grid where the database tables hold meta data about images or other data sets held externally to the database. The external reference is returned as part of the resulting information.
R3	Select from a database table that contains a blob. This is similar to R2 but in this case the referenced data set is managed by the RDBMS. The reference to the blob is returned as part of the resulting information.
R4	Select with a large number of rows each with a small volume of data returned directly to the calling process.

Table 2

In order to implement the Matrix prototype as a Web Service, many scenarios like these have to be catered for in one Grid Service interface. The method a Grid Service would use to handle scenario 'R1' from Table 2 is shown following in Figure 2. This scenario involves the Matrix Service to be discovered, created, sent a query to return a small number of rows, then data returned.

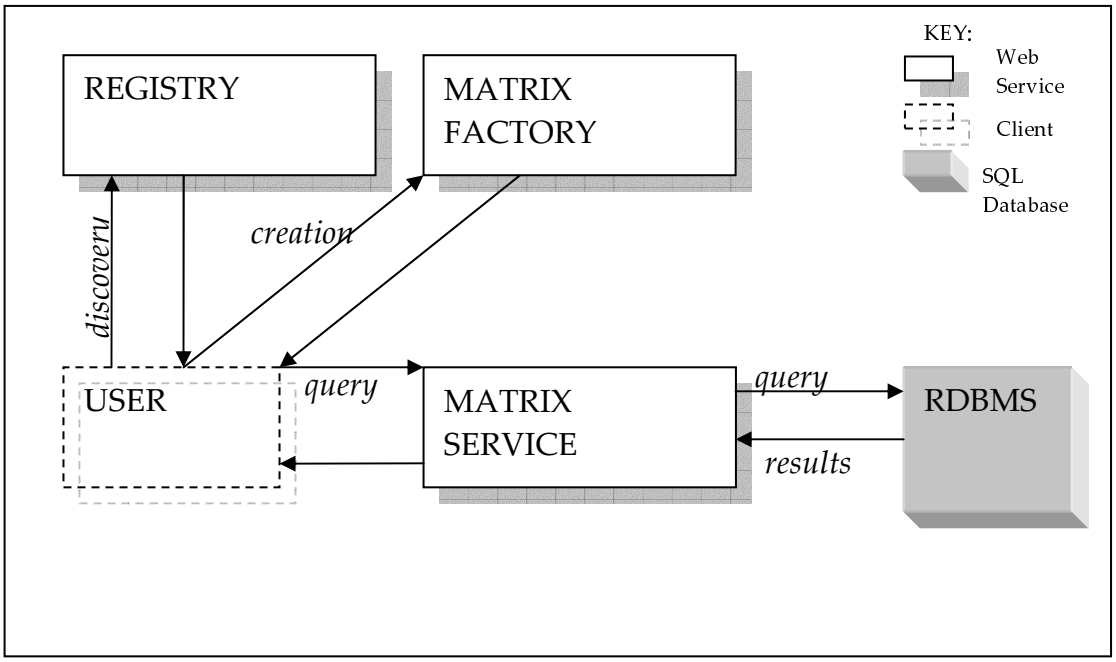

Figure 2

The diagram shows the discovery of a factory in order to create the Matrix service, and a query being sent to the service on an instance being found by the user. The Matrix service itself handles querying to the RDBMS, through configuration controlled server side. The actual Java implementation of this system is covered in 2.4 covering the background research on the Matrix prototype.

2.3 Benchmarking

As shown above, the implementation requires many front end actions before a database lookup actually occurs. Without the direct connection to the database, the process could be very slow, depending on the implementation. Before any alterations were made to the IBM Matrix code, benchmarking was performed to see how much overhead end users of a Grid Service would experience, and how that would compare to say, a direct SQL lookup to a relational database.

For the purposes of this benchmarking, a sample table with rows of contacts info was used, such as surnames, initials, addresses, and phone numbers. The sum of the characters in the fields of each record was roughly 200 bytes each. The aim was to see how much overhead would occur for the retrieval of varying amounts of records, and how the transformation of the records to XML would slow down the process.

The conditions to find each result were as follows. The java interface MatrixServiceClient was used to perform a query on the database, returning a set number of results each run. For each different test, the number of records returned was increased. Initially 0 results were returned, then 20 results and higher were used, so a comparison of speeds could be found for different amounts of data the MatrixService was processing. Each test was run until 5 consecutive results were within 20ms of each other. The time recorded was from just before the query was sent to the MatrixService, to when the completed query results were available. The times did not include the constant overhead required to locate and create a MatrixService. The test assumes an instance of the MatrixService object has already been located.

Figure 3 below shows the query processing time plotted on a graph against the number of results returned in the query.

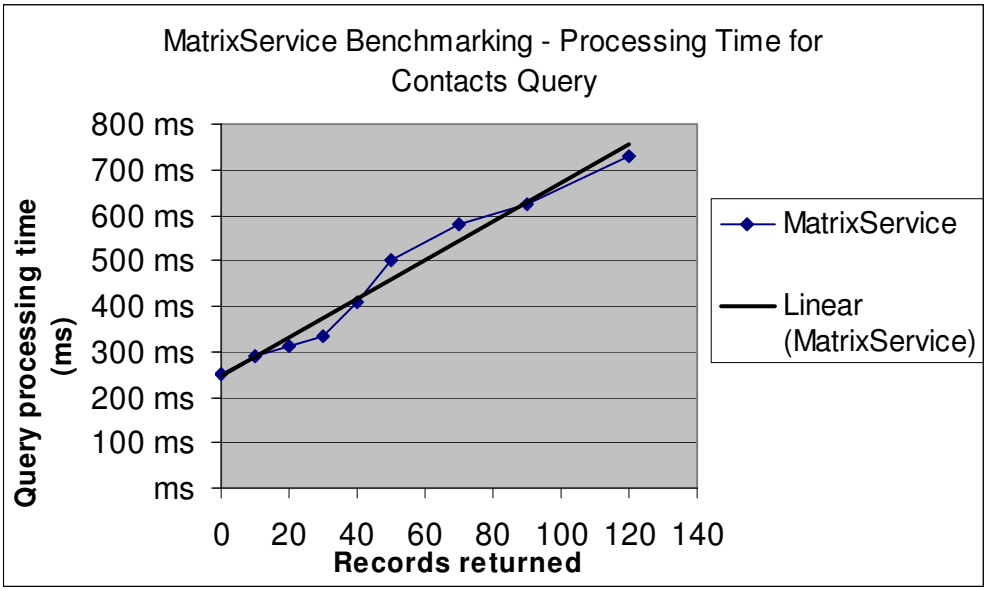

Figure 3

Appendix 8.3 contains the table of results that were used to build Figure 3. Using these figures it is possible to calculate the increase of query processing time per number of records returned in the query. The linear line shown on the figure above is calculated in this way.

For the situation described in this Benchmark, each extra record resulted in an additional 4ms processing time by MatrixService.

For a direct connection to the SQL database, returning results directly and without XML transformation, the benchmark results found were 10ms for return all 120 records of the database returned.

In conclusion of the benchmarking, this extra 4ms processing time is minimal for the scenario identified as R1 in Table 2, where only a small number of records are returned from a Grid Data Service query. However scenario R4 involves a large volume of rows, where the delay would add up considerably if a query involving, say, 1000 records was performed.

2.4 Matrix prototype of Web Service

The integration of a Grid Service into the other existing components is a fairly important part of the design stage. For this purpose, I adapted the original prototype for use with multiple databases simultaneously, and also for different databases such as mySQL and PostgreSQL. At this stage I found the JDBC connections in Java to support multiple databases like this seamlessly, making Matrix independent of the actual RDBMS implementation.

For the actual design of the prototype, the documentation provided with the Matrix initial release prototype [10] includes the following diagram, Figure 4

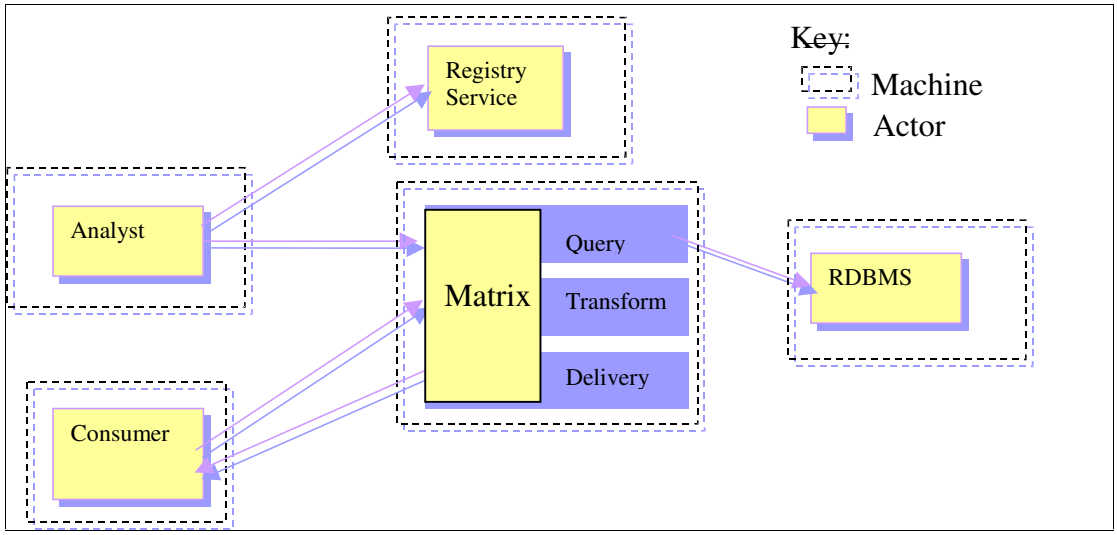

Figure 4

This model was the basis for the prototype, containing the actors *Analyst*, *Consumer*, and *Matrix*. The scenarios for this model involve the Analyst interacting with Matrix to perform a query resulting in a set of data. The Analyst can then deliver the results to a Consumer for processing. Using this model as the basis of the Grid Service, it is possible to build on the interfaces of the three main functions of the MatrixService, the Query, Transform and Delivery capabilities.

- Query – Execution of queries by the RDBMS

- Transform – Converting the data returned by a query. This could involve data compression or encryption.
- Delivery – How the data is returned, and who it is returned/available to.

The Matrix prototype was implemented using only standards available on the web today. The implementation involved WSDL to deploy Web Services onto a Java Application Server. For this purpose, Tomcat was used as the Application Server.

2.4.1 Interfaces

Four web services are used, *MatrixDeliveryService*, *MatrixDeliveryServiceFactory*, *MatrixService* and *MatrixServiceFactory*. These are defined using WSDL, so methods are available which can be messaged, and give a return such as a MatrixService. These are the services that are used to operate the MatrixService.

The factories are used to create the corresponding Web Service, with *MatrixService* handling the Query, and *MatrixDeliveryService* handling the delivery. In the prototype implementation, there is no Registry or Transform functions other than the consistent conversion of query results into XML. The process used by this prototype to discover the Matrix Service and perform a query was as follows

locate MatrixServiceFactory at default location
create MatrixService object
send QTD with query and delivery info

For the Matrix Prototype, the QTD provides all information about the query to the MatrixService, such as the query required, security information, how to return the data. For the purposes of the background research, the QTD format is not fully examined, but a QTD for performing an example query can be found at Appendix 8.2 . It is formed in XML with set fields for each entry in the query to the MatrixService.

2.4.2 Query

The Matrix prototype architecture has a UML diagram [10] for the core java classes of the query service it provides, shown following as Figure 5.

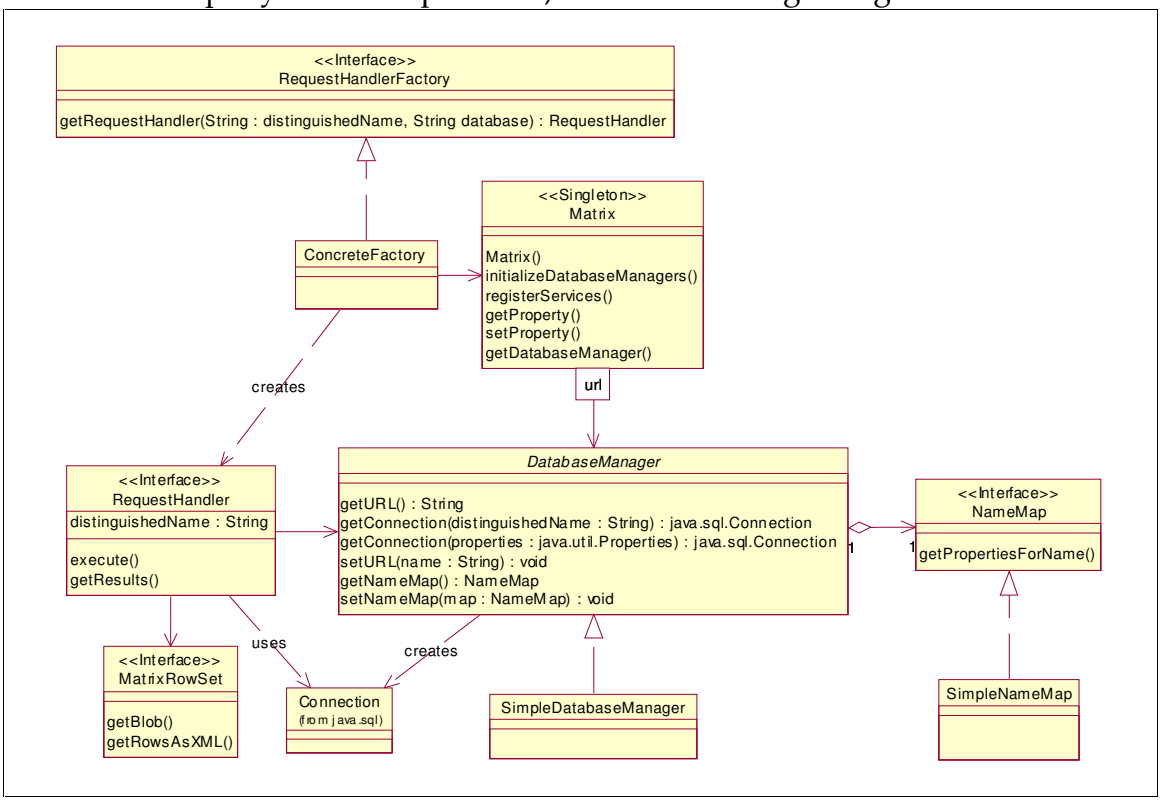

Figure 5

In the prototype implementation, each *DatabaseManager* manages a connection to a RDBMS. Requests are handled by a *RequestHandler* object. A *RequestHandler* is created by a *RequestHandlerFactory*. The action of these classes and which are used depend on the contents of the QTD request document.

2.4.3 Delivery

Also available on the prototype architecture is Figure 6, the UML diagram for the DeliveryService [10].

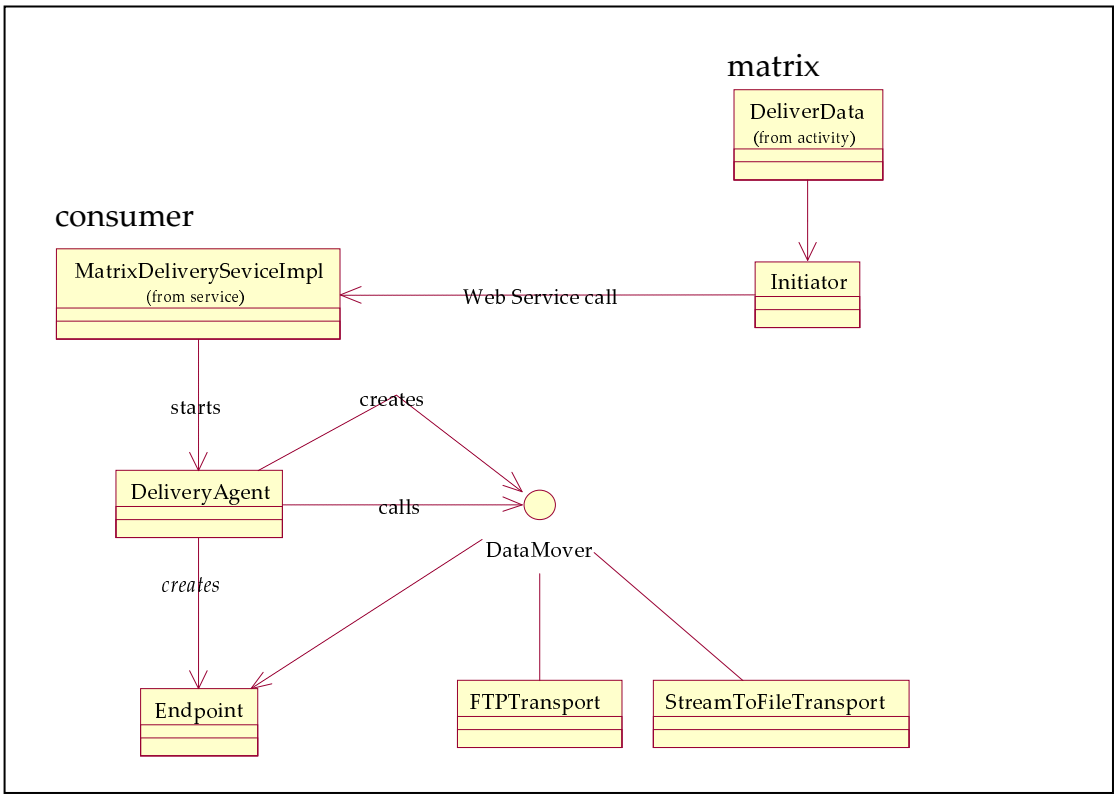

Figure 6

In this prototype, the receiving point of data from the Matrix Service was implemented as a *MatrixDeliveryService* class. The activity involves locating the *MatrixDeliveryService* from the original Matrix Web Service, by using information passed in the QTD. The Delivery Order contains information such as the delivery endpoint, QoS and any user certificates. The *DeliveryAgent* can then use this Delivery Order and initiate the delivery. FTP or StreamToFile can be loaded and used as the delivery mechanism for the query results.

3 Design

3.1 Overview of OGSA-DAI

As the basis of my Grid Service, code for the Data Access and Integration service was used from Release 1 of the OGSA-DAI software (v0.3) available from http://www.ogsa-dai.org.uk/.

Using this release of OGSA DAI, the primary component available from the package is the GridDataService object (GDS), the main Grid Service for handling requests. This provides the data access and data integration services. Release 1 of the OGSA-DAI is a step above the Matrix prototype in that it provides integration with the OGSA service model. Also available is support for heterogeneous data sources, so different RDBMS systems can be used. For the purpose of the OGSA-DAI service, a document orientated system is used for data access and delivery.

Each GridDataService object in this design connects to a single database. In this java implementation, JDBC drivers are used to handle the connection to the RDBMS system. Each GridDataService instance is responsible for both querying and delivering data. The interface to GridDataService is through the method *performScript()*, defined in portTypes. This is done through the standard method of defining methods in the OGSA. Queries are made in XML documents, which contain the statement to be performed, and also delivery instructions. A sample GridDataService script can be found at Appendix 8.4 , performing a simple query. The format of the GDS script is described in section 3.3 . This version of OGSA-DAI required data that was being used to update a database, to be part of the request, and data that is retrieved as part of the request be returned with the response.

A diagram illustrating the systems and processes involved in the usage of OGSA-DAI is shown below, followed by an explanation.

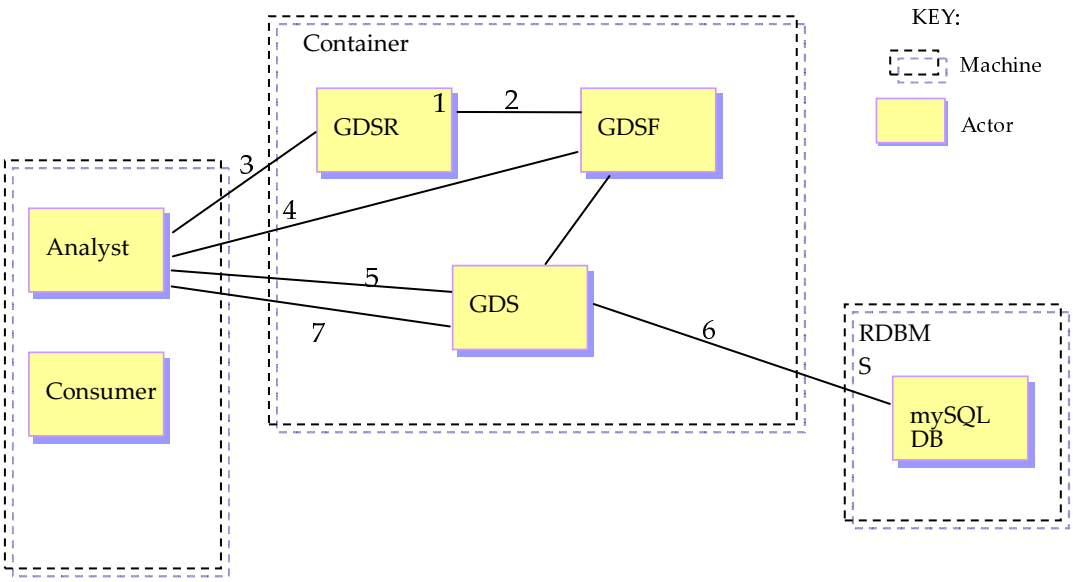

Figure 7

Figure 7 shows the operation of the GridDataService that results in a query being passed back to the original requester, the only possible recipient.

1. On execution of the GDS, the OGSI container creates the persistent registry service, GDSR (GridDataRegistryService).
2. The OGSI container creates the persistent GridDataServiceFactory, from configuration held in the server. The created GDSF is registered in the GDSR, and is configured for the access of a single Data Resource.
3. The system is setup, the Analyst actor can now query the Registry in order to search for a suitable Factory that can produce a GridDataService against the Data Resource it requires.
4. With a handle to the suitable GDSF, service data can be queried, and the createService() portType can be used. This results in a GridDataService being created.
5. Using the performScript() portType on the GDS, a XML document script can be submitted to the GDS for execution of a given query.
6. The GDS handles connections to the data resource, in this case using JDBS to connect to a mySQL database.
7. The results are returned to the Analyst, ending the request.

3.2 Delivery Mechanisms

Release 1 of the OGSA-DAI software does not implement any delivery mechanisms other than a synchronous data return with the request. The implementation returns results as XML, in the simplest scenarios XML RowSets are returned with XML metadata describing columns rows and data types. In order to make changes to the design that results in requests that are not returned as XML, such as data transformations, a new delivery mechanism is required. The change in design is shown in Figure 8, where there is another step added from Figure 7.

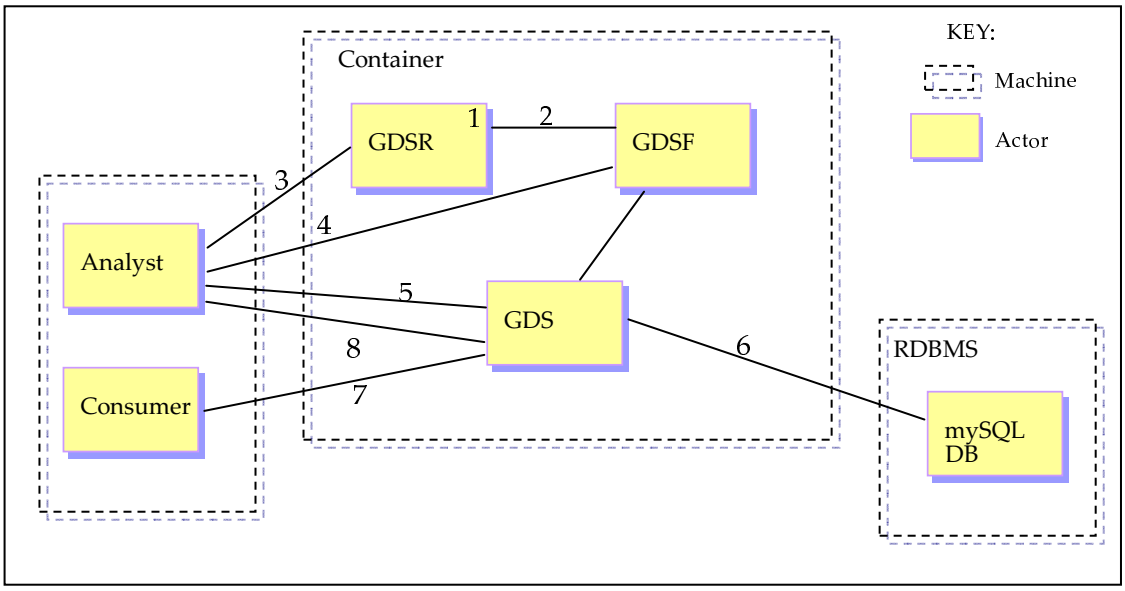

Figure 8

The request script containing the query is delivered as normal, step 5. The delivery mechanism is altered, the new steps 7 and 8 are shown next.

7. The GDS handles delivery of the data to a given location, through a given protocol.
8. On completion, the status of that delivery is returned at the end of the synchronous request in XML format, so the analyst will know the status of the delivery when the request is completed.

3.3 Grid Data Service Script

The Grid Data Service Script in OGSA-DAI contains the instructions of a query to be carried out by OGSA-DAI, including parameters such as the Data Resource to be used, a statement to be made, and the recipient of the delivery. It is written as a XML document, conforming to the XML schema shown in Appendix 8.5 . This defines a Header section of the document, and a Body

section. The Body section contains the query and the Delivery section, for the action required after the query is completed.

- The Header entry contains a ScriptName, Version entries, and Originator field.
- The Body entry contains the Statement and the Delivery entry
- The Delivery entry contains a From and To, along with unimplemented choices for Mode and Mechanism.

3.4 Scientific Applications

The AstroGrid project is introduced in section 1.4 , giving a requirement for large volumes of scientific data to be handled in a Grid Data Service. Before any design decisions are made, the following main cases were examined from the proposal [13] for the project.

- Growth of data and archives.
- Large Database science.
- Storage and data management requirements.
- Data intensive computing.
- The ability to browse simultaneously multiple datasets.
- Grid Technology.
- Remote analysis services.
- Collectivisation and the empowerment of the individual.
- A uniform archive query and data-mining software interface

The OGSA-DAI package satisfies many of these conditions, in that it provides a uniform query mechanism, based on Grid Technology. Large Database Science can be left up to the RDBMS servers rather than the interface, while data transformation could provide facilities for remote analysis.

One large problem faced by AstroGrid is the federation of different databases, which may be in many different formats, allowing for many different queries. Large astronomical databases can run into several TB size, such as the HST archive at 7 TB in 2001 [13] . As scientific data is often represented by large volumes of binary data, a different approach is required for OGSA-DAI to handle the data.

3.4.1 BinX - Binary XML Description Language

In order to integrate large amounts of scientific data into the Grid, a protocol is needed to be able to access blocks of scientific data, without requiring, say, a 2TB database to be loaded and transferred, a very impractical method of doing it. For that purpose, BinX, or Binary XML Description Language, is currently being developed [15]. BinX aims to provide the ability to use XML to describe the physical representation of arbitrary binary data files. This would provide a method of representing small volumes scientific data on the Grid, taken from an archive of a large amount of binary data. By using a standard way of describing binary data, BinX can be integrated into an interface for controlling Scientific Data available in OGSA-DAI, using existing standards of XML for data markup.

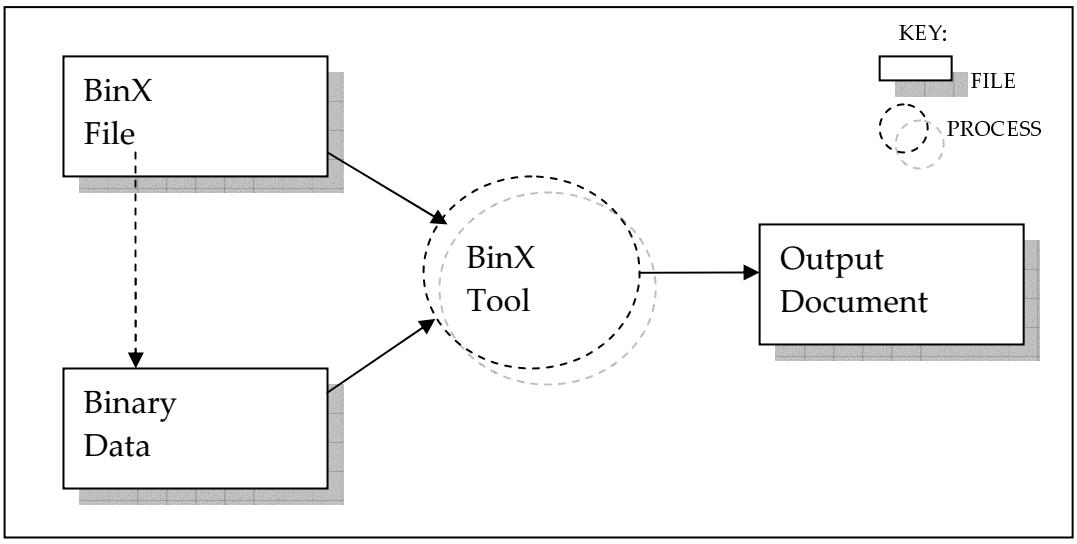

Figure 9

Figure 9 is based on a diagram from the BinX proposal [14], showing the use of a BinX descriptor file and a BinX tool to produce a document from binary data. The BinX File is a XML document, describing the structure and format of the binary data file, using a URI to point to that file. To provide platform independence and cross compatibility, BinX caters for different bit/byte ordering. The Binary data described can be of many primitive types, BinX includes support for IEEE floating point standard [16], allowing for the format of most scientific data.

BinX provides the ability to 'Slice' data from multidimensional arrays. This interface for slicing allows a subset of the entire collection of binary data to be returned, providing an opportunity for integration into OGSA-DAI. By returning just slices of data, it is possible to overcome the difficulties of

accessing small amounts of large multi-TB databases of scientific data over the grid.

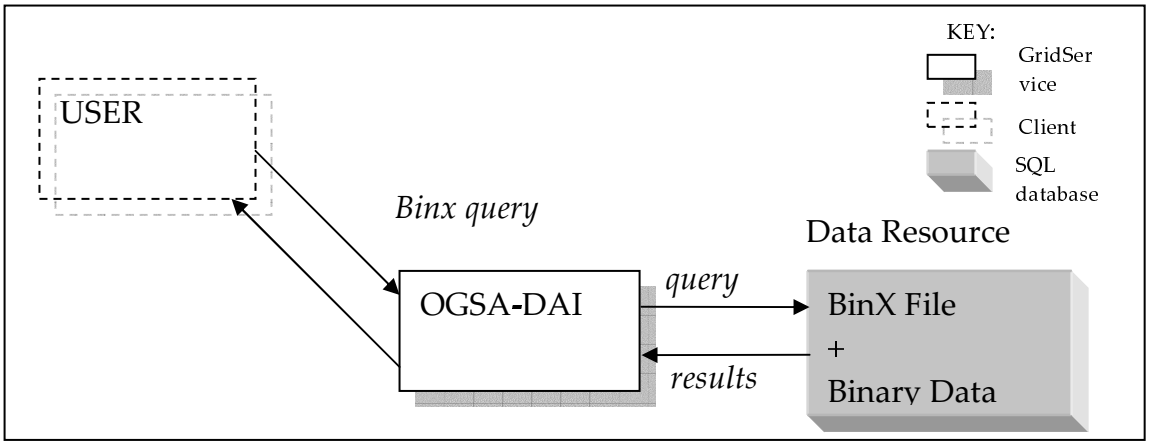

Figure 10

Figure 10 shows that the user of the system would pass a BinX query to OGSA-DAI. As a Grid Data Service, it handles the data resource containing the Binary Data and the BinX File. Using the query, the required subset of binary data can be retrieved from the Binary Data, and returned in a XML format as the results of the request.

The ability to update Binary Data proves a more complicated case, you cannot include binary data in XML, and so a binary set of data cannot be included in a request to OGSA-DAI. For small, incremental updates, it is possible to include the values of the binary data in the XML request to OGSA-DAI. A similar process to querying data is used in Figure 10, compared to the update process in Figure 11.

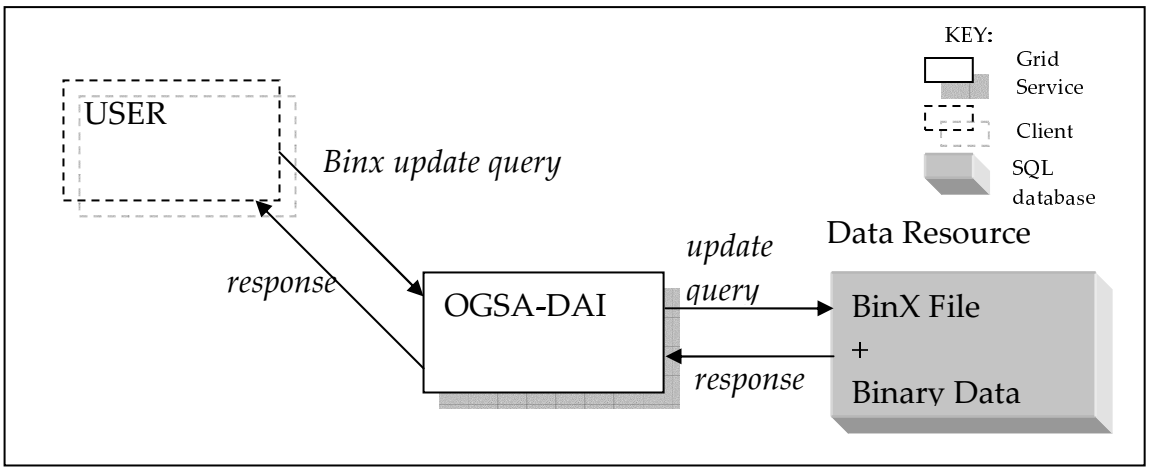

Figure 11

For the actual implementation of BinX, a BinX file can reference data by allowing complex structures to be described in XML format, for example a Bitmap Image file can be represented, using arrays, custom data structures, and different data types. Structures such as a file header containing a few data values can be specified in the following way:

```
<struct>
  <integer-32 varName="imageHeaderSize"/>
  <integer-32 varName="imageWidth"/>
  <integer-32 varName="imageHeight"/>
  <short-16 varName="bitsPerPixel"/>
</struct>
```

A basic data structure, with 4 arrays of *short* data values, would be represented by the following BinX file.

```
<?xml version="1.0" encoding="UTF-8"?>
<bx:array xmlns:bx="http://www.epcc.ed.ac.uk"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.epcc.ed.ac.uk binx.xsd"
 src="test.bin"
 offset="16"
 byteOrder="bigEndian"
 binType="short">
  <bx:dim name="w" indexFrom="0" indexTo="4"/>
  <bx:dim name="x" indexFrom="0" indexTo="4"/>
  <bx:dim name="y" indexFrom="0" indexTo="19"/>
  <bx:dim name="z" indexFrom="0" indexTo="19"/>
</bx:array>
```

An implemented method of data retrieval using BinX is by taking a slice of a data array. With the above example of four array dimensions, a Slice of data could be taken by specifying the index of two dimensions, giving the results as the two dimensions whose indices were not specified. For example w and x could be returned if the query was supplied with the data that y=0 and z=0, so all values of w and x would be returned where y and z were satisfied.

Using these techniques of building up data structures, and arrays for defining large sets of data, it is realistically possible to describe large sets of scientific data using BinX files.

3.5 Data Transformation

The OGSA-DAI Release 1 package only includes the standard transformation of query results to include XML metadata before delivery. This project introduces methods on the delivery mechanism for improved handling of scientific data. New capabilities are added for handling large volumes of data, by adding compression, security and authentication to results returned.

3.5.1 Data Compression

A method for handling large volumes of data can be approached by using compression on the data being returned from a query. This can be achieved using the GZip compression/decompression algorithm [17] on the XML results being returned. By adding the ability to specify the results to be compressed using the GZip algorithm to the script of the DAI interface, the data transformation can be made.

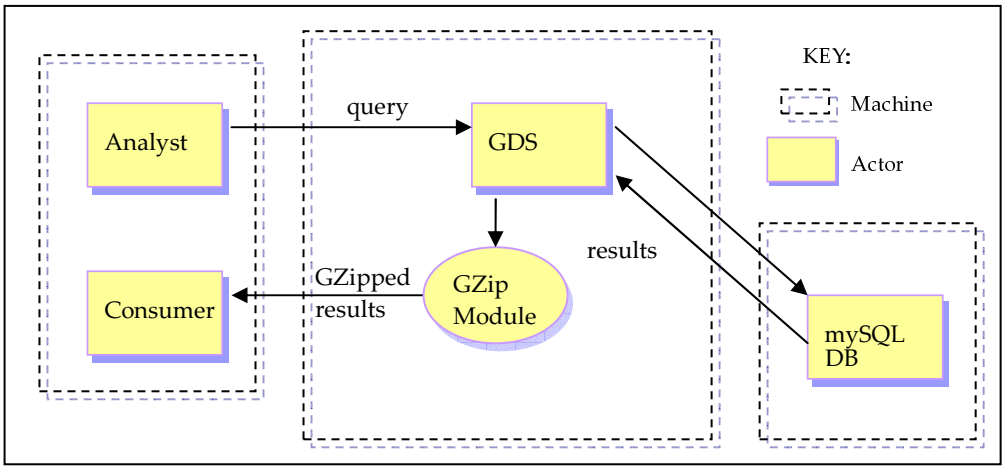

Figure 12

The design of this requires the implementation to return the results using the new Socket delivery mechanism, as the compressed results will not conform to XML, so cannot be returned as the results of the original query.

3.5.2 Data Privacy and Authentication

A delivery mechanism using a socket to a given recipient as delivery would require additional capabilities for data privacy and authentication. A possible delivery mechanism for a scenario involving small sets of data required regularly could be via SMTP. It is possible to use this and the existing S/MIME standard [18] for security, developed by RSA Data Security as an extension to the MIME standard. S/MIME can be used to encrypt and/or sign messages of

data, providing privacy and authentication of the sender of the data, and data integrity.

Authentication is achieved by using S/MIME to create a message of two parts, one being a content part, with the other a Cryptographic Signature part. The main body contains the results, while the authenticating message contains a hash of the main body, encrypted with X.509 certificate encryption [19]. This provides authentication through the successful decryption of the hash using the senders public key, and integrity of the message by comparing the hash with the hash of the message body, notifying of damage in transit. The process is shown in Figure 13.

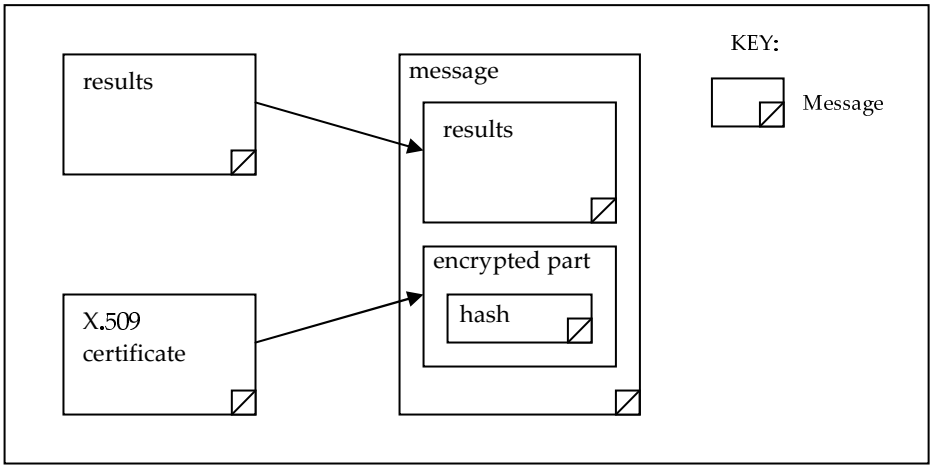

Figure 13

Privacy can be achieved through enveloping the entire signed message into one encrypted part. This provides encryption of all data, while maintaining authentication and integrity through the hashed checksum. The process of signing the message is the same as previously in Figure 13, but there is a final stage of encryption of the entire message, shown in Figure 14.

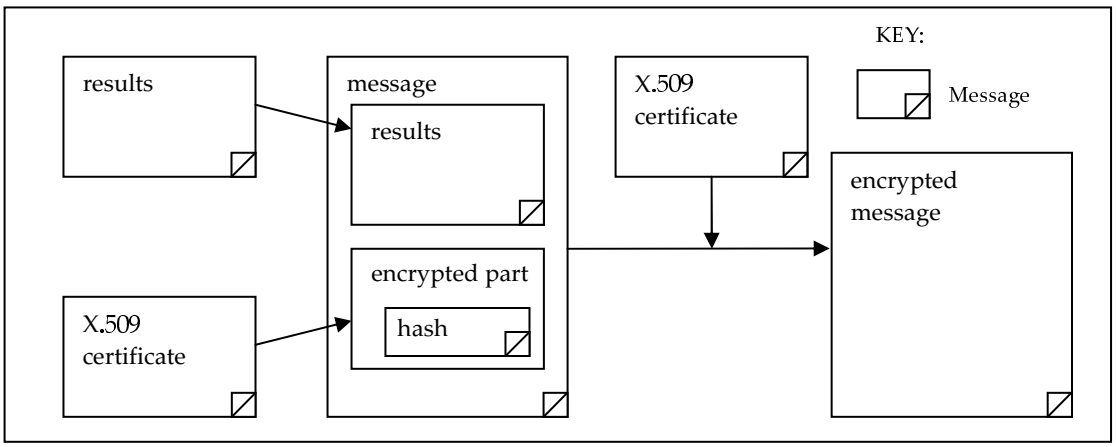

Figure 14

3.5.3 DES Encryption

These privacy and authentication methods in 3.5.2 require a large amount of overhead for the encryption of an entire message using RSA encryption. A more efficient way of encrypting data is by using symmetric session keys, which do not add headers or split the message into parts as involved in S/MIME. For this method, the same key is used to both encrypt and decrypt the message. For this encryption, a method such as DES can be used. In order to transport the key, a private-public keypair can be used. By using the public key of a RSA keypair, the DES key can be encrypted and included in the request to OGSA-DAI. The message data can then be sent back encrypted in the response. The process of encapsulating the DES key in the XML Query, encrypted by an RSA key that is sent to the OGSA-DAI interface, is shown below in Figure 15.

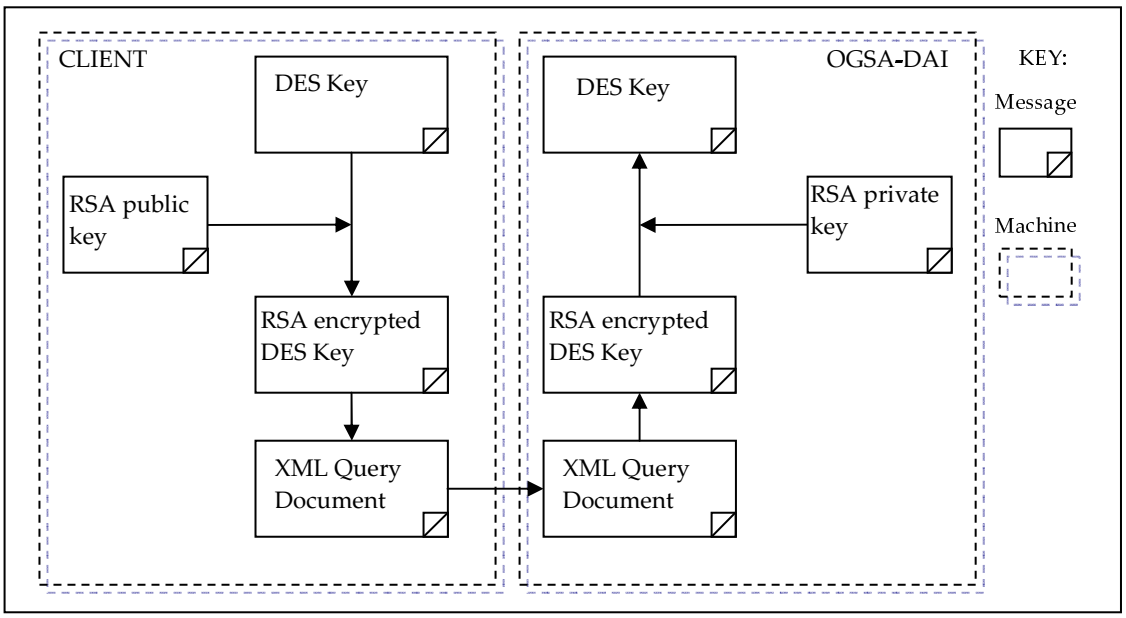

Figure 15

By restricting the private RSA key to only the OGSA-DAI package, the security of the DES key is ensured.

4 Implementation of Design

4.1 Grid Delivery Activity

In order to implement a new delivery mechanism into OGSA-DAI, the method for specifying the recipient of the query results is by specifying a TCP address and port to send the data to. It is possible to add the functionality into the current system by adapting the current Grid Data Service Script, the design of which of which is described in 3.3 . By adding to the possible entries of the <To> field, it is be possible to specify a recipient, and transfer the data to them.

4.1.1 Mechanism Implementation

The sequence diagram Figure 16 shows how the process has changed, the Grid Data Service now sends the results to the Consumer, with the status of that operation returned as the response to the Analyst when the script has completed. It is assumed an instance of the GDS has already been created from a factory, and the Analyst has the handle of this instance.

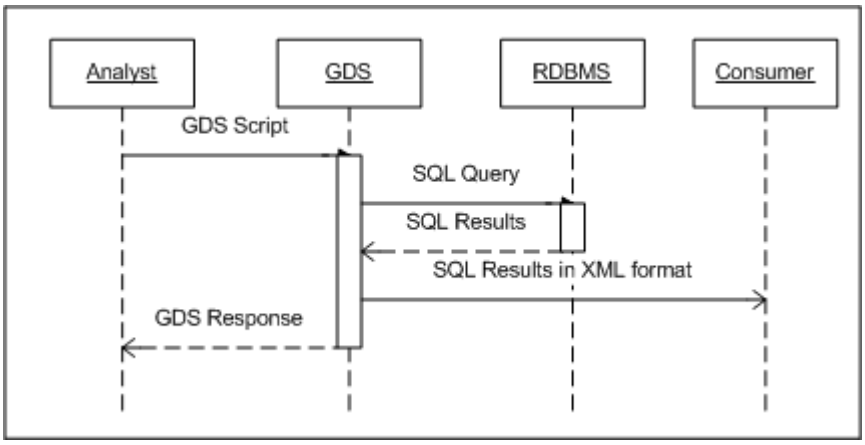

Figure 16

For this, the delivery mechanism code follows this chain of events:
start delivery
if sending to a listening server
create a socket to the given ip and port
print the query results to that socket
close the socket
return the status of the delivery as the script response
end delivery

4.1.2 Query Format

In the GDSS, the field for specifying the recipient of the Delivery Activity in the OGSA-DAI is:

```
<To>response</To>
```

With *To* being the XML field name, and *response* being the normal recipient to return to using the default script response. By also accepting an ip and port as the recipient, a TCP connection can be made for an alternative delivery mechanism. The format chosen was to accept all the recipient data in the <To> field, but the OGSA-DAI scripts can be easily be adapted to allow extra fields instead, such as <IP> and <Port>.

```
<To>server:ip:port</To>
```

for example;

```
<To>server:192.168.0.4:5000</To>
```

4.1.3 Delivery Response

The response is a XML document, with 2 fields, *<statuscode>*, a numeric value for the response, and *<status>*, a short text description of the delivery.
As errors may occur, such as the GDS trying to send data to a TCP socket that is not listening, there are 3 possible status responses that may be sent back.

- `<statuscode>100</statuscode>`
  `<status>success</status>`
  Returned on success of sending the results to the destination specified in the script.
- `<statuscode>200</statuscode>`
  `<status>socket error</status>`
  Returned on failure of sending the results to the destination specified in the script, when it was not possible to send the data over the TCP socket to the destination.
- `<statuscode>201</statuscode>`
  `<status>syntax error</status>`
  Returned on failure to parse the field for the destination IP and PORT.

The Delivery Activity does not handle any input from the Consumer actor, the recipient. It is assumed that if the data is sent successfully over the TCP connection, then the delivery has been successful.

4.2 BinX Binary Data Queries

For the handling of binary data requests using BinX, the implementation was chosen to have the BinX file and Binary Data in a database tier, and retrieve the data as needed to parse a GDSS script involving a BinX data request. The row structure of an entry in the database is as follows

```
+-------+---------------+-------------+----------+
| id    | description   | xml         | data     |
+-------+---------------+-------------+----------+
| [int] | [varchar(20)] | [text]      | [blob]   |
+-------+---------------+-------------+----------+
```

The *id* contains an integer that can be used to reference each entry of data. The *description* is a short description outside of the binx xml entries for the file. The *xml* column contains what would be the Binx File. The *data* column contains the binary data. This is stored as a blob in the database structure.

4.2.1 Query Syntax

The script for performing a query on BinX data is similar to the OGSA-DAI query described in section 3.3 , except the <*statement*> field allows the following syntax, similar to a SQL query:

```
BINX SLICE <col>,<row> FROM <table>.<id> USING <values>
```

Where

- <col> - the name of the dimension to be used as the column of the sliced data
- <row> - the name of the dimension to be used as the row of the slice data
- <table> - the name of the table in the database where the BinX data is situated
- <id> - the id used to find the row with the requested data in the database
- <values> - a list of values to specify for the dimensions not involved in the slice, specified in a comma-delimited list. *e.g. w=3,x=7*.

4.2.2 Query Implementation

The sequence diagram for a BinX request is shown in Figure 17, showing an instance of the Grid Data Service being sent a script involving a BinX Slice.

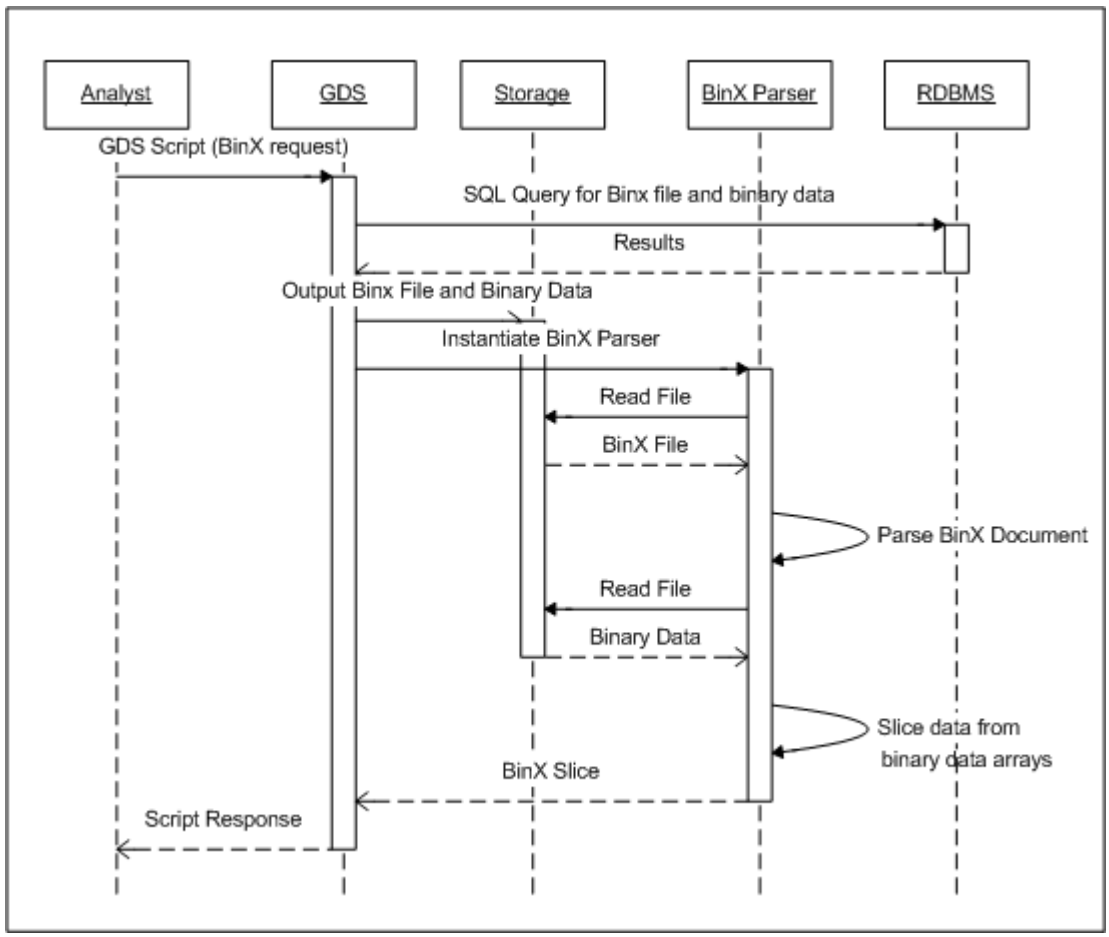

Figure 17

The BinX File and Binary data are read out of the database into the local storage of the Grid Data Service. The BinX file is altered so it points to the new destination of the Binary data file, by altering the *src*= field. On instantiating the BinX Parser, it reads in the BinX File, and reads in the Binary Data. Using the values given in the original statement, a Slice can be made, and returned to be handled by the Delivery response of the GDS.

4.2.3 Output Format

The data from a slice is transformed into a XML document, with the header containing the column count, and the data-type description for each column. Sample output of 2 data values is shown below.

```
<RowSet>
<properties></properties>
<metadata>
  <column-count>1</column-count>
  <column-definition>
    <column-type-name>class java.lang.Short</column-type-
name>
    <column-type-name>class java.lang.Short</column-type-
name>
  </column-definition>
</metadata>
<data>
  <row>
    <col>40</col>
    <col>41</col>
  </row>
</data>
</RowSet>
```

4.3 BinX Binary Update Queries

BinX is still a work in progress, and is liable to change after the completion of this project and report. The current parser available has no ability to update the binary data, this recoded implementation allows for the update of values directly to binary arrays. In order to perform an update on a set of scientific data, such as a large scientific array, a mechanism to change single values was implemented.

4.3.1 Update Query Syntax

The statement format to perform such an update is shown below.

```
BINX UPDATE <value> FROM <table>.<id> AT <position>
```

Where

- <value> - the value to use as the new entry at the specified position in the array

- <position> - the nth data entry in the array to replace, with 0 being the first data value in the array.
- <table> - the name of the table in the database where the BinX data is situated
- <id> - the id used to find the row with the requested data in the database

Using this method, a small change in a binary data file is possible. With this operation, no data is returned on a successful update, allowing for any extension to return arbitrary data.

4.3.2 Update Implementation

The Sequence Diagram for the process of updating a value in a Binary Data value is shown in Figure 18.

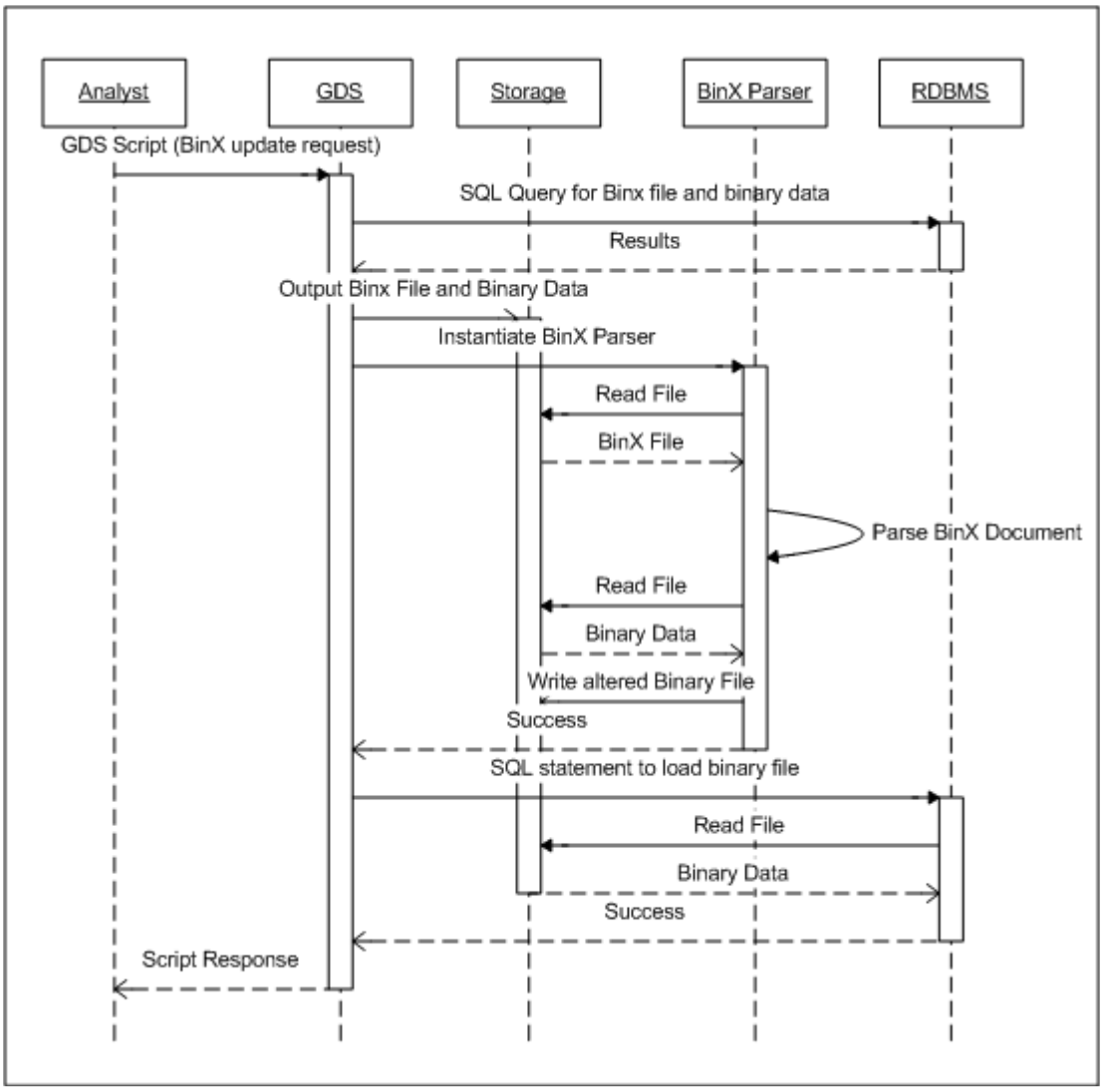

Figure 18

The java steps for this process are shown below.

start BinX update handler
retrieve BinX file and binary data using given id and table
write both files to disk
instantiate BinX parser to read BinX file
write full binary data array to file, altering the statement at given position
perform SQL statement to load binary file back into database, overwriting original
end BinX update handler

4.4 Data Compression

GZip compression was used to perform data compression on results. This results in a stream of data that isz not an XML document. Only the new delivery mechanism detailed in 4.1 can be used, allowing the results to be delivered to a given recipient and a normal XML response from the script returned. In order to specify that this data transformation should be performed during the delivery of the script, the following entry must exist in the <delivery> section of the script submitted to the Grid Data Service.

```
<Encoding>GZIP</Encoding>
```

This alters the delivery mechanism in 4.1 by adding another stage before delivery is made.

if GZip encoding is specified
compress stream using java builtin classes
continue delivery

4.5 Data Privacy with S/MIME encoding

The first data encryption mechanism implemented was S/MIME encoding. S/MIME is normally for use in messages sent over SMTP, but for the purpose of this project only S/MIME was used, and not the ability to send to SMTP servers. The ability to sign messages was implemented, using a certificate on the server. Also implemented was S/MIMEs ability to encrypt the entire message, so only the recipient could see any data being transported. The method to specify that encryption should be used is similar to the method to specify the data should be compressed, and included in the script sent to the GDS.

```
<Encoding>SMIMESIGN</Encoding>
```

This results in a message that has an attachment for authentication of the sender. A SHA1 hash is produced from the message contents, and this is encrypted using a private key from a RSA private/public keypair. The holder of the public key can now decrypt this and compare the decrypted SHA1 hash with the contents of the message body. Only the holder of the private RSA key used to encrypt the SHA1 hash could have produced the signature.

```
<Encoding>SMIMEENCRYPT</Encoding>
```

With this encoding method, the signature stage of producing a SHA1 hash and encrypting this using a private key is performed. A secondary stage of encrypting the entire message with the public key of the recipient produces a S/MIME message which only the holder of the recipient's private key can view.

For the production of S/MIME attachments and messages, a S/MIME encryption provider for java was used, produced by ISNetworks, available from http://www.isnetworks.com/smime/ . These java classes allowed for the creation of the S/MIME messages given the content

Pseudo code for the events in java for the entire process are shown below.

keystore is loaded
OGSA private key loaded into keystore
creation of message to insert data into
set content of message to data
create signed attachment of SHA1 hash
encrypt using private key from keystore

create enveloped message body part
encrypt using recipients public key
return enveloped message

This S/MIME is then returned for processing by the Delivery Activity of the script submitted the GDS.

4.5.1 DES Encryption

Using RSA encryption for large amounts of data requires greater processing power than methods like DES. For the implementation of symmetric key encryption, session encryption, the key to be used is supplied in the XML document script making the request to the GDS.

As discussed in the design, the 56bit DES key (of 64bits length) cannot be sent in plaintext in the document, it must be encrypted first. Using a public RSA key of the OGSA-DAI service, the DES key is encrypted, so only the OGSA-DAI service can decrypt it. When this is passed in the script to perform a request, the key can be decrypted using the private key stored by the GDS. This results in only the request sender and GDS, via the private RSA key, having the session key. The results transmitted can then be encrypted with the DES key. To have the script results encrypted in DES, the following entries are used in the <delivery> section of the GDSS.

```
<Encoding>DES</Encoding>
<Key>[key]</Key>
```

Where

- [key] – the RSA encrypted DES key of originally 64 bits length

The process of encryption is again handled by the delivery activity part of OGSA-DAI. It occurs when delivery is called for, and is summarized by the following pseudo code

extract RSA encrypted DES key from GDS script
decrypt RSA key using private RSA keypair
encrypt message using DES key
return encrypted message

5 Program Testing

5.1 Introduction

The tests outlined here check the coding of the mechanisms implemented into the OGSA-DAI package, with a usage test to examine each feature.

In all of the tests, the OGSA-DAI package was deployed under Jakarta Tomcat 4.1.12, the Java Application server, having followed the install instructions of the package. The Database Roles were set up to access a local mySQL server, in which test data was stored. All scripts were run on the GDS via the OGSA-DAI client, using the following syntax in the OGSA-DAI directory.

java uk.org.ogsadai.client.Client
http://localhost:8080/ogsa/services/uk/org/ogsadai/wsdl/GridDataServiceRegistry
"src\xml\GridDataServiceFactoryScript.xml" "src\xml\ScriptName.xml"

The GridDataServiceFactoryScript.xml is a predefined script containing instructions for creating a Grid Data Service and what config to use. For each test, a different ScriptName.xml file was used, this is the Grid Data Service Script. The GridDataServiceFactoryScript was the same for each test, this script controlled the configuration used for creating a Grid Data Service, and which Data Resource to access. All tests were done to the same database, in a mySQL server. A sample command line test session is shown in appendix 8.6 .

The structure of test databases used is included on the cd accompanying the report, along with the scripts used for testing. A test is performed for each design feature implemented in section 4, in the Tests subfolder. In order to receive data from tests that sent their response to a listening socket, the linux command used is as follows.

netcat -lp 5000 > TestOutputN.txt

This listened for a TCP connection on port 5000, writing the data received to a file then exiting when the connection was closed. The output for each test is available in the Tests subfolder. Any server can be specified, the test server used for these tests was a networked server obtainable by the DNS name relle.dyndns.org.

5.2 Test 1 – Socket Delivery

In order to test the TCP socket delivery system, a test involving a single query returned to a listening server was chosen. The script ran was Test1.xml. The test is successful if the results are sent over TCP and *statuscode* 100 for success was returned. The script selected the column id from table arraysample and gave the recipient of the Delivery to be

<To>server:relle.dyndns.org:5000</To>

Using the method described in the introduction, the results of this test were recorded in the file TestOutput1.txt. The results were compared to those returned using the normal Delivery method as the script response and were found to be identical. The first test was found to be successful, and the socket delivery of results was demonstrated.

5.3 Test 2 – BinX Query

In order to test the BinX methods of retrieving arrays of binary data, a test was performed to execute a BinX Slice on a set of data in the database, which is available on the project cd. The script used was Text2.xml. The action performed is as follows.

<DefineParameter name="table">arraysample</DefineParameter>
<DefineParameter name="binxvalues">z=2</DefineParameter>
<DefineParameter name="id">1</DefineParameter>
<DefineParameter name="binxcol">x</DefineParameter>
<DefineParameter name="binxrow">y</DefineParameter>
<Statement name="xyz2" dataResource="MyDataResource">
BINX SLICE <UseParameter reference="binxcol"/>,<UseParameter reference="binxrow"/> FROM <UseParameter reference="table"/>.<UseParameter reference="id"/> USING <UseParameter reference="binxvalues"/>
</Statement>

The <Statement> results in

BINX SLICE x,y FROM arraysample.1 USING z=2

The test took the BinX set of data stored in the table arraysample, using the id of 1 to locate it. The sliced array of data was dimensions x and y, giving the 3rd

dimension z a value of 2 to take the slice across. The output produced is recorded in TestOutput2.txt, and is also included in Appendix section 8.7 .

5.4 Test 3 – BinX Update Query

The aim of this test is to update values in a binary array existing on a data resource attached to the Grid Data Service. It is successful if querying the binary array after updating the values gives an array of values including the updated value.

To test the BinX method of updating values in a binary array of data, a test was performed to update a single value in the binary array. Then another query was used to check the updated values. The script for executing the update was Test3.xml, and to query the new values is Test3b.xml. The results from both scripts are recorded in TestOutput3.txt and TestOutput3b.txt on the cd. Test3.xml invokes the following statement.

BINX UPDATE 26 FROM <UseParameter reference="table"/>.<UseParameter reference="id"/> AT 40

which translates into

BINX UPDATE 26 FROM arraysample.1 AT 40

As a result of this statement, the value at position 40 (the first position is position 0) was changed to 26. The tests were executed on binary arrays of shorts. When Test3b.xml is executed, the data returned is as follows.

```
<data><row>
        <col>26</col>
        <col>41</col>
        <col>42</col>
etc.
```

Position 40 had been updated to the given value, so Test 3 is deemed successful.

5.5 Test 4 – Data Compression

The purpose of this test is to perform a query to the GDS, have the response sent back compressed, then check the compression by decompressing the response. The script to perform this test is Test4.xml, and selects the column id from table arraysample, with the following field in the <Delivery> section of the script

<Encoding>GZIP</Encoding>

After executing this script, the output is shown in TestOutput4.txt. In order to decompress this, the following linux command was used

gzip -d < TestOutput4.txt > TestOutput4decompress.txt

The output of the query could be found in TestOutput4decompress.txt in plain text. This gave a successful test.

5.6 Test 5 – S/MIME signing

To test S/MIME signing, a query is made to the GDS which includes instructions to add a signed signature to the results. The test is successful if a message is returned with the signed signature. The script used was Test5.xml, which specifies the following encoding to be used

<Encoding>SMIMESIGN</Encoding>

The script executes the same query as Test 4, returning the ids available in the table arraysample. After executing this script, the output was recorded as TestOutput5.txt.

This contains a MIME message of 2 parts, the first part being the plain results that can be found as the decompressed results of Test4. Part 2 is found to be the RSA encrypted SHA1 hash the 1st part of the message. As a result of this, Test 5 was found to be successful.

5.7 Test 6 – S/MIME encryption

To test the S/MIME signing and enveloped encryption, the query made contains the instructions to have the GDS encrypt the entire message after signing. The test is successful if the returned message is fully encrypted and

contains the query results after decryption. The script used for this test was Test6.xml, the statement was the same as for test 4 and 5. The option

<Encoding>SMIMEENCRYPT</Encoding>

was used for delivery in the script. After execution the script results were saved as TestOutput6.txt. On examination it was found to be the correct S/MIME encrypted message. Decryption was achieved using the public key of the recipient, and the correct signed message was found inside. Test 6 was successful.

5.8 Test 7 – DES session encryption

To test the DES encryption of results, the final test involves executing a script containing a RSA encrypted DES key, used for encrypting the results returned. Test success is found by the accurate results returned being decrypted using the original DES key. The SQL query used is the same query for Tests 4 to 6. The filename for the test is Test7.xml. The following options for Delivery are used to produce the test results.

<Encoding>DES</Encoding>

In order to produce the values of the <Key> fields that are passed, a DES key was generated to be used to handle the encryption and decryption required. This key is represented in java in the following way.

```
byte[] keyBytes = {
        (byte)0x48, (byte)0x61, (byte)0x29, (byte)0x32,
        (byte)0x56, (byte)0x35, (byte)0x57, (byte)0x31
    };
```

This is RSA encrypted using the private number pair, with the final value being used in the script shown at Appendix 8.8 . On execution of Test7.xml, the results were recorded as TestOutput7.txt.

The decryption of this file using the DES key above resulted in the expected plaintext query results, resulting in a successful test.

6 Conclusions and Future Work

During this project, a number of delivery and data transformation mechanisms were successfully implemented into the OGSA-DAI package. The ability to deliver data to listening TCP sockets was implemented. Through the redesigning of the BinX package it was possible to implement functionality to handle accessing and alteration of binary arrays of data. It is possible to return two dimensional slices of data transformed to XML format, and also the ability to update the binary data using XML documents for the OGSA-DAI scripts. The ability to handle large volumes of results has been added to by allowing for data compression as a data transformation option. Data Privacy and Authentication is implemented by allowing for data transformation into S/MIME messages. This allows for a secure signature or encryption of the entire message. For a more efficient method of encryption of large amounts of data, DES encryption has been implemented through a user specified key.

The implementation of data delivery to specified TCP sockets allowed for the addition of the data compression transformation, by allowing the delivery of data that was not XML compliant. This enhances the ability of OGSA-DAI to handle large volumes of data that may occur from requests such as large amounts of scientific data. For scientific Grid projects like AstroGrid, the OGSA-DAI BinX integration implements many of the features set out for in their original mission. The BinX Binary Data interface allows for uniform archive querying, while OGSA-DAI caters for the ability to access multiple datasets simultaneously. The ability to retrieve small sets of data from large arrays is made possible, with many possible implementations of use for integrating large amounts of binary data into the Grid. Security of the returned results has also been added, while Grid Security does exist for accessing resources, this implementation adds data privacy and authentication for data resources being returned, using current methods such as RSA and DES encryption.

For future work on data access and integration into the grid, there are still improvements that could be made on the integration of BinX into the OGSA-DAI package. The existing implementation requires the database to be a mySQL database, and running on the same filesystem as OGSA-DAI, as the binary data is loaded using SQL commands pointing to the file system. Regarding the security mechanisms implemented, the draft paper [20] contains a proposal for the implementation of security into OGSA and the Grid through the use of a SAML message format for requesting and

expressing authorization. This idea could be used for to expand the implementation of security in this project with a standard for encryption of results returned, from a maintainable 3rd party source.

Grid technology is a rapidly changing field; standards are currently being designed as the scenarios for Grid use are identified. This project has used the currently available standards, in the future alternative technologies for the implementation may be available.

7 References

[1] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", June, 2002. http://www.globus.org/research/papers/ogsa.pdf *(last accessed 7 May 2003)*

[2] S. Tuecke et. al "Open Grid Services Infrastructure (OGSI)", February, 2003. http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-23_2003-02-17.pdf *(last accessed 7 May 2003)*

[3] B. Collins, A. Borley, N. Hardman, A. Knox, S. Laws, J. Magowan, M. Oevers, E. Zaluska, "Grid data Services – Relational Database Management Systems - Version 1.0" presented at Conf. of the Global Grid Forum 5, Edinburgh, Scotland, 21-24 July, 2002.

[4] E. Christensen, F. Curbera, F. Meredith, S. Weerawarana, "Web Services Description Language (WSDL) 1.1", March, 2001. http://www.w3.org/TR/wsdl *(last accessed 7 May 2003)*

[5] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *International Journal of High Performance Computing Applications*, 15 (3). 200-222. 2001.

[6] W. Hoschek, G. McCance, "Grid Enabled Relational Database Middleware". presented at Conf. of the Global Grid Forum 3, Frascati, Italy, 7-10 October, 2001.

[7] B. Collins, N. Hardman, A. Knox, S. Laws, J. Magowan, M. Oevers, "IBM - OGSA - DAI - WP6 - Scenarios and Patterns - Version 1.0", OGSA-DAI v1.0 Manual, May, 2002.

[8] M. Atkinson, V. Dialani, L. Guy, I. Narang, N. Paton, D. Pearson, T. Storey, P. Watson, "Grid Database Access and Integration: Requirements and Functionalities" presented at Conf. of the Global Grid Forum 7, Tokyo, Japan, 4-7 March, 2003

[9] D. Pearson, "Data Requirements for the Grid Scoping Study Report", Technical Report for "Databases and the Grid BOF" at GGF4, Toronto, Canada, February, 2002.

[10] B. Collins, N. Hardman, A. Knox, S. Laws, J. Magowan, M. Oevers. "IBM - OGSA - DAI - WP6 - Architecture and Design - Version 1.3", OGSA-DAI v1.0 Manual, August, 2002.

[11] P. Watson, "Databases and the Grid", Technical Report CS-TR-755, University of Newcastle, 2001. http://www.cs.man.ac.uk/grid-db/papers/dbg.pdf *(last accessed 7 May 2003)*

[12] S. Laws, A. Knox. "Open Grid Services Architecture – Data Access and Integration – Grid Data Service", OGSA-DAI v1.0 Manual, February, 2003.

[13] Peter Allan, et al. "AstroGrid April Proposal", April, 2001. http://wiki.astrogrid.org/bin/view/Astrogrid/AprilProposal. April 2nd 2001 *(last accessed 7 May 2003)*

[14] M. Westhead, "BinX - The Binary XML Description Language", April, 2002. http://www.ph.ed.ac.uk/ukqcd/community/the_grid/xml_schema/BinXIntro2.pdf *(last accessed 7 May 2003)*

[15] M. Westhead, M. Bull, "Representing Scientific Data on the Grid with BinX – Binary XML Description Language", January, 2003. http://www.epcc.ed.ac.uk/~gridserve/WP5/Binx/sci-data-with-binx.doc *(last accessed 7 May 2003)*

[16] "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August, 1985.

[17] P. Deutsch, "GZIP file format specification version 4.3", RFC 1952, May, 1996. http://www.faqs.org/rfcs/rfc1952.html *(last accessed 7 May 2003)*

[18] B. Ramsdel, "S/MIME Version 3 Message Specification", RFC 2633, June, 1999. http://www.faqs.org/rfcs/rfc2633.html *(last accessed 7 May 2003)*

[19] R. Housley, et. al. "Internet X.509 Public Key Infrastructure Certificate and CRL Profile", RFC 2459, January, 1999. http://www.faqs.org/rfcs/rfc2459.html *(last accessed 7 May 2003)*

[20] V. Welch et. al, "Use of SAML for OGSA Authorization" presented at Conf. of the Global Grid Forum 7, Tokyo, Japan, 4-7 March, 2003.

8 Appendices

8.1 Sample WSDL document

```
<?xml version="1.0"?>
<definitions name="StockQuote"
      targetNamespace="http://example.com/stockquote/definitions"
      xmlns:tns="http://example.com/stockquote/definitions"
      xmlns:xsd1="http://example.com/stockquote/schemas"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns="http://schemas.xmlsoap.org/wsdl/">

      <import namespace="http://example.com/stockquote/schemas"
           location="http://example.com/stockquote/stockquote.xsd"/>

      <message name="GetLastTradePriceInput">
            <part name="body" element="xsd1:TradePriceRequest"/>
      </message>

      <message name="GetLastTradePriceOutput">
            <part name="body" element="xsd1:TradePrice"/>
      </message>

      <portType name="StockQuotePortType">
            <operation name="GetLastTradePrice">
                  <input message="tns:GetLastTradePriceInput"/>
                  <output message="tns:GetLastTradePriceOutput"/>
            </operation>
      </portType>
</definitions>
```

This XML document is based on a W3C example available at the WSDL definition at http://www.w3c.org/TR/wsdl. It defines two messages, and a portType. The portType defines the operation of taking an input message, and defines the output message associated with it. The elements are defined in a separate XML XSD.

8.2 Matrix prototype QTD

```
<?xml version="1.0" encoding="UTF-8"?>
<QTD:QTD processName="StreamingDelivery"
xmlns:QTD="http://www.ibm.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com ../xsd/QTD.xsd ">
      <QTD:Activity name="SubmitQuery">
            <QTD:Param name="QueryString">SELECT * from
      myimages</QTD:Param>
      </QTD:Activity>
      <QTD:Activity name="DeliverData">
            <QTD:Param name="SourceType">stream</QTD:Param>
```

```
            <QTD:Param name="SourceAddress">
            http://localhost:8080/Matrix/StreamingServlet</QTD:Param>
            <QTD:Param name="SourceRow">1-5</QTD:Param>
            <QTD:Param name="SourceColumn">3</QTD:Param>
            <QTD:Param name="DestinationType">file</QTD:Param>
            <QTD:Param name="DestinationPath">C:/temp</QTD:Param>
            <QTD:Param
            name="DestinationName">file.$ROW.$COL.jpg</QTD:Param>
            <QTD:Param
            name="DestinationAddress">localhost</QTD:Param>
            <QTD:Param name="DestinationPort">8080</QTD:Param>
      </QTD:Activity>
      <QTD:Result/>
</QTD:QTD>
```

Used in the Matrix prototype to perform a query, this QTD would perform a database lookup of '*SELECT * from myimages*', returning the files by writing them to the given location of *DestinationPath*.

8.3 Contacts Table from Benchmarking

Number of results	MatrixService output	Ascii output direct from database
0	250 ms	<0.01 ms
10	290 ms	<0.01 ms
20	310 ms	<0.01 ms
30	335 ms	<0.01 ms
40	410 ms	<0.01 ms
50	500 ms	<0.01 ms
70	580 ms	<0.01 ms
90	625 ms	<0.01 ms
120	730 ms	<0.01 ms

Table of actual results found from benchmarking the Matrix prototype release. Note the time taken to discover the service and create an instance was recorded at roughly 4 seconds each time, only the time taken to perform the query and return the results was recorded.

8.4 Sample GDS script

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- (c) International Business Machines Corporation, 2002, 2003. (c) University of
Edinburgh 2002, 2003.-->
<!-- See OGSA-DAI-Licence.txt for licencing information.-->
<GridDataServiceScript>
 <Header>
    <ScriptName>Example 1</ScriptName>
    <Version>
       <Config>config</Config>
       <ScriptEnvironment>environment</ScriptEnvironment>
    </Version>
```

```
    <Originator>GSR of Originator</Originator>
  </Header>
  <Body>
        <Documentation>Select with data delivered with the response</Documentation>
        <DefineParameter name="table">myimages</DefineParameter>
        <DefineParameter name="id">id = 2</DefineParameter>
        <Statement name="xyz" dataResource="MyDataResource">
               SELECT * FROM <UseParameter reference="table"/> WHERE <UseParameter
reference="id"/>
        </Statement>
        <Delivery name="delivery">
               <Mechanism type="bulk"/>
               <Mode type="full"/>
               <From>xyz</From>
               <To>response</To>
        </Delivery>
        <Execute name="execute">xyz</Execute>
  </Body>
</GridDataServiceScript>
```

8.5 GDSS Xml Schema

The following XML schema is used for scripts sent to the Grid Data Service.

```
          <?xml version="1.0" encoding="UTF-8"?>
       <xs:schema targetNamespace="http://ogsadai.org.uk/P2R1/schemas/gdss"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:GDSS="http://ogsadai.org.uk/P2R1/schemas/gdss" elementFormDefault="qualified">
           <xs:annotation>
               <xs:documentation xml:lang="en">Basic types used in the GDS-Script and
other scripts.</xs:documentation>
           </xs:annotation>
           <xs:complexType name="VersionType">
               <xs:sequence>
                   <xs:element name="Config" type="xs:string"/>
                   <xs:element name="ScriptEnvironment" type="xs:string"/>
               </xs:sequence>
           </xs:complexType>
           <xs:complexType name="HeaderType">
               <xs:sequence>
                   <xs:element name="ScriptName" type="xs:string">
                       <xs:annotation>
                           <xs:documentation>Client supplied name that is used to
refer to this script instance.</xs:documentation>
                       </xs:annotation>
                   </xs:element>
                   <xs:element name="Version" type="GDSS:VersionType">
                       <xs:annotation>
                           <xs:documentation>Resource requirements for the
script.</xs:documentation>
                       </xs:annotation>
                   </xs:element>
                   <xs:element name="Originator" type="xs:string" minOccurs="0">
                       <xs:annotation>
                           <xs:documentation>Identity of client submitting the
script.</xs:documentation>
                       </xs:annotation>
                   </xs:element>
               </xs:sequence>
           </xs:complexType>
           <!-- Support for basic Engine primitives generated here-->
           <xs:simpleType name="DocumentationType">
               <xs:annotation>
                   <xs:documentation> Element to documentation inside the
script.</xs:documentation>
               </xs:annotation>
               <xs:restriction base="xs:string"/>
           </xs:simpleType>
           <xs:complexType name="DefineParameterType" mixed="true">
               <xs:annotation>
```

```
                    <xs:documentation> Mechansim for
parameterisation.</xs:documentation>
                </xs:annotation>
                <xs:attribute name="name" type="xs:string" use="required"/>
            </xs:complexType>
            <xs:complexType name="UseParameterType">
                <xs:annotation>
                    <xs:documentation> Mechansim for parameterisation.
</xs:documentation>
                </xs:annotation>
                <xs:attribute name="reference" type="xs:string" use="required"/>
            </xs:complexType>
            <xs:complexType name="ExecuteType">
                <xs:annotation>
                    <xs:documentation> Container for execution
information.</xs:documentation>
                </xs:annotation>
                <xs:simpleContent>
                    <xs:extension base="xs:string">
                        <xs:attribute name="name" type="xs:string" use="required"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:complexType>
            <!-- Support for Delivery generated here -->
            <xs:simpleType name="ToType">
                <xs:annotation>
                    <xs:documentation> where a data set is to be delivered
to</xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string"/>
            </xs:simpleType>
            <xs:simpleType name="FromType">
                <xs:annotation>
                    <xs:documentation> where a data set is supposed to come
from</xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string"/>
            </xs:simpleType>
            <xs:complexType name="MechanismType">
                <xs:annotation>
                    <xs:documentation> the delivery mechanism</xs:documentation>
                </xs:annotation>
                <xs:attribute name="type" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:NMTOKEN">
                            <xs:enumeration value="bulk"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:complexType>
            <xs:complexType name="ModeType">
                <xs:annotation>
                    <xs:documentation>the mode of operation that is to be used for
delivery </xs:documentation>
                </xs:annotation>
                <xs:attribute name="type" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:NMTOKEN">
                            <xs:enumeration value="full"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:complexType>
            <xs:complexType name="DeliveryType">
                <xs:annotation>
                    <xs:documentation> container for delivery
information</xs:documentation>
                </xs:annotation>
                <xs:choice maxOccurs="unbounded">
                    <xs:element name="Mechanism" type="GDSS:MechanismType"/>
                    <xs:element name="Mode" type="GDSS:ModeType"/>
                    <xs:element name="From" type="GDSS:FromType"/>
                    <xs:element name="To" type="GDSS:ToType" maxOccurs="unbounded"/>
                </xs:choice>
                <xs:attribute name="name" type="xs:string" use="required"/>
            </xs:complexType>
            <!-- Support for Statement generated here -->
```

```
            <xs:complexType name="StatementType" mixed="true">
                <xs:annotation>
                    <xs:documentation> container for statements</xs:documentation>
                </xs:annotation>
                <xs:sequence minOccurs="0">
                    <xs:any namespace="##any" processContents="lax"
maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:string" use="required"/>
                <xs:attribute name="dataResource" type="xs:string" use="optional"/>
                <xs:attribute name="prepared" type="xs:boolean" use="optional"
default="false"/>
            </xs:complexType>
            <!-- top level GDS-S support -->
            <xs:complexType name="BodyType">
                <xs:annotation>
                    <xs:documentation> Root element for the body conent
</xs:documentation>
                </xs:annotation>
                <xs:choice maxOccurs="unbounded">
                    <xs:element name="Documentation" type="GDSS:DocumentationType"
minOccurs="0" maxOccurs="unbounded">
                        <xs:annotation>
                            <xs:documentation>Provide human readable documentation of
any of the body elements.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="Statement" type="GDSS:StatementType"
minOccurs="0" maxOccurs="unbounded">
                        <xs:annotation>
                            <xs:documentation>Statement carries the payload to be
delivered to the database.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="Delivery" type="GDSS:DeliveryType">
                        <xs:annotation>
                            <xs:documentation>Request the delivery
method.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="Execute" type="GDSS:ExecuteType"
maxOccurs="unbounded">
                        <xs:annotation>
                            <xs:documentation>Execute task.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="DefineParameter" type="GDSS:DefineParameterType"
minOccurs="0" maxOccurs="unbounded">
                        <xs:annotation>
                            <xs:documentation>Explicit definition of a
parameter.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                </xs:choice>
            </xs:complexType>
            <xs:complexType name="GridDataServiceScriptType">
                <xs:sequence>
                    <xs:element name="Header" type="GDSS:HeaderType">
                        <xs:annotation>
                            <xs:documentation>Header information.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                    <xs:element name="Body" type="GDSS:BodyType">
                        <xs:annotation>
                            <xs:documentation>Script payload.</xs:documentation>
                        </xs:annotation>
                    </xs:element>
                </xs:sequence>
            </xs:complexType>
            <xs:element name="GridDataServiceScript"
type="GDSS:GridDataServiceScriptType">
                <xs:annotation>
                    <xs:documentation xml:lang="en">Root element.</xs:documentation>
                </xs:annotation>
            </xs:element>
        </xs:schema>
```

8.6 Sample test session

```
C:\>cd ogsa\ogsadai

C:\ogsa\ogsadai>setenv

C:\ogsa\ogsadai>java uk.org.ogsadai.client.Client http://localhost:8080/ogsa/ser
vices/uk/org/ogsadai/wsdl/GridDataServiceRegistry "src\xml\GridDataServiceFactor
yScript.xml" "E:\tests\Test1.xml"
GDSR Number of locators returned: 1
GDSR Locator: http://192.168.0.6:8080/ogsa/services/ogsadai/GridDataServiceFactoryP2R1

Script results:
<GDSSR xmlns="">
<Header>
<ScriptName>Test1</ScriptName>
<Version/>
</Header>
<Body>
<Response><Status>
[GridDataServiceScript:Test1] - finished execution
</Status><Result
name="execute"><statuscode>100</statuscode><status>success</status></Result>
</Response></Body>
</GDSSR>
```

8.7 Test 2 - BinX Query results


```
<RowSet>
<properties></properties>
<metadata>
<column-count>
10</column-count><column-definition><column-type-name>class java.lang.Short</column-
type-name></column-definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
<column-definition><column-type-name>class java.lang.Short</column-type-name></column-
definition>
</metadata>
<data><row>
        <col>40</col>
        <col>41</col>
        <col>42</col>
        <col>43</col>
        <col>44</col>
        <col>45</col>
        <col>46</col>
        <col>47</col>
        <col>48</col>
        <col>49</col>
</row>
<row>
        <col>50</col>
        <col>51</col>
        <col>52</col>
        <col>53</col>
        <col>54</col>
```

```
        <col>55</col>
        <col>56</col>
        <col>57</col>
        <col>58</col>
        <col>59</col>
</row>
</data></RowSet>
```

8.8 RSA encrypted DES key for Test 7

```
<Key>74120220120026930772202390514363916934700883932627887110733107464
493783352668
18180340870290728969247339180636682621837347880283736865842148365427844
44182193
29615132045193970551812481645310127275884377581999115871004793003777060
60035444
22580892378744229859365241359342163519133742923641873252784110485970534
34094250
10715205667156117048288891786136740900091186240406495532927408801532470
7015477
21746096783030421590870049542657592931254572817833168726068131328760253
53527606
39938501663965490965704772698205796259445487237447730060999487423913801
74518
51508851229993318337606640316083491108639997499335855446065908574241032
18648</Key>
```