Implementation of the Random ByteIO interface for OGSA-DAI

Odysseas Bournas

August 24, 2007

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2007
Abstract

The OGSA ByteIO working group has defined a minimal Web Service interface for accessing blocks of bytes from data sources in the Grid. The interface maps easily to existing APIs, such as the POSIX API, so that existing client applications that have no notion of the Grid can easily be connected to it and access distributed data as if they were local files. It is hoped that applications that provide access to data sources in the Grid will decide to support the ByteIO interface so that it can become a standard to which arbitrary clients can speak.

The ByteIO interface is divided into two parts, namely the Random ByteIO and the Streamable ByteIO interfaces. Each one addresses a unique set of use cases, but both focus on accessing sequences of bytes. The objective of the project was to implement the Random ByteIO interface and connect this to the OGSA-DAI middleware, an application that provides access to data sources in the Grid but did not support the interface. This was accomplished and a number of use case scenarios were implemented that demonstrate the potentials of the new functionality and how this can be utilized by clients.
## Contents

1 Introduction 1

2 Background 3

2.1 The problem behind Data Access in the Grid 3

2.2 OGSA-DAI 4

2.2.1 Existing OGSA-DAI mechanisms for accessing data 5

2.3 A new mechanism for accessing data 7

2.3.1 The Random ByteIO interface 8

2.3.2 Web Service Description 9

2.3.3 Use case scenarios for the new mechanism 11

3 Implementing the Random ByteIO interface 13

3.1 File Access mechanism 14

3.2 Thread Safety 16

3.3 The Read operation 17

3.4 The Write operation 19

3.5 The Append operation 20

3.6 The Truncate-Append operation 21

3.7 Bindings 22

4 Connecting the Random ByteIO Service to OGSA-DAI 23

4.1 The OGSA-DAI Software 25

4.2 The Random ByteIO Grid Service 26

4.2.1 The Random ByteIO resources 28

4.3 New Activities 32

4.3.1 The CreateRandomByteIOResource activity 32

4.3.2 The AppendToRandomByteIOResource activity 34

4.3.3 The WriteToRandomByteIOResource activity 36

4.3.4 The ReadFromRemoteRandomByteIOResource activity 39

4.4 Deploying the Random ByteIO service 41

5 Demonstrating the new functionality 43

5.1 Client Software Design 43

5.1.1 Managing endpoint references in the client-side 44

5.2 Implementations of basic Random ByteIO use case scenarios 45
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.1</td>
<td>Asynchronous database query</td>
<td>46</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Insert into Database</td>
<td>52</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Copying data between databases</td>
<td>57</td>
</tr>
<tr>
<td>6</td>
<td>A provisional real use case scenario implementation</td>
<td>60</td>
</tr>
<tr>
<td>6.1</td>
<td>Medical Processing</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>Testing</td>
<td>62</td>
</tr>
<tr>
<td>7.1</td>
<td>Unit tests</td>
<td>62</td>
</tr>
<tr>
<td>7.2</td>
<td>System tests</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>Additional work</td>
<td>64</td>
</tr>
<tr>
<td>8.1</td>
<td>Interoperability testing</td>
<td>65</td>
</tr>
<tr>
<td>8.1.1</td>
<td>The factory service</td>
<td>65</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Results</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions</td>
<td>71</td>
</tr>
<tr>
<td>9.1</td>
<td>Future Work</td>
<td>73</td>
</tr>
<tr>
<td>A</td>
<td>Project Management</td>
<td>74</td>
</tr>
<tr>
<td>A.1</td>
<td>Project Plan</td>
<td>74</td>
</tr>
<tr>
<td>B</td>
<td>Java I/O</td>
<td>76</td>
</tr>
<tr>
<td>C</td>
<td>Client</td>
<td>78</td>
</tr>
<tr>
<td>C.1</td>
<td>User Interface</td>
<td>78</td>
</tr>
<tr>
<td>C.2</td>
<td>Adding a new plug-in to the Client</td>
<td>80</td>
</tr>
<tr>
<td>D</td>
<td>Notes on the interoperability requirements</td>
<td>81</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Input and Output specification of the CreateRandomByteIOResource activity ........................................ 32
4.2 Input and Output specification of the AppendToRandomByteIO activity ........................................ 35
4.3 Input and Output specification of the WriteToRandomByteIOResource activity ........................................ 37
4.4 Input and Output specification of the ReadFromRemoteRandomByteIOResource activity ......................... 39
5.1 Timings .............................................................................. 50
B.1 Characteristics of several Java I/O mechanisms .................................................................................. 77
List of Figures

2.1 An OGSA-DAI data workflow for querying, transforming and delivering data to a client application .................. 6
2.2 Illustration of the arguments that are used by the operations of the Random ByteIO interface to specify subsets of data resources ....... 10
3.1 Invoking a Read operation on the Random ByteIO service .............. 14
3.2 The Read Operation ............................................. 18
3.3 The Write Operation ............................................. 20
3.4 The Truncate-Append Operation .................................... 22
4.1 Architecture diagram of resources and services in OGSA-DAI 3.0 with the Random ByteIO interface connected to it .................. 24
4.2 UML Class Diagram for Random ByteIO service .................... 28
4.3 Template file for the Random ByteIO resource ..................... 29
4.4 UML Class Diagram for the Random ByteIO data resource ........... 30
4.5 Creating and initializing a Random ByteIO resource .................. 31
4.6 The process of creating a new Random ByteIO resource ............... 33
4.7 Example of a workflow construction for creating a new Random ByteIO resource .................................................. 34
4.8 The AppendToRandomByteIOResource activity ....................... 36
4.9 Example of a workflow construction for copying data from a file into a Random ByteIO resource ............................................. 37
4.10 The WriteToRandomByteIOResource activity ........................ 38
4.11 Reading blocks of data from a stream and copying these into an array by resizing it ......................................................... 39
4.12 The ReadFromRemoteRandomByteIOResource activity ................ 40
5.1 UML Diagram of the Client implementation. Five plug-ins are shown for accessing Random ByteIO resources ...................... 44
5.2 Reading a database BLOB in parts .................................. 47
5.3 Writing a BLOB inside a Random ByteIO resource ................... 48
5.4 Reading data from a resource in chunks ............................... 49
5.5 Inserting data into a database using the Random ByteIO .......... 53
5.6 Writing data into a Random ByteIO resource in blocks of bytes .... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Inserting data into a Random ByteIO resource that originate from a third party data provider that supports the Random ByteIO interface.</td>
</tr>
<tr>
<td>5.8</td>
<td>Writing data into a Random ByteIO resource by reading these from a third party provider.</td>
</tr>
<tr>
<td>5.9</td>
<td>Copying data between databases.</td>
</tr>
<tr>
<td>8.1</td>
<td>Creation of a Random ByteIO resource for the purpose of the interoperability tests.</td>
</tr>
<tr>
<td>8.2</td>
<td>Response message of a &quot;CreateResource&quot; operation as it was recorded by the TCPMonitor tool.</td>
</tr>
<tr>
<td>8.3</td>
<td>SOAP Header of a Read request message as it was recorded by the TCPMonitor tool.</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to thank my supervisor, Dr Amy Krause, for her guidance and support throughout the project.
Chapter 1

Introduction

As more and more data are being recorded and organized inside structured collections every day, managing these effectively and efficiently has become a great challenge. Much of the promise of the Grid is its potential for seamless data access and management across geographically distributed computing and storage facilities [1, 9]. Client applications should be able to connect to the Grid and easily gain from the plethora of available data resources in it (subject to the appropriate security mechanisms). This includes locating, accessing and integrating data, while the Grid infrastructure will be hiding the underlying complexity that derives from the distributed, heterogeneous nature of the resources.

The Global Grid Forum (GGF) [14] realized the need for a standard interface to exist that would facilitate the provision of access transparency and provide an efficient mechanism for accessing large amounts of bulk data from data sources in the Grid. The task of specifying such an interface was carried out by the OGSA ByteIO Working Group [15], which eventually published a rendering document of the ByteIO specification that maps to a WSRF-compliant [28] web service.

The interface defines a set of operations that allow accessing sequences of bytes from data sources. That way client applications may access subsets of data, resulting in less amount of data being moved, analyzed and stored either permanently or temporarily. The intention was that the interface would map easily to existing APIs, such as the POSIX API [22], so that existing applications that have no knowledge of the Grid can be connected to the latter and access distributed data resources as if they were local files. It is now hoped that applications that provide access to data sources in the Grid will decide to support the interface so that this can become a standard to which arbitrary clients can speak. The specification can be found here [6].

The ByteIO interface is divided into two parts, namely the Random ByteIO and the Streamable ByteIO interfaces. Each one addresses a unique set of use cases, but both focus on accessing sequences of bytes. The aim of this project was to implement the

---

1The term "data source" is used to denote any facility that produces or accepts data in the Grid (i.e. database system, file system, etc.). For more information see here [9].
Random ByteIO interface and connect this to the OGSA-DAI middleware, an application that provides access to data sources in the Grid but did not support the interface. This included extending OGSA-DAI to support the new interface via its own access framework. OGSA-DAI is described in section [2,2]. A number of basic use case scenarios that had been identified by the ByteIO Working Group and recorded here [18] have been implemented. These demonstrate the potentials of the new functionality and the way it can be utilized by clients via OGSA-DAI. Finally, it should be noted that although achieving good performance was one of the primary design goals of the ByteIO interface, the current project focused more on the functional aspect of the implementation. In any case this does not imply that the design decisions were false, however one should not expect to find extensive performance investigations in this document. Performance tuning of the implementation described here has been left as future work.

The remainder of the dissertation is structured as follows. Chapter 2 provides the background of the project. The implementation of the Random ByteIO interface is given in chapter 3 and chapter 4 discusses how the service was connected to the OGSA-DAI middleware. Chapter 5 describes the client that was developed for interacting with the new service and the implementations of the basic use case scenarios that have been identified by the ByteIO working group. Chapter 6 describes a provisional real use case scenario that incorporates most of the implementations that are presented in chapter 5 to illustrate the potentials of OGSA-DAI when this supports the Random ByteIO interface. Chapter 7 discusses the tests that have been carried out for assuring that the written code meets all the desired properties (i.e. correctness, good quality) and chapter 8 describes additional interoperability tests that have not been the primary aim of this project, but have been found essential for completing this. Finally in chapter 9 conclusions are drawn about the project and future work is suggested.
Chapter 2

Background

The motivation for this project is the need that OGSA-DAI supports a new standard interface that has been proposed by the Global Grid Forum for accessing data from data sources in the Grid in an effective, flexible and efficient manner.

This chapter sets the context of the problem of data access in the Grid and gives reasons why a new mechanism is needed. OGSA-DAI is introduced, along with its existing mechanisms for accessing data. Finally, the new Random ByteIO interface is described and the benefits from connecting this to OGSA-DAI are illustrated by providing some use case scenarios.

2.1 The problem behind Data Access in the Grid

The Grid is a hardware and software infrastructure that is used for solving computational and data intensive applications by providing seamless access to geographically distributed computing machines, storage facilities, visualization equipment and scientific instruments [1]. Data, software and hardware in it are made available via remote resource[^1] and these can be shared among the end-users.

Client applications should be able to connect to the Grid and access resources as though they belonged to a single large computer, while the underlying complexity will be hidden by middleware technologies. Experience however has shown that this is a complex undertaking, merely due to the diversity of the resources and their distributed nature across geographical locations and organizations.

Seamless data access is one aspect of the full Grid vision, as this is described in more detail here [2]. As its name suggests, this concerns accessing resources that produce or consume data, termed as data sources or data sinks respectively. Data are usually organized and shared as structured collections held in databases, XML documents or

[^1]: The term "resource" is used to "denote any capability that may be shared or exploited in a networked environment" [9].
flat binary files and can be physically distributed across different organizations. Their size increases fast and can grow significantly large, to the extent of several terabytes or petabytes soon. Maintaining and managing such amount of distributed data raises big challenges for the Grid developing community. An attempt has been made to list and analyze these here [7].

Applications will need to access data from multiple data resources and request that complex series of processing steps are performed on these. The output data may be too large to store inside the users’ machines and these will need to be analyzed, visualized, archived, or replicated. What is more, data will need to be transferred across different locations while the processing takes place, integrated with other data, transformed and stored.

Sophisticated mechanisms will need to exist that will allow accessing data in an integrated and efficient manner. The mechanisms will need to be flexible and efficient to accommodate for the large-scale, distributed nature of the data. They will also need to be standard, so that applications can access resources in a uniform way, irrespectively of data formats and local data management systems (e.g. database systems, file systems).

OGSA-DAI is a middleware that provides mechanisms for accessing data over the Grid in an integrated fashion. This is described next and reasons are given why this needs to be extended to satisfy the requirements that have been set above.

### 2.2 OGSA-DAI

This section describes the Open Grid Services Architecture - Data Access and Integration (OGSA-DAI) project [23]. OGSA-DAI is a middleware architecture that uses web services technology to provide uniform access to data sources over the Grid [3]. The architecture is based on standard interfaces that are being specified by the Global Grid Forum to ensure that data resources will be made available to the users via a set of standard mechanisms.

OGSA-DAI has primarily focused on providing remote access and integration to structured collections of data, such as relational and XML databases, although its interface allows to easily extend it to support other data types as well. Databases are exposed to the Grid via web services that allow manipulating their data in a consistent way, regardless of the underlying technology (i.e. local database management systems). Users can request that multiple operations are performed on the data by submitting workflows of activities to an OGSA-DAI execution service that is responsible for contacting the data resources and scheduling operations that manipulate their data. The term “activity” is used to denote a particular action the user wishes to be performed on the server-side and can vary from a data resource manipulation (i.e. query or update) to a data transformation (e.g. data compression) or data delivery to some third party. Moreover, activities can be connected together by forwarding the output data of one to the input of the another to form data workflows for requesting that a complex series of processing steps
are executed on the server-side. Following the execution of the activity workflow, a response message is sent back to the client, containing the status of the execution and the returned data if the user requested any.

An obvious advantage of OGSA-DAI’s activity framework is that this way computation is moved closer to the data and the number of messages that need to be exchanged between server and client for performing a series of operations is reduced [2]. What is more, the operation-specific OGSA-DAI activities serve as plug-ins for adding new capabilities to the OGSA-DAI middleware without needing to change its core source. This is also facilitated by the nature of the users’ requests, which describe exactly the activities that they wish to be executed on the server-side, and hence these can be treated as individual modules.

Since clients and server interact via web services, request and response messages will need to be exchanged using the SOAP [24] communication protocol. As such, the exchanged data will need to be transferred through the wire in XML format and this adds more processing for encoding and decoding the messages on the sender and the receiver respectively. OGSA-DAI hides this complexity by providing an abstraction for the creation of request and response messages (i.e. Java API), which is known as Client Toolkit. For each server-side activity available in OGSA-DAI, there exists a corresponding client-side activity that helps the user include the former in the request message.

In summary, OGSA-DAI is a high-level extensible framework for seamless access and integration of data across heterogeneous data resources over the Grid using web services. The current project aims to extend OGSA-DAI by implementing and adding new activities that will let accessing data resources in efficient ways, using a new standard interface that GGF has specified. Client-side activities will also be created to allow clients utilize the corresponding server-side ones. Next section gives an overview of the access mechanisms (i.e. activity framework) that OGSA-DAI supports at the moment and reveals the gaps that the new interface is meant to fill in.

### 2.2.1 Existing OGSA-DAI mechanisms for accessing data

Using existing OGSA-DAI mechanisms, a user can submit a workflow of activities to the server to request that data are fetched from some remote data resource (e.g. relational database), manipulated and delivered to some recipient service, which may be the same user or a third party. While processing the workflow, OGSA-DAI will attempt to stream the data through a pipeline of activities until these reach their final destination. This allows the activities to execute concurrently on different threads while each one processes a different block of data. Figure 2.1 illustrates the process.

---

2i.e. activities that are connected together to form data workflows.
The design is efficient since it allows parallelizing the execution of the workflow by dividing the latter into several tasks (i.e. activities) and executing these on different threads. It is also flexible, since a user may request that additional operations are performed on the data, by simply adding more activities between the first and the last one in the workflow. As such it can be considered as the optimal solution for handling data workflows over the Grid and leaves to the access mechanisms to decide how they will best exploit its potentials according to the users’ requests.

One drawback of the current OGSA-DAI access mechanisms is that they do not support partial data access. In other words, a client application cannot request that only a subset of a data resource flows through the pipeline and in cases that the whole amount of data are not of interest to the former, this can lead to unnecessary data processing and transfers that can stall the pipeline and downgrade the overall performance. As one would have expected, the problem becomes even worse when the size of the data resource grows a lot larger than the amount of data that needs to be accessed in it. Combining this with the nature of the Grid, where large-scale and distributed data are common, the current mechanisms are considered highly inflexible and inefficient.

Another issue about the existing access mechanisms that has been identified concerns the way data are delivered to recipient processes. Currently, data can be delivered either inside SOAP messages or by utilizing the FTP or SMTP protocols. In the first case, the data are inserted to the body of the SOAP message by encoding these into XML. This is considered as a highly inefficient way for transferring large amounts of binary data, since these will need to be encoded into an ASCII text format, known as "Base64", before they can be added inside the SOAP message body. Consequently, significant time is lost during the encoding and decoding of the data on the sender and the receiver processes respectively. What is more, the encoded data are expected to be

---

3More complex workflows that integrate data from many data sources can also be constructed, but how this is achieved is not the purpose of this document.

4An access mechanism can be thought of as an action that schedules operations on the data workflow to be executed on behalf of the user.
approximately 35% larger than the original binary data, resulting in increased size of transfers. To conclude, transferring large amounts of binary data inside SOAP messages can be highly inefficient and should be avoided whenever possible.

A much more suitable way for transferring large binary data would have been to use the FTP protocol\(^5\) since data need not be encoded in text format in this case but can be transferred as bytes inside TCP packets. However, FTP is considered non-flexible since it cannot transfer subsets of data (and so does SMTP). What is more, an FTP server will need to be available on the client-side and listening to incoming messages. Data will be pushed from the OGSA-DAI server to the FTP and the client has no direct control over the transfer. The same applies to the SMTP protocol. Additionally, data are pushed to the receiver in an asynchronous manner in the case of SMTP \[^8\]. That is, the receiving process has no explicit control over the transfer and cannot know when a message will be delivered. For all reasons described here, transferring large amounts of binary data via FTP or SMTP is considered inflexible, inefficient and also contradicts in terms with the Grid vision that has been set in section 2.1 and which requires that resources are accessed seamlessly, as if they were local to the client application.

Next section introduces a new interface for data access that is expected to address the problems that have been identified here for the current access mechanisms of OGSA-DAI.

### 2.3 A new mechanism for accessing data

A new interface has been specified by the Data Access and Integration Services (DAIS) working group of GGF, named "ByteIO", to provide more flexibility in data access over the Grid and enhanced performance. This supports operations for defining and accessing chunks from data resources at the byte level. As such, it is more suitable for accessing binary data rather than character or other types of structured data. What is more, the operations can be easily mapped to existing APIs, such as the POSIX API, to allow easier adoption of the interface by existing code. The interface has been especially designed for achieving maximum performance when accessing large amounts of bulk binary data by letting users choose the transfer mechanism that is most efficient for their data access scenario. An implementation of the interface can support arbitrary transfer mechanisms by simply plugging these into it and advertising them to the users. Furthermore, the "plug-in" nature of the transfer mechanisms allows to easily extend the implementation to support more efficient mechanisms at any time in the future.

The interface can clearly fill in the gaps that have been identified in the current OGSA-DAI version. New OGSA-DAI access mechanisms can be constructed that would utilize its features to allow users placing chunks of data resources inside a pipeline of activities.

\(^{5}\)Although one would have expected the GridFTP protocol to be preferred in this case, since it is built on top of FTP and is more flexible, efficient and secure. However, the current OGSA-DAI version does not support this.
for processing in various ways. This will also provide a solution to the problem of inefficient and non-flexible data delivery. To illustrate how this might work in practice, a new OGSA-DAI activity can be constructed that would be initialized with the details that are needed to perform a ByteIO specific operation on the data (e.g. read a subset of a data resource) and added into an existing workflow of activities, like the one of figure 2.1. When OGSA-DAI executes the activity, this can read a subset of a data resource and forward the data to the next activity in the workflow for manipulating these in various ways.

The example shows that the new mechanism does not replace the data workflow model of OGSA-DAI, but allows to utilize this in more efficient ways, by eliminating existing bottlenecks (e.g. inefficient transfers of data).

The ByteIO interface is divided into two parts, namely the "Random ByteIO" and the "Streamable ByteIO" interfaces. Both allow accessing blocks of bytes from data resources, but they are targetted in different use case scenarios. As their names suggest, the first provides operations for accessing chunks of data in a random way (i.e. data can be skipped while the data resource is accessed), whereas the second one can be used for reading or writing blocks from a stream.

The current project aimed at the implementation of the first interface only and hence later we will be referring to this alone. Next, the discussion steps into the details of the Random ByteIO interface to illustrate how this manages to satisfy its goals in practice. Most of the things that are presented there derive from the interface’s specification, which can be found online at [6].

2.3.1 The Random ByteIO interface

The specification describes a stateful web service (i.e. grid service) that can be used by data providers to expose their data to the Grid. According to this, the data are encapsulated inside stateful WS-Resources that can be made publically available via their unique identifiers. WS-Resources are key components of Grid services and in our case they serve as virtual data resources that provide an abstraction of the actual data resources and also support a number of attributes for describing these. A Random ByteIO WS-Resource comprises a set of resource properties (i.e. attributes) that form its exposed state and one or more web services that provide a particular interface (i.e. the Random ByteIO) for manipulating these. The service components are described in more detail later in the chapter and in section 3.

Clients can contact any Random ByteIO web service directly to request that they read or write data from the resources in a seamless manner, while differences (e.g. how data are managed on the remote servers) will be hidden by the server-side resource implementations. The direct access to the resources implies that the interface is not OGSA-DAI specific but can be implemented on top of any technology that provides access to data sources over the Grid. Since this is also a standard interface, it can facilitate the integration of data across different middleware systems (e.g. OGSA-DAI,
UNICORE, GLOBUS, etc.) and help reaching closer to the full seamless data access vision of the Grid.

### 2.3.2 Web Service Description

The ByteIO authors have published a complete WSDL document describing the Random ByteIO Grid Service. The Web Service Description Language (WSDL) is an XML grammar used for describing the exposed interface of a web service and the way to access this [27]. The document describes all messages that can be exchanged between clients and service providers in terms of requests and responses and the datatypes of their arguments as well. Since the Random ByteIO is a Grid Service, its WSDL needs to comply to the WSRF specifications [28] and as such the resource properties that constitute the resource-state and the operations for accessing this are also specified. Finally, a WSDL document would also define how messages will be represented on a particular protocol (e.g. HTTP, SMTP, etc.) and the location (i.e. URL) of the web service as well. However, the last two definitions are missing from the given WSDL and it has been left to the implementors to decide how messages will be exchanged most efficiently and securely in their implementations.

The WSRF specifications recommend that the **factory** pattern is used for creating and accessing multiple resources in the Grid [29]. According to this, for every type of resource there should be two web services available. One for managing the resource’s lifecycle (e.g. creation, destruction), and another for manipulating its state. The Random ByteIO specification only defines the second service, leaving to the implementors to decide how resource lifecycle will be dealt with most efficiently in their implementations. In our case, resource creation is performed by using the OGSA-DAI framework. This is described in chapter [4].

#### Operations of the Random ByteIO

The interface provides a set of operations for manipulating the data contained inside the resources. In particular, a client can read or write blocks of bytes by invoking one of the following operations:

1. `read(startOffset: unsignedLong, bytesPerBlock: unsignedInt, numBlocks: unsignedInt, stride: long): byte[]`
2. `write(startOffset: unsignedLong, stride: long, data: byte[]): void`
3. `append(data: byte[]): void`
4. `truncAppend(offset: unsignedLong, data: byte[]): void`

The operations provide flexible ways to access the resource-specific contents in a file-like way. Each is accompanied by a number of arguments that let the client specify
the size and positions of the blocks of bytes it wishes to read or write. The startOffset argument denotes the offset into the data resource the client wishes to start reading or writing data. Since data will be accessed in blocks, the client will also need to specify the number of bytes per block (i.e. bytesPerBlock) it wishes to access, the number of blocks (i.e. numBlocks) and the distance in number of bytes between the starting positions of neighbour blocks (i.e. stride). Figure 2.2 illustrates how the arguments can be used to access blocks of bytes from a resource.

Figure 2.2: Illustration of the arguments that are used by the operations of the Random ByteIO interface to specify subsets of data resources.

The operations allow accessing blocks of data in a POSIX-like way, making the resources look to the client application as if they were local files. This facilitates the provision of access transparency, which is much of the promise of the Grid and certainly a great need if existing applications that have no prior knowledge of the Grid are to be connected to it.

The Random ByteIO Resource Properties

The Random ByteIO resources comprise the following resource properties (i.e. attributes) for describing the data resources they represent. These will form the exposed state of every resource.

Size The total Size in bytes of the Random ByteIO resource
Readable A boolean indicating whether the resource is readable or not
Writable A boolean indicating whether the resource is writable or not
TransferMechanism The transfer mechanisms that are supported by the resource
CreateTime The time the resource was created
ModificationTime The time the resource was last modified
AccessTime The time the resource was last accessed
Resource properties can be accessed via a set of standard operations that are defined in the Web Services Resource Framework (WSRF) specifications. These are listed below:

1. `getResourceProperty(QName resourceProperty)` Returns the value of the specified resource property.

2. `queryResourceProperties(Expression query)` Use a query expression, such as XPath, to query the resource properties document.

3. `getMultipleResourceProperties(QName[] resourceProperties)` Returns the values of the resource properties specified.

Since these are standard operations that are supported by any grid service, their implementations can be reused for building the Random ByteIO service as well. In particular, the Random ByteIO service makes use of OGSA-DAI’s RP implementations, as it will be shown in chapter 4.

### 2.3.3 Use case scenarios for the new mechanism

The applications that are expected to benefit most from the connection of the Random ByteIO interface to OGSA-DAI are the ones that need to access and integrate large amounts of distributed binary data. Non surprisingly, such applications exist both in the industrial and scientific universes. A number of real use case scenarios are discussed next that illustrate the imperative for mechanisms that allow partial random data access to exist and be connected to OGSA-DAI.

**High Energy Physics**

The Large Hadron Collider at CERN is expected to commence its operation the year 2007 and produce data at a rate of 1.6GBytes/sec or a total of 10 PetaBytes per year [30]. The data will be of different formats and stored inside relational databases, XML databases or flat binary files. Scientists will need to analyze these for investigating many fundamental scientific problems, such as the possible origin of mass in the universe. Furthermore, the analysis will comprise many processing steps that will generate intermediate data for storing, transferring and analyzing again. This is a pure evidence that mechanisms that will allow accessing and integrating subsets of data resources are essential. Using the Random ByteIO interface and OGSA-DAI, a scientist may submit complex data workflows for requesting that subsets of data resources (e.g. parts of a large images) are read, integrated, processed by other OGSA-DAI activities (i.e. compressed) and delivered to him. Processing parts of data rather than the whole amount of this is expected to give faster results and increased efficiency.
Bioinformatics

The genetic sequence (DNA) of thousands of organisms is being decoded and stored inside geographically distributed structured collections. The size of the data is vast and increases exponentially at a rate of one genome sequence every 10 seconds [9]. Bioinformatics focuses on solving problems such as gene discovery and protein folding by applying operations on the DNA data. The key idea here is that the DNA sequence can be divided into many parts that are called genes and each one contains code that maps to a particular protein sequence. Knowing the protein that can cure some disease, scientists will need to discover the gene that encodes it so that the protein can be produced in large quantities and provide medicines. However, the DNA sequence is huge and can contain 20,000 to 25,000 genes [11]. As such, complex and computationally intensive operations will need to be applied to the distributed DNA data, comprising numerous comparisons between the protein and gene sequences until the right combination is found. The use case proves that gene discovery can benefit from mechanisms that will allow accessing parts of the DNA rather than the complete data. The Random ByteIO interface can be used to access the DNA sequence randomly at the server-side and pass the subsets of data to various processing OGSA-DAI activities to perform combinations between gene and protein sequences. Moreover, OGSA-DAI will allow integrating data from many databases, so that more complex workflows can be performed on the subsets of data. The data need not be transferred to the user, but these can be processed and visualized in the server-side and since the activities will be performed on small subsets of data these will complete faster, resulting in enhanced performance and less time to solution.
Chapter 3

Implementing the Random ByteIO interface

A stateful, WSRF-compliant Random ByteIO Grid Service was developed that allows users to manage and access multiple Random ByteIO resources at the server-side. The service consists of a stateless frontend that exposes the operations that are provided by the interface to the clients and a stateful backend that comprises the Random ByteIO WS-Resources, where the data and resource properties (i.e. the resource state) are kept. Client requests will be forwarded by the frontend service to the appropriate resources so that their data can be accessed. As such, the actual implementation of the operations that the Random ByteIO interface provides was placed inside the resources. These will be made publically available to the clients via unique identifiers that are called endpoint references in the WSRF lingo and encapsulate the id of the resource that is to be accessed and the URL of the frontend service that manages this (i.e. forwards requests to it). Upon receiving a request, the service looks into the presented endpoint reference and fetches the appropriate resource from some repository (see section 5.1 for more). From then on, the service may access the resource on behalf of the client to read or write data. A response is finally returned back to the client when the operation completes. Figure 3.1 illustrates this process.

The implementation of the Random ByteIO interface was accomplished in two stages. During the first, the operations that are provided by the interface were implemented and tested for accessing parts of plain files that resided on the server. For this purpose, a Web Service was developed that could accept client requests for accessing server-side data using the Random ByteIO interface. The service had no notion of WS-Resources at that stage. Later, the previous implementation was connected to OGSA-DAI by utilizing its resource framework for constructing and managing the Random ByteIO WS-Resources. The "connection" stage is described in chapter 4 since this involves additional complexity and cannot be understood without explaining the OGSA-DAI development framework for Grid Services. This chapter describes the implementation of the operations alone, and for this purpose it is assumed for the time being that the Random ByteIO resources are already available to the service and that this can fetch
them from the repository and access their state.

![Diagram of invoking a Read operation on the Random ByteIO service](image)

Figure 3.1: Invoking a Read operation on the Random ByteIO service

The implementation of the operations was based on a number of design decisions that allowed for correctness and maximum performance. These are described next for setting the background of the implementation.

### 3.1 File Access mechanism

It has already been noted that the Random ByteIO resources will store their data inside files and access these via the operations of the interface. There is a special reason that justifies the decision. In subsection 2.3.1 it was stated that the Random ByteIO resources serve as abstractions of the actual data sources they are associated with for manipulating their data. The resources will be able to provide random access to the latter only when they have access to the full data sources (e.g. a user may request accessing the first and last byte of a data source). Since this is the case, instead of temporarily copying the data source to the server each time an operation of the interface is invoked for accessing random blocks of bytes, which would have been very inefficient, the data source is copied "inside" the Random ByteIO resource when this is initialized and is
kept there. For this purpose, each resource owns a unique file in the server’s file system for keeping the data that should make available via the operations of the Random ByteIO interface.

Since data will be accessed from local files in the server, the implementations of the Random ByteIO operations will need to utilize the I/O mechanisms that the chosen programming language supports, and which in our case was Java 1.5. Java supports a large number of I/O mechanisms to allow developers choose the one that suits best their needs and write efficient code fast and easy. These are described in more detail and compared with each other in chapter B of the Appendix. As it was described in the previous chapter, achieving excellent performance is one of the primary goals of the ByteIO interface and as such, care has been taken to keep the overheads of the implementation down to a minimum by choosing the most appropriate access mechanism for implementing the operations.

From the Java I/O mechanisms that are presented in the Appendix, the Random Access Files one appears to map naturally to the problem that needs to be addressed, and was finally chosen to build the interface implementation on top of. There are three main reasons that justify this choice.

1. Data will be accessed from files located in the server’s file system.
2. The interface will need to provide both upstream and downstream access to the resource-specific files.
3. The Random ByteIO operations will work on blocks of data that may not be contiguously stored inside the files.

The Random Access Files I/O allows accessing a file both upstream and downstream by placing the file pointer at the appropriate position into the file. Upstream file access is not very straightforward to achieve using streams for I/O, since a new stream would need to be constructed for accessing data from an earlier position in the file and the old stream be closed. The construction of a new stream would mean the closing and reopening of the file and given that this action would be repeated many times, it would consume lots of system resources and the code’s performance is expected to downgrade significantly.

One of the main drivers that led to the creation of the Random ByteIO interface was the need for accessing chunks of data from large data collections. The Random ByteIO interface will need to provide a way to skip data the client is not interested in and perform operations on the rest. The file pointer of the Random Access File mechanism allows to quickly navigate to the starting position of the required data inside the resource-specific file and access these while any unwanted data are skipped. Chunks of data can be this way extracted or inserted into the file without loading its whole data into the server’s main memory. In the case of a streamable I/O on the other hand, the whole stream will need to be traversed until the required blocks of data are reached, resulting in poor performance, especially when dealing with very large files.
A drawback of the Random Access File mechanism is that it is unbuffered and this can result in poor performance when large size of data is accessed. The Java core does not currently provide any wrapper implementation for a buffered Random Access File mechanism and therefore the only solution to the problem is to implement our own wrapper class. This is discussed in more detail in subsection 5.2.1 under the "performance investigations" paragraph.

Finally, it should be noted that OGSA-DAI needs to be a scalable server, able to collect requests from various clients and satisfy these concurrently. That is, the Random ByteIO operations must be non-blocking, allowing different users to access different data at the same time. The Java NIO packages that are discussed in the Appendix allow to implement non-blocking memory accesses. However, this will not be needed in our case since Apache Tomcat [31], the web services container that will be used for deploying the Random ByteIO service into, can serve multiple client requests concurrently by executing these in separate threads automatically. In addition, the implementation will need to ensure that concurrent accesses to the same resources will not cause any inconsistencies. In other words, this needs to be thread-safe. This was accomplished by synchronizing the threads that accessed the same shared data. This is discussed in more detail in the next section.

3.2 Thread Safety

The implementation was made thread-safe by synchronizing the threads that access same Random ByteIO resources with locks. Locks can be used to assure that only one thread at a given time (i.e. the one that owns the lock) will be accessing shared data. Currently, JDK 1.5 offers two ways to implement locks, one by using the synchronized facility and another by using ReentrantLocks. The latter provides extended functionality for managing the locks in more flexible ways and is especially useful for writing high performance code. However, it was finally decided to use the synchronized mechanism to implement the locks in the code. The reasons that justify the decision are given below:

1. A thread that needs to access the data or state of a Random ByteIO resource that its lock is already taken by some other thread, will not have any other processing to do, but will need to block until the lock is released. Using the synchronized facility, each thread that requires a lock that is already taken will block anyway.

2. The operations "Read", "Write", "Append" and "Truncappend" of the Random ByteIO interface will need to be thread safe and therefore, when a thread invokes any of these it will need to acquire the lock at the very first start of the operation and release this in the end. Using the synchronized facility, the whole operation (method) will be locked automatically.

3. The synchronized facility lets the Java virtual machine manage the locks and hides the complexity of how this can be achieved. This way, it is also ensured that a
thread will release it lock even when an unexpected anomaly appears during the runtime.

Since each resource needs to have its own lock so that threads will not synchronize when accessing different resources, the *synchronized* keyword was added in the declarations of the methods that needed to be made thread safe. This automatically instantiates a lock\(^1\) together with the instantiation of the particular resource. Subsequent operations that target the resource will first need to acquire the particular lock instance before they can access any shared data.

A thread will busy-wait until it acquires the lock and then enter the critical section to read or write data and update the resource properties to reflect the new resource state. The lock is automatically released by the Java virtual machine when the thread exits the *synchronized* method.

### 3.3 The Read operation

A client may retrieve blocks of bulk data from a Random ByteIO resource by invoking the *read* operation. The operation expects as arguments the starting offset inside the resource’s data file, the number of bytes per block the user wishes to read, the number of blocks, the stride between blocks and finally the requested data transfer mechanism, which the resource must support. Furthermore, the operation needs to be thread safe, meaning that no other thread should be able to access the resource while this is being read.

Upon receiving the client’s request and fetching the targeted resource, the service invokes a *read()* method that is contained inside the implementation class of the Random ByteIO resource for reading data from the resource’s data file. The retrieved data are stored inside a binary array and these are returned to the client using the preferred transfer mechanism. The binary array is allowed to contain less amount of data than that requested when the size of the resource is not large enough. The service also updates any resource properties needed and throws appropriate exceptions, according to the Random ByteIO specification, when failures occur.

Figure 3.2 illustrates the process of reading some blocks of data from a Random ByteIO resource.

The service first checks whether the resource exists, is readable and supports the requested transfer mechanism. If not, it throws appropriate exceptions. In any other case it invokes the read method that is implemented by the resource class to retrieve a binary array with the requested data. A returned array of size 0 indicates that the starting offset is greater than the size of the resource.

The Java programming language allows only static arrays to be declared, meaning that their size cannot change after these are initialized. This adds more complexity to the

\(^1\)This is hidden from the programmer by the Java virtual machine.
implementation, since when short reads are being used it is not straightforward to calculate the size of the final binary array a priori.

The simplest solution to this problem would be to first create an array X with size equal to the total number of bytes requested. Then, after all requested data is read from file, in case the array X contains less data that its size allows, a new array Y can be created with size equal to the number of bytes that were actually read into X, and copy the data from X to Y. Finally, the array Y is returned to the client and X is garbage collected. Although this solution is simple to implement, it is not very efficient since it consumes a lot of memory. At first glance, two arrays are being used for storing the same piece of data and the latter will need to be copied from one array to the other. Furthermore, when an array is initialized in Java, its elements are set to zero and a place in memory is reserved for storing these. When the size of the final array Y is much smaller than the array X, then it is obvious that a lot of memory space is reserved for X for no practical reason (e.g. imagine the size of X is 100Mbytes and Y is 10Kbytes).

Another possible solution to the problem would have been to use Java’s collection types, such as Vectors, or ArrayLists. These are structures that allow collecting and manipulating objects. An ArrayList for instance allows resizing it dynamically. However, the
information in this case is not stored sequentially in memory and therefore the solution is expected to be highly inefficient. Moreover, each byte would need to be wrapped inside a Byte object before this is stored in the collection, and this will slow down the run even more.

The solution that was finally decided and implemented was to initialize a binary array with the exact size of data that will be returned to the client before these are read from the file. This was made possible thanks to the "Size" resource property that holds the size of data that are made available by the resource. Combining this with the details of the read request (i.e. the operation’s arguments), it is possible to calculate exactly the number of bytes that will be read from the resource. This solution is expected to be most efficient, since it uses only one binary array that contains only useful data. Comparing with the previously described solutions, no time is lost while copying data from one array to another, no memory is wasted on unused space in the array and data is stored sequentially in memory, allowing for various compiler optimizations.

Blocks of data are read cyclically and stored inside the array until this gets full. Each time a block of bytes is read from file, the file pointer is moved to the starting position of the next block, according to the stride value. What is more, the stride can be either positive or negative.

The operation was made thread safe by using the synchronized facility that was explained in section \[3.2\]. Upon successful completion of the read operation, the AccessTime resource property is updated.

### 3.4 The Write operation

A client can invoke the write operation to store blocks of data at given offsets inside a Random ByteIO resource. In addition to the requested data, the client will need to supply the starting offset at which to begin writing the blocks of data in the resource, the number of bytes that will need to be written per block, the stride between blocks and finally the preferred data transfer mechanism. The write operation will need to be thread safe and synchronized blocks are being used for this purpose. The operation is illustrated in figure \[3.3\].

Upon receiving a write request, the service will retrieve the target resource and poll its resource properties to check whether this is writable and supports the transfer mechanism requested. In case it does, then it will invoke the write() method implemented by the resource class and pass the client’s request to it.

Similarly to the read operation, the blocks of data are written cyclically until there are no more data to write. Every time a block is recorded, the file pointer is shifted to the starting position of the next block according to the value of the stride. Blocks are allowed to overlap when the stride is less than the size of a single block, or leave "holes" in the file when the stride is greater. The "holes" may be existing data that are skipped
during the write operation or data in unknown state when the file is extended. The written data replace any existing data that are stored in the same location in the file. What is more, the stride can take either positive or negative values, allowing to write a sequence of data blocks in ascending or descending absolute offset order respectively. In case the stride is negative and the start of file is reached during the write process, the service throws a *CustomFault* that contains the number of bytes that were recorded before the exception was raised, as shown in figure 3.3. Clients may choose to ignore this fault or process it so that they know exactly how many bytes were eventually stored inside the resource.

Upon successful completion of the write operation, the "Size", "AccessTime" and "ModificationTime" resource properties are updated to reflect the new resource state.

![Figure 3.3: The Write Operation](image)

### 3.5 The Append operation

The *append* operation allows appending data to the end of the Random ByteIO resource. Arguments to the operation are the data to be appended and the preferred transfer mechanism. Similarly to the write operation, upon receiving the client request, the service will fetch the resource and check whether this is writable and supports the transfer mechanism requested. It will then invoke the *append()* method implemented by the resource class to write the data inside. Before any data are stored into the resource, the
file pointer will need to be shifted to the end of the file to mark the starting location of the new data.

The operation needs to be thread safe and therefore any thread must acquire the resource-specific lock before it appends any data to the resource. Finally, the "Size", "AccessTime" and "ModificationTime" resource properties are updated, similarly to the write operation.

### 3.6 The Truncate-Append operation

The *truncate-append* operation allows to truncate a Random ByteIO resource to a given offset and optionally append data to the end of the truncated resource. Arguments to the operation are the offset to which to truncate the resource and optionally the data to append and the transfer mechanism to use. Upon receiving the client’s request, the service fetches the resource and checks whether this is writable and supports the requested transfer mechanism. It then invokes the `truncappend()` method, implemented by the resource class. This waits until the resource’s locks is acquired and proceeds to truncate the resource-specific file by invoking the `setLength()` method of the `RandomAccessFile` class, passing the requested offset value as a parameter. A `TruncateNotPermitted` fault is thrown if this operation fails. After the resource is truncated, the service checks whether the client included any data to append inside the request message. In case it finds any, these are stored inside the resource by invoking its `append()` method that was described before. The operation updates the "Size", "AccessTime" and "Modification-Time" resource properties. Figure 3.4 illustrates how the operations works.
3.7 Bindings

It has already been noted that the given WSDL document does not mandate any particular way messages should be represented on the communication protocol. It was decided to use the "document/literal" style and "HTTP" for transferring the data. There are two reasons that document/literal style was used. Firstly, this style is WS-I compliant, meaning that it is supported by the Web Services Interoperability organization [16]. This increases the chances that our web service product will be interoperable with other Random ByteIO implementations. The importance of interoperability is described further in chapter [8]. Secondly, the design of the given WSDL indicates that this is the preferred style to use, since there are no <type> declarations inside the "message" XML elements, but these are declared before they are inserted into the messages instead. This is explained in more detail here [17].
Chapter 4

Connecting the Random ByteIO Service to OGSA-DAI

At the second stage of the implementation, the Random ByteIO service that was described in the previous chapter was connected to the OGSA-DAI middleware so that the latter supports a standard mechanism for accessing parts of data from sources or sinks in the Grid. Combined with the existing OGSA-DAI access mechanisms, this allows for more flexibility and enhanced efficiency in data resource access and integration. Utilizing this new functionality would comprise the following steps:

1. Request that a new Random ByteIO resource is created and deployed in the OGSA-DAI server using the latter’s resource framework.

2. Load the Random ByteIO resource with the required data by submitting OGSA-DAI data workflows that are constructed from existing and new activities that allow accessing Random ByteIO resources.

3. Access data from the resource, either by invoking operations of the Random ByteIO service directly, or by submitting OGSA-DAI data workflows.

A set of web services forms the presentation layer of the OGSA-DAI middleware. This encapsulates the functionality that is required for exposing the resources that reside in the server using web services. For each type of resource there is an associated service that manages its state and data. Since the Random ByteIO interface defines a new type of resource, with its own resource properties and data, a new Random ByteIO service had to be connected to the presentation layer of OGSA-DAI for managing the new resource. Figure 4.1 shows how the Random ByteIO implementation relates to the existing software components of OGSA-DAI 3.0.
The OGSA-DAI core services are summarized below.

**Data Request Execution Service (DRES)** Clients may submit requests to this service for performing either a data access, data integration or data transformation operation in the OGSA-DAI server. Requests are constructed from data-centric OGSA-DAI workflows and these are forwarded to the *Data Request Execution Resource* (DRER) so that they can be processed. Workflows that are executed by a DRER may interact with any available resource in OGSA-DAI for accessing its state and data, including the new Random ByteIO ones.

**Data Resource Information Service (DRIS)** The service provides operations for retrieving information about resources that reside in the OGSA-DAI server (i.e. their resource properties). In addition, the client can invoke the `resolve()` operation of the service to retrieve the endpoint reference of an existing resource. This can be useful for applications that are ignorant of OGSA-DAI and expect an endpoint reference for accessing a resource rather than the OGSA-DAI specific resource id.

**Session Management Service (SMS)** State and data can be stored across multiple re-
quests inside Session resources. These are managed by the SMS.

**Request Management Service (RMS)** Every time a request is submitted to a DRER, a new Request resource is created and managed by this service. The resource allows monitoring the execution of a data workflow and is especially useful in the case of asynchronous executions. For instance, a client may request that data are copied from a database into a Random ByteIO resource and check periodically in the future to see if the operation has been completed or not by invoking operations of the RMS that target the specific request resource.

It should be noted that OGSA-DAI contains two more services, named Data Source and Data Sink, which allow accessing data from their corresponding resources in a stream-like fashion. Their implementation is similar to that of the Random ByteIO since they are also used for accessing virtual data resources. However, the Random ByteIO differs in that it is a standard interface that allows data to be accessed randomly and not from streams.

The new activities that have been developed are described in section 4.3. Next, the discussion steps into the details of the OGSA-DAI architecture to illustrate the new service was finally "glued" to the existing code.

### 4.1 The OGSA-DAI Software

It has already been noted that OGSA-DAI is built on top of a standard interface that has been designed by the GGF and in particular the Database Access and Integration Services (DAIS) working group. Maintaining a well engineered interface is important for two main reasons, described in more detail here [2]. Firstly, it makes easier to expose databases to the Grid, by reusing code that is the same for every database implementation. Secondly, it allows clients accessing any data resource via the same set of operations, while the complexity of any particular data resource implementation is hidden in software.

Keeping the above in mind, care has been taken so that the implementation of the Random ByteIO service would not alter the OGSA-DAI core. This was successfully accomplished thanks to the sophisticated, object-oriented design of OGSA-DAI. For almost every class in source code, there is one or more interfaces that define its functionality. Although this must have been costly to design, it allows to easily extend the functionality of OGSA-DAI and gain from code reuse. What is more, interfaces ensure that the new code will integrate with the existing one.

OGSA-DAI comes in two flavours. The WSRF version of OGSA-DAI is compatible with the Globus 4.0 Toolkit’s implementation of WSRF, whereas the WS-I version is compatible with the UK OMII’s implementation of WS-I. The Random ByteIO service was decided to be built on top of the WS-I version of OGSA-DAI. The service was deployed into the Tomcat web services container and Axis 1.4 was used to deal with the
SOAP messages. The reason Globus 4.0 has not been used for building the grid service was that it ships with Axis 1.2RC2 and problems were identified when attempting to create the stub classes from the given WSDL with the WSDL2Java tool. It appeared that this version of Axis, although stable and reliable, does not address all JAX-RPC specifications that appear inside the given WSDL (in particular the xsd:any element) and consequently the WSDL-to-Java mapping was failing. The solution was to upgrade to Axis 1.4, which was compatible with version 3.0 of OGSA-DAI WS-I, but not with version 4.0 of Globus and hence OGSA-DAI WSRF.

OGSA-DAI WS-I provides a rich framework for building Grid Services that is similar to that of Globus. This allows for resource discovery (e.g. fetching resource from resource id), resource management (e.g. creation and initialization of resources) and resource access (e.g. resource properties access). Next follows a detailed description of how the Random ByteIO service was connected to OGSA-DAI and its framework used for resource manipulation.

4.2 The Random ByteIO Grid Service

The Random ByteIO implementation that has been described in chapter 3 was connected to the OGSA-DAI middleware by using its resource framework to accomplish resource management and access. The new Grid Service was deployed into the Tomcat web services container so that this can be accessed directly by any client via its endpoint address. This way, the service still remains independent from OGSA-DAI, meaning that a client does not need to have any knowledge of the latter’s existence to invoke any of the service’s operations. This is important since the Random ByteIO is hoped to become a standard service that arbitrary clients can talk to, irrespectively of the way this is implemented.

Any request to the service will be forwarded by the HTTP Server (i.e. Apache Tomcat) to the AxisRandomByteIOService implementation class for instantiating the former. This implements the interfaces ServiceLifecycle and Random ByteIO. The first provides the method init() that allows the JAX-RPC run-time system to initialize the service instance. The second interface provides the operations of the Random ByteIO interface that the service must make available to its clients for accessing the corresponding resources.

The service class passes any invocations that concern operations of the Random ByteIO interface to the associated provider class that implements these, called AxisRandomByteIOProvider. The requested resource is first retrieved by invoking the getResource() method of the provider class. This extracts the resource id from the request message and uses OGSA-DAI’s ResourceManager to retrieve the resource instance that corresponds to the presented id. In case the resource does not exist inside OGSA-DAI’s

1 The ResourceManager class provides operations for managing the OGSA-DAI’s resource repository.
resource repository, a ResourceUnknownFault is thrown. Otherwise, the requested operation is executed on the resource.

The service includes one more provider class that implements the standard WSRF-specific operations that are used for accessing resource properties. When a client invokes any of these operations, the service forwards the invocation to another provider class, called AxisWSResourcePropertiesByteIOProvider. In subsection 2.3.2, it was stated that the implementation of these operations is expected to be the same for every Grid Service that is built on top of the same Resource Framework (i.e. OGSA-DAI). In the case of the Random ByteIO however, this was partly true. The reason is that the WSDL document of the Random ByteIO service uses a newer specification for describing the operations that is slightly different to that of OGSA-DAI. In more detail, the namespace of the ByteIO WSRF-Resource Properties specification is:

- http://docs.oasis-open.org/wsrf/rpw-2

The namespaces point to different XML-Schemas that produce non-interoperable implementations. Luckily, the differences were minor. These are listed below:

1. The argument to the getMultipleResourceProperties() operation must be of type GetMultipleResourceProperties_Element in OGSA-DAI, and of type GetMultipleResourceProperties in ByteIO.
2. The argument to the queryResourceProperties() operation must be of type QueryResourceProperties_Element in OGSA-DAI, and of type QueryResourceProperties in ByteIO.

The problem was solved by reusing the operations’ implementation of OGSA-DAI, located in class AxisWSResourcePropertiesProvider and changing the types of arguments to the ones that were compatible with the Random ByteIO stubs. The solution was tested and worked successfully. More on testing can be found in chapter 7. Figure 4.4 presents the UML Class diagram of the service’s implementation.
4.2.1 The Random ByteIO resources

All known resources to the OGSA-DAI server are stored in a central repository, managed by a singleton object of the ResourceManager class. OGSA-DAI supports three types of resources that can be used to expose data in the Grid. These are the DataSourceResource, the DataSinkResource and the DataResource. For each type of resource OGSA-DAI provides interfaces to guide its implementation and ensure that the new code will integrate with the OGSA-DAI core.

The Random ByteIO resources were implemented on top of OGSA-DAI’s data resources, since these appear to map naturally to the needs of the implementation. The reason the other two types of resources were discarded was that they provide operations for accessing blocks of binary data in a stream-like fashion, which contradicts in terms with the random access requirement of the Random ByteIO.

Since the Random ByteIO resources will be constructed dynamically by the client, a new resource template had to be created. Upon receiving a request to create a new Random ByteIO resource, OGSA-DAI will use the template to load the configuration of the new
resource and append any resource-specific information to it, like the id and the creation time. This will result in a new resource-specific configuration file in the server’s file system. More details about this operation can be found in section 4.3. The resource template is presented in figure 4.3.

<table>
<thead>
<tr>
<th>RandomByteIO Resource Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>id=uk.org.ogsadai.RANDOM_BYTEIO_TEMPLATE</td>
</tr>
<tr>
<td>type=uk.org.ogsadai.DATA_RESOURCE</td>
</tr>
<tr>
<td>creationTime=null</td>
</tr>
<tr>
<td>terminationTime=null</td>
</tr>
<tr>
<td>PROPERTIES</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>CONFIG</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>ACTIVITIES</td>
</tr>
<tr>
<td>uk.org.ogsadai.WriteToRandomByteIOResource=uk.org.ogsadai.WriteToRandomByteIOResource</td>
</tr>
<tr>
<td>END</td>
</tr>
<tr>
<td>dataResourceClass=uk.org.ogsadai.resource.dataresource.rbyteio.RandomByteIODataResource</td>
</tr>
</tbody>
</table>

Figure 4.3: Template file for the Random ByteIO resource

The template contains the implementation class of the Random ByteIO resources and any activities they support. All Random ByteIO resource property values are subject to change dynamically while clients interact with the resources and therefore no static resource-specific properties are defined. Normally, a Random ByteIO resource would be constructed by the server automatically without the user needing to know anything about it and this is why the configuration section has been left empty. Moreover, the values of the identifiers id and creationTime will be overriden by the OGSA-DAI server during the creation of the resource.

Since OGSA-DAI supports various application-specific data resources (e.g. XML data resource, JDBC data resource, Random ByteIO data resource, etc.), it is important that it knows how to distinguish these and access their implementations. This is accomplished by including the name and package of the data resource’s implementation class inside the template. As soon as the resource is instantiated, the server will invoke the class’s initialize() method to initialize all resource properties that constitute its state. The current state and configuration of the resource can be retrieved by invoking the getState() method of the class.

Figure 4.4 presents the UML Class diagram of the Random ByteIO data resource implementation.
The state of the resource is managed by a separate class, named `SimpleRandomByteIODataResourceState`. This implements a resource-specific wrapper of the `DataResourceState` interface that allows including state configuration that the OGSA-DAI core is not aware of for the particular data resource (neither should be), such as the name of the file that will hold the data and its location in the server’s file system. The `initializeRandomByteIOProperties()` method of the class adds the resource properties inside the resource’s property set and sets their values. Figure 4.5 illustrates the process of creating and initializing a new Random ByteIO resource.

The persistence and configuration components of OGSA-DAI read the configuration of the resource and create a `DataResourceState` object containing this. The resource manager then retrieves the resource-specific implementation class from the state object and invokes its `initialize()` method passing to it the `DataResourceState` object as an argument. The latter is wrapped inside the implementation of the Random ByteIO-specific state wrapper and this constructs and initializes the resource properties by invoking the `initializeRandomByteIOProperties()` method.

Figure 4.4: UML Class Diagram for the Random ByteIO data resource
Resource Properties

OGSA-DAI supports two resource property types, namely the Persisted Resource Properties and the On-Demand Resource Properties. The main difference between the two is that the first are used for properties that tend to have static values (i.e. do not change very often), whereas the second ones are more suitable for properties that their values change dynamically in time. Obviously, the Random ByteIO resource properties had to be implemented on top of on-Demand Resource Properties, since the state of a Random ByteIO resource is expected to change every time the resource is accessed.

Each On-Demand Resource Property is given a reference to a callback class that manages its value and is responsible for storing and returning the newest one. The callback class that was created is called RandomByteIOCallback and this implements the On-DemandResourcePropertyCallback interface that provides the methods for getting and setting the resource property values.

Finally, the qualified name of each resource property is defined inside the interface RandomByteIOPropertyNames. This has the following format:

- \{http://schemas.ggf.org/byteio/2005/10/random-access\}PropertyName

Furthermore, each qualified name is wrapped inside an OGSA-DAI class called SimpleResourcePropertyName that facilitates the access of the former’s information.
4.3 New Activities

A number of new activities have been developed to allow the client accessing the Random ByteIO resources via OGSA-DAI. Activities are the base units of OGSA-DAI’s functionality, allowing to construct data-centric workflows that can be executed on a Data Request Execution Resource. The new activities were placed inside the package `uk.org.ogsadai.activity.rbyteio`.

A corresponding client-side activity has also been written for each new activity that was made available in the OGSA-DAI server. Client-side activities have been described in more detail in section 2.2. The new ones were placed inside the package `uk.org.ogsadai.client.toolkit.activities.rbyteio`.

4.3.1 The CreateRandomByteIOResource activity

A client can request that a new Random ByteIO resource is created via the `CreateRandomByteIOResource` activity. This will be added inside the workflow that will be submitted to the Data Request Execution Resource and therefore the latter has to support the new activity as well. The activity input and output specification is described in table 4.1.

<table>
<thead>
<tr>
<th>Inputs &amp; Outputs</th>
<th>Name</th>
<th>Type</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>RBYTEIO_ID</td>
<td>String</td>
<td>Yes</td>
</tr>
<tr>
<td>Output</td>
<td>output</td>
<td>String</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.1: Input and Output specification of the CreateRandomByteIOResource activity

The activity allows the user to specify a desired resource id for the new resource or leave this input blank so that OGSA-DAI chooses a unique identifier for it. Since the activity creates new data resources, it needs to implement the `ResourceFactoryActivity` and `ResourceManagerActivity` interfaces. The first provides a method to set a resource factory object, responsible for creating new data resources from resource templates and adding these to the OGSA-DAI server by accessing the Resource Manager. Access to the latter is also required so that a unique resource id can be returned to the activity in case the client has not specified any.

The newly created resource is configured according to the `uk.org.ogsadai.RANDOM_BYTEIO_TEMPLATE` resource template that was described in subsection 4.2.1. Before the resource is added to the server, it is customized by setting its unique id and file name that will hold the data. The particular file is also created by the activity and placed inside a temporary directory in the server. The file is named after the resource id so that it will increase the chances it will be unique. However, if another file with the same name exists in the server, an exception is thrown back to the client describing this. Upon successful completion of the activity the new resource id is written into the output so that the client can retrieve it using the client-toolkit.
Figure 4.6 illustrates the sequence of internal steps that are performed by the OGSA-DAI server when a new Random ByteIO resource is created.

**Client-side activity**

A client-side activity has been created to allow the client access the inputs and outputs of the activity described before in a simplified manner. Since the activity is not targeted at a particular resource, it extends the `BaseActivity` base class. A client can invoke the `addRandomByteIOID()` method of the client-side activity to pass the requested resource id to the appropriate activity input. Alternatively, in the case of activity workflows, the client can invoke the `connectRandomByteIOIDInput()` method to connect the output of some other activity to the input of the discussed activity. The resource id that is eventually created and recorded inside the activity’s output can be retrieved by invoking the `nextResult()` method, or connected to some other activity by invoking the `getOutput()` method that returns the first `SingleActivityOutput` object.

The code snippet of figure 4.7 shows how to use the client-side activity to construct a workflow that will create a new Random ByteIO resource.
RandomByteIO resource creation

//Create ByteIO resource
CreateRandomByteIOResource create = new CreateRandomByteIOResource(); (1)

if (!resourceID.equals("") { 
    ResourceID randomByteIOD = new SimpleResourceID(resourceID);
    create.addRandomByteIOID(randomByteIOD); (2)
}

//Deliver to Request status
DeliverToRequestStatus deliverToRequestStatus =
    new DeliverToRequestStatus();
deliverToRequestStatus.connectInput(create.getOutput()); (3)

//Construct workflow (4)
PipelineWorkflow createWorkflow = new PipelineWorkflow();
createWorkflow.add(create);
createWorkflow.add(deliverToRequestStatus);

Figure 4.7: Example of a workflow construction for creating a new Random ByteIO resource

(1) Create an instance of the Client-Side activity that will allow accessing the inputs and outputs of the server-side activity.
(2) In case the user has specified a particular resource ID, this is added to the activity’s input with the addRandomByteIOID() method.
(3) The activity’s output can be retrieved with the getOutput() method and connected to the DeliverToRequestStatus activity for monitoring the status of the workflow execution.
(4) Create the data workflow by adding both activities in it. This will be submitted to the Data Request Execution Resource.

4.3.2 The AppendToRandomByteIOResource activity

A Random ByteIO resource will probably be not much of a use if it does not contain any data. For the purpose of loading a newly created Random ByteIO resource with data, a new activity has been developed that targets the specific resource and must be supported by it, named AppendToRandomByteIOResourceActivity. The activity input and output specification is described in table 4.3.

Since the activity is targeted at a particular Random ByteIO resource, it needs to im-
Table 4.2: Input and Output specification of the AppendToRandomByteIO activity

<table>
<thead>
<tr>
<th>Inputs &amp; Outputs</th>
<th>Name</th>
<th>Type</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>INPUT</td>
<td>byte[]</td>
<td>No</td>
</tr>
</tbody>
</table>

To implement the ResourceActivity interface. This allows accessing the requested resource from inside the activity. In more detail, when the Activity Framework encounters an activity that implements the ResourceActivity interface, it invokes the interface’s set-TargetResourceAccessor() method to set the particular resource accessor. In the case of the Random ByteIO resource, this is the resource class that implements the Random ByteIO interface. Following this action, the activity can access the resource via the methods of its own class.

Input to the activity is a stream of bytes, represented by lists of byte arrays or BLOBs (Binary Large Objects). Using lists of byte arrays as input and output to the activity allows data to efficiently stream through the workflow. A block of binary data is read at every activity iteration and stored inside the Random ByteIO resource using the implementation of the Append operation that was described in the previous chapter. The latter ensures that the implementation of the activity is independent from that of the Random ByteIO resource, allowing the former to work with other Random ByteIO interface implementations as well.

Figure 4.8 illustrates how the activity works. Notice that the data is stored inside the resource in multiple activity iterations.

Finally, it is worth mentioning that the activity extends the RegularIterativeActivity base class, since data will be processed in blocks every iteration. Moreover, the block size was set to be 2048 bytes, the standard block size value the OGSA-DAI core uses.

**Client-side Activity**

Since the activity is targeted at a particular data resource, the client-side activity needs to implement the ResourceActivity interface and extend the BaseResourceActivity base class. These will mainly allow to access the id of the particular resource the activity concerns. Furthermore, the client-side activity allows to write binary data into the data input in two ways. Firstly, data can originate from the client itself and written inside the Random ByteIO resource by invoking the addInputData() method of the client-side activity and passing an InputStream. Secondly, the output of another activity can be connected to the data input of the discussed activity by invoking the connectDataInput() method and passing a SingleActivityOutput object to it. This will allow resulting binary data of other activities to be stored inside the ByteIO resources.

---

2BLOB is a binary large object that may be stored inside a row of a database. Typically a BLOB will represent an image, video, sound, or other kind of large binary data and its size can reach several GigaBytes, depending on the particular database it is stored inside.
The code snippet of figure 4.9 shows how to create a workflow that copies data from a file into a Random ByteIO resource.

(1) Data can be read from a file using OGSA-DAI’s built-in ReadFromFile activity.
(2) Use the AppendToRandomByteIOResource client-side activity to set the ID of the resource the append operation concerns and the origin of the input data by connecting it to the output of the ReadFromFile activity.
(3) Construct the activity workflow from the activities.

4.3.3 The WriteToRandomByteIOResource activity

The activity can be used for inserting data into a Random ByteIO resource by using the latter’s implementation of the Write operation that was described in the previous chapter. As such, it has a lot in common with the previously described AppendToRandomByteIOResource activity, which serves a similar purpose but only allows to append data to the resource. The activity’s input and output specification is described in table 4.3.

Figure 4.10 illustrates how the activity works.

The activity expects as inputs the arguments of the Write operation and the data that will be recorded into the resource. These can be provided either by the client itself or by connecting the outputs of a preceding activity in the pipeline to the corresponding inputs of the current one. Streaming data into the resource using the Write operation of the Random ByteIO interface is not an easy thing to achieve, since the operation stores...
Figure 4.9: Example of a workflow construction for copying data from a file into a Random ByteIO resource

<table>
<thead>
<tr>
<th>Inputs &amp; Outputs</th>
<th>Name</th>
<th>Type</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>INPUT_DATA</td>
<td>byte[]</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_STARTING_OFFSET</td>
<td>long</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_BYTES_PER_BLOCK</td>
<td>integer</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_STRIDE</td>
<td>long</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.3: Input and Output specification of the WriteToRandomByteIOResource activity

data at random offsets in the resource. It was finally decided to gather all binary data from the activity’s input stream inside one binary array and write this into the resource in one go.

There are two reasons that justify this decision. Firstly, computing the starting offset into the resource for each block of data the activity receives would have added a lot of complexity to the activity code and secondly, one would have expected the Write operation of the Random ByteIO interface to be used for writing a chunk of bytes inside the resource and not large amounts of bulk data. For the latter reason, streaming data into the resource is not expected to increase the performance of the implementation a lot.

The size of the final binary array that would be recorded into the resource was not possible to calculate a priori, since the activity cannot know how much data the input

---

1It may have been beneficial here if the Random ByteIO interface provided an operation for getting the offset of the last block that was recorded into the resource.
stream will contain. Therefore, the array had to be resized every time a block of data was received by the activity. However, this was not straightforward to accomplish, since the Java programming language deals only with static arrays. The solution was to create a new larger array each time a block of data was read from the input stream and copy the previous and new data into this. The pointer of the old array would then be set to the new array, allowing Java to garbage-collect the unreferenced old data. The code snippet of figure 4.11 illustrates this process.

Finally, since the activity is targeted at a particular resource (i.e. the one that the data will be recorded into), this needs to implement the ResourceActivity interface for setting the resource accessor. Furthermore, since data will be read from a stream in every activity-iteration, it also needs to extend the RegularIterativeActivity base class.

**Client-side activity**

The corresponding client side activity implements the ResourceActivity interface and extends the BaseResourceActivity base class for getting and setting the id of the resource the operation concerns. The activity allows assembling a client request that will contain the arguments of the Write operation and the resource id that the server-side activity expects. Furthermore, since the activity is expected to be in workflows for writing resulting data from other activities into a Random ByteIO resource, a method has also been implemented for connecting the output of a preceding activity in the pipeline to the data input of the current activity.

An example of how someone may utilize this client-side activity is given in a later
Adding blocks of data into an array by resizing this

```java
// gather data of all blocks inside "bytes" array.
while ((bytesRead = data.read(blockbytes, 0, BLOCK_SIZE)) != -1) {
    int newSize = bytes.length + bytesRead;
    byte[] temp = new byte[newSize]; // The new array

    // copy old data to new array (bytes -> temp)
    System.arraycopy(bytes, 0, temp, 0, bytes.length);
    // copy new data to new array (tempbytes -> temp)
    System.arraycopy(blockbytes, 0, temp, bytes.length, bytesRead);
    bytes = temp; // Set bytes to refer to new array
}
```

Figure 4.11: Reading blocks of data from a stream and copying these into an array by resizing it.

4.3.4 The ReadFromRemoteRandomByteIOResource activity

The Random ByteIO interface is expected to become a standard that every data provider should implement for exposing its data to the Grid. As such, it was decided to implement another activity that would allow reading data from remote Random ByteIO resources into OGSA-DAI. The input and output specification of the activity is presented in table 4.4.

<table>
<thead>
<tr>
<th>Inputs &amp; Outputs</th>
<th>Name</th>
<th>Type</th>
<th>Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>INPUT_STARTING_OFFSET</td>
<td>long</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_BYTES_PER_BLOCK</td>
<td>integer</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_BLOCKS</td>
<td>integer</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_STRIDE</td>
<td>long</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_TRANSFER_MECHANISM</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_SERVICE_URL</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>Input</td>
<td>INPUT_RESOURCE_ID</td>
<td>String</td>
<td>No</td>
</tr>
<tr>
<td>Output</td>
<td>OUTPUT</td>
<td>byte[]</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.4: Input and Output specification of the ReadFromRemoteRandomByteIOResource activity.

For contacting the remote resource, the activity will need as inputs the id of the remote resource and the location (i.e. URL) of the Random ByteIO service that manages this.
Then by using the stub classes of the remote service the activity can target a particular remote Random ByteIO resource and retrieve the requested data. For the latter purpose, the arguments of the remote Read operation will also need to be given to the activity, via additional inputs. More details on how the activity manages to contact the remote service can be found in chapter 5.1 where a client for the Random ByteIO service is being described.

It should be noted that for contacting the remote resource, the activity constructs its endpoint reference by using the resource id and URL of the service that the user has specified. This restricts the implementation, since it only allows to contact remote resources that are managed by Random ByteIO services that are deployed in OGSA-DAI. Normally, one would have expected the activity to use discovery mechanisms to retrieve the endpoint reference of a remote resource and not construct it by itself. However, this has been out of scope by this project.

Following the connection to the remote resource, the OGSA-DAI server acts as a client, fetching blocks of bytes from the remote Random ByteIO resource and storing these inside a binary array. Upon the successful completion of the Read operation, the retrieved data are added inside a list and this is forwarded to the output of the activity. A list is being used because many OGSA-DAI activities that expect byte arrays as input require that the data are grouped inside lists. This originates from the requirement that data need to be streamed through the activity pipeline and this can be achieved by making activities processing lots of small byte arrays inside a list, rather than one large array itself [4].

Figure 4.12 illustrates how the activity works in practice.

Figure 4.12: The ReadFromRemoteRandomByteIOResource activity

The returned data are forwarded to the output of the activity in one go, and these are
not streamed. This is because of two main reasons. Firstly, the remote Read operation returns the data to the activity in one block and as such there is no real benefit from streaming these in the output, since they will already be stored inside the activity anyway. Secondly, partitioning the data after these have been retrieved and forwarding blocks of them to the output at every iteration is not expected to increase the performance significantly, since the Read operation of the Random ByteIO interface will normally be used for accessing chunks of bytes and not large amounts of bulk data.

**Client-side activity**

The corresponding client-side activity provides methods for assembling a request that will contain the arguments that the activity expects. Moreover, since the activity is expected to be used inside workflows for forwarding the returned data from a remote Random ByteIO resource to another activity for processing these, a getDataOutput() method has been implemented for connecting the output of the discussed activity to the input of another OGSA-DAI activity that expects a list of byte arrays.

The code snippet of figure 5.8 in later chapter shows how the client-side activity can be used for building a data workflow.

### 4.4 Deploying the Random ByteIO service

The service can be deployed into the Tomcat web services container following the normal OGSA-DAI setup procedure. This is accomplished by placing the web service’s schema inside the OGSA-DAI source distribution and editing the build files to include the details of the new web service. In more detail, the following files were edited:

**build-stubs.xml** A new XML element was added inside the target "createOGSADAIStubs" for creating the Random ByteIO stubs using the WSDL2Java tool of Axis 1.4. The service’s schema was placed under the folder schema/ogsadai/axis-specific-wsdl/rbyteio.

**build-binary.xml** A new XML element was added inside the target "buildWAR" to include all Random ByteIO specific classes inside the OGSA-DAI web services archive. New XML elements were added inside the target "deployServices" to deploy the Random ByteIO service into the Tomcat web services container.

**NStoPkg.properties** The file was edited to instruct the WSDL2Java tool of Axis where to place the Random ByteIO stub classes (i.e. what Java package). The package was decided to be uk/org/ogsadai/service/axis/rbyteio, following the OGSA-DAI namespace conventions.

Following the above edit, the service can be deployed by executing ant twice. Once for building the OGSA-DAI binary release and another for deploying this into Tomcat. If
everything goes well, the new service will appear under the *dai* project in the Tomcat Web Services container, along with all other OGSA-DAI core services. The service can now be invoked directly by any client, since TOMCAT will automatically forward any request that contains the service’s URL to it.
Chapter 5

Demonstrating the new functionality

Following the implementation of the service and its connection to OGSA-DAI, a client application needed to be developed that would demonstrate the potentials of the new functionality. This chapter describes the client application and the implementations of a number of basic use case scenarios for the Random ByteIO interface that have been proposed by the ByteIO Working Group and specified here [18].

5.1 Client Software Design

So far it should have been made clear that there are two ways of accessing data from a Random ByteIO resource. This can be accomplished either by contacting the Random ByteIO service directly and invoking one of its operations, or by submitting OGSA-DAI data workflows that would manipulate the data of the resources via activities. The client that had been developed allows for both ways to be utilized.

The client provides an interactive command line interface for acquiring the user’s requests and triggering the appropriate operations that are to be executed on the server-side. The client supports a large number of commands. These are listed in section C.1 of the Appendix.

Figure 5.1 illustrates the class diagram of the client application. The interactive interface is contained inside the Client class and this is the first point of contact to the user. Requests are forwarded to the appropriate client "plug-ins" for executing these on the server-side. A plug-in is defined in its own class and encapsulates the required functionality for building a particular user request and submitting this to the server. This design allows to easily extend the client by adding more functionality to it via additional plug-in implementations. What is more, the new plug-ins can benefit from the existing functionality of the client (i.e. mechanisms to create new resources, or to construct an endpoint reference), resulting in reduce of code reuse and no changes to the core source. A new plugin can be installed following the steps that are described in section C.2 of the Appendix.
Currently the client implementation provides four plug-ins for submitting OGSA-DAI data workflows that are constructed from the activities that have been discussed in section 4.3 and one plug-in for invoking operations of the Random ByteIO service directly. The way these work is described in more detail in section 5.2, where implementations of use case scenarios for the Random ByteIO interface are presented.

### 5.1.1 Managing endpoint references in the client-side

Care has been taken so that the client implementation would be interoperable with any Random ByteIO implementation and not only with the one that has been connected to OGSA-DAI. There was an issue here with the mechanism that created new Random ByteIO resources. This was not part of the standard ByteIO, but rather OGSA-DAI specific since the resources were being created using OGSA-DAI data workflows.

In more detail, upon the creation of a new resource, OGSA-DAI returns back to the client the resource id (output of the corresponding activity). Although this could be

---

1 The benefits of interoperability are discussed in more detail in chapter 8.
passed as argument to other activities for constructing OGSA-DAI data workflows that would access the data of the particular resource, it was not making it possible to invoke any operation of the Random ByteIO service directly. The reason was that more information was needed to accomplish this, such as the URL of the Random ByteIO service that manages the resource and the way the id would be wrapped inside any request message. This information is normally stored inside an endpoint reference (EPR) object, which is an XML document of particular structure (i.e. XML Schema) that is used for identifying a particular resource \[29\]. Upon invoking any remote operation that targets the resource, the information that is contained within its EPR is copied into the Header of the SOAP message and the receiver extracts it from there.

The client needed a mechanism for retrieving the endpoint references of the resources that were created by OGSA-DAI. Section[8.1.1] describes such a mechanism in more detail. Here it will suffice to say that the current client implementation creates an endpoint reference from scratch and initializes it with all required information for identifying a particular resource. This is achieved by invoking the \textit{createEPR()} method that is contained inside the \textit{CreateRandomByteIO} class and passing the resource id as argument to this.

Finally, for making use of the information that is stored inside an endpoint reference, the client should be able to copy this into the Headers of the SOAP messages that are sent to the server using the stub classes of the Random ByteIO interface. This can be accomplished by invoking the \textit{setSOAPHeader()} method of the \textit{RByteIO} class.

Knowing how to process an endpoint reference correctly, the client can access the data of any Random ByteIO resource for which it has the associated endpoint reference and therefore it is considered interoperable with any service implementation of the corresponding interface.

5.2 **Implementations of basic Random ByteIO use case scenarios**

The use cases that are discussed here concern basic ways of utilizing Random ByteIO resources and do not refer to any particular real-case access scenario. The intention is to demonstrate that the Random ByteIO service, connected to other existing technology (e.g. OGSA-DAI) that hides the underlying complexity of the Grid, allows for accessing distributed data in flexible and efficient ways, as if they were local to the client application. Real applications are expected to incorporate the implementations presented here in their codes for accessing Random ByteIO resources in various application-specific ways.

Since the aim of the project was to connect the Random ByteIO service to OGSA-DAI and the latter provides a Web Service interface to structured collections of data in the Grid, the use cases focus on database access and integration scenarios. In particular, the following use cases are investigated:
1. Asynchronous database query
2. Inserting data into a database
3. Copying data between databases

5.2.1 Asynchronous database query

Applications often need to access data in an asynchronous manner. Data need not be delivered in one step but in multiples instead, allowing the client application to process chunks while others are being retrieved, possibly in different threads. This way the client need not be kept busy-waiting until the whole transfer is completed, and by overlapping communication with computation the performance can be increased. Furthermore, in case the data are too large to store inside the client’s memory as a whole, blocks of these can be fetched from the resource, processed and transferred back to the latter in multiple iterations. That way the client will never need to hold the full amount of data inside its memory at the same time.

Implementing an asynchronous query using the Random ByteIO service and the OGSA-DAI middleware was straightforward to accomplish, since the service provides operations for accessing chunks of bytes from a Random ByteIO resource and a client that holds an endpoint reference to the latter can access its data or resource properties at any time in the future. The operation can be accomplished following the steps below:

1. The client requests that OGSA-DAI creates a new Random ByteIO resource.
2. The client requests that the data the user wishes to read is copied into the newly created Random ByteIO resource.
3. The client retrieves chunks of data from the resource by issuing a single or multiple Read requests directly to the Random ByteIO service.

A new Random ByteIO resource will first need to be created so that data can be accessed from it via the operations that the interface provides. This is accomplished by submitting a data workflow to the Data Request Execution Resource that is built from the CreateRandomByteIOResource activity. The newly created resource will be empty at first and need to be loaded with the data the user is interested in. This can be achieved by submitting another workflow that would fetch the required data from some data source or sink in the Grid (i.e. database) and store these inside the Random ByteIO resource. OGSA-DAI already provides a large number of activities for accessing data resources in the Grid. By connecting the output of any of these activities to the input of the AppendToRandomByteIOResource activity, one can create a stream of data for copying these into the Random ByteIO resource. Finally, having the id of the newly created resource and knowing the URL of the Random ByteIO service that manages this, the client can invoke the Read operation of the service to retrieve chunks of bytes from the resource at any time in the future.
Figure 5.2 illustrates the steps that were described here for accessing a database BLOB in parts.

To demonstrate the process of querying a database asynchronously, client code has been written for receiving an image stored inside a database in chunks of bytes. The image is reconstructed in the client-side by combining all chunks together, so that it can be compared with the original image and check that this is identical. This way, the application can also serve as a test case for proving the correctness of the Random ByteIO interface implementation.

Accessing a database BLOB in parts

A PostgreSQL database has been set up in the same machine the OGSA-DAI server runs on, and a table has been constructed for storing the BLOBs inside. The table’s schema is given below:

---

2 The location of the database should not matter however, since OGSA-DAI can target any remote database via specific resources.
Images are stored using the \textit{bytea} datatype of PostgreSQL. This represents large binary data (up to 1 GByte in size) that are contained within the same column of the table they are defined.

A client can request that a BLOB is extracted from a database and stored inside a Random ByteIO resource by submitting a pipeline of activities to the Data Request Execution Resource. Figure \ref{fig:blob-extraction} illustrates a client code snippet that creates an OGSA-DAI data workflow for performing the discussed operation. Notice that this is built using four client-side activities connected together.

\begin{center}
\begin{Verbatim}
//Construct a new SQLQuery object from your SQL query string. \hfill (1)
SQLQuery query = new SQLQuery();
query.addExpression("select blob from myblobs where name = 'stairs.jpg';");
query.setResourceId(new SimpleResourceID(id2));

//Retrieve list of blobs from tuple with TupleSplit \hfill (2)
TupleSplit split = new TupleSplit();
split.connectDataInput(query.getDataOutput());
split.setNumberOfResultOutputs(1);

//Need to remove the list begin - list end markers from the tupleSplit output \hfill (3)
ListRemover remover = new ListRemover();
remover.connectInput(split.getResultOutput(0));

//Append BLOB to ByteIO Resource \hfill (4)
AppendToRandomByteIOResource appendToResource = new AppendToRandomByteIOResource();
appendToResource.setResourceId(rbyteIO_ID);
appendToResource.connectDataInput(remover.getOutput());

//Create a request which holds all the activities defined. \hfill ...
PipelineWorkflow workflow = new PipelineWorkflow();
workflow.add(query);
workflow.add(split);
workflow.add(remover);
workflow.add(appendToResource);
\end{Verbatim}
\end{center}

Figure 5.3: Writing a BLOB inside a Random ByteIO resource
1. Data can be queried from the database using OGSA-DAI’s built-in `SQLQuery` activity. This takes as arguments the id of the resource that targets the database that contains the data and the query that the user wishes to be performed.

2. Upon successful completion of the "SQLQuery" activity, a tuple object will be constructed by the activity, representing the returned database tuple. The activity `TupleSplit` can be used for retrieving a list of BLOBs from the tuple. The reason that a list of BLOBs is returned and not the BLOB itself, is that the table may contain multiple rows with BLOBs. In case the results contained two BLOBs, the `TupleSplit` activity would perform the following:

   \[
   \text{tuple}((\text{BLOB1}, \text{BLOB2})) \rightarrow \{\text{BLOB1}, \text{BLOB2}\}.
   \]

3. The list of BLOBs will need to be further processed in order to extract the BLOBs from the list and pass these to the activity that writes data into the Random ByteIO resource. This is accomplished by the `ListRemover` activity, which removes the list-begin and list-end markers from the results and outputs the BLOB(s) only. Again, in case the results contained two BLOBs, the activity would perform:

   \[
   \{\text{BLOB1}, \text{BLOB2}\} \rightarrow \text{BLOB1,BLOB2}
   \]

4. Finally, the activity `AppendToRandomByteIOResource` can be used for appending the binary data (i.e. the BLOBs) to a particular Random ByteIO resource.

Following the recording of the image inside the resource, the data can now be accessed in chunks of bytes by invoking operations of the Random ByteIO service directly. The client retrieves the stub classes of the Random ByteIO service and adds to the SOAP header the id of the resource the operations will concern. The data of the resource are divided into chunks of equal size and these are fetched and appended to a local file one after the other inside a for-loop. Each chunk will consist of one contiguous block of bytes, and hence the number of blocks for the "Read" operation should be one and the value of stride does not matter. The code snippet of figure 5.4 illustrates the process.

```
for (int i=0 ; i < chunks-1 ; i++) {
    //Reads ((bytesOfFile+chunk-1)/chunks) bytes of data
    data = byteio.read(i*((bytesOfFile+chunks-1)/chunks), (bytesOfFile+chunks-1)/chunks, 1 , 0);
    //append chunk of bytes to file
}
//Last chunk may contain less data than((bytesOfFile+chunks-1)/chunks) bytes.
data =byteio.read((chunks-1)*((bytesOfFile+chunks-1)/chunks), bytesOfFile-(chunks-1)*((bytesOfFile+chunks-1)/chunks), 1 , 0);
//append chunk of bytes to file
```

Figure 5.4: Reading data from a resource in chunks
For specifying the size of each chunk, the client will need to know the total amount of data the resource contains. This can be accomplished by invoking the `getResourceProperty()` operation of the Random ByteIO service to read the value of the `Size` resource property. The client can then specify the exact size of each chunk of bytes along with its starting offset in the resource and request that this is retrieved in every iteration of the for-loop. Since the number of bytes in the resource may not be divided equally by the number of chunks, the size of the last chunk is calculated by subtracting the number of bytes that have already been fetched from the total number of bytes the resource contains.

**Performance Investigations**

The execution time of the operation described in the previous subsection was measured and compared to that of retrieving the same image as a whole, using existing OGSA-DAI mechanisms. The results have been summarized in table 5.1.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>1 Chunk</th>
<th>50 Chunks</th>
<th>100 Chunks</th>
<th>OGSA-DAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>≃ 1.8MBytes</td>
<td>3,450 msec</td>
<td>3,680 msec</td>
<td>4,650 msec</td>
<td>2,320 msec</td>
</tr>
<tr>
<td>≃ 1.3MBytes</td>
<td>2,800 msec</td>
<td>3,150 msec</td>
<td>4,160 msec</td>
<td>2,110 msec</td>
</tr>
<tr>
<td>≃ 0.9MBytes</td>
<td>2,078 msec</td>
<td>2,500 msec</td>
<td>3,410 msec</td>
<td>1.690 msec</td>
</tr>
</tbody>
</table>

Table 5.1: Timings

The existing functionality of OGSA-DAI appears to perform better than the Random ByteIO implementation for retrieving images in one chunk. The main reason is that OGSA-DAI uses buffered streams for accessing data from resources and this can result in enhanced performance, especially when the size of the requested data is large (i.e. for the larger size of images shown in table 5.1). On the other hand, the implementation of the Random ByteIO interface uses the Random Access File I/O mechanism of Java for accessing data in a random manner and this is unbuffered by default. Accessing one byte at a time and moving the file pointer to the location of the next byte in the file consumes a lot of system resources and tends to be expensive. However, it can be argued here that the Random ByteIO interface was not designed for accessing contiguous parts of data in first place. This would be a typical use case for the Streamable ByteIO standard interface that comes with the ByteIO specification but was not implemented by this project. The Streamable ByteIO provides operations for accessing data from streams, and thus it can be built on top of Java I/O mechanisms that use buffered streams for accessing data and not Random Access Files (see section B of the Appendix for more).

Still, the performance of the Random ByteIO implementation can be further increased if a buffered wrapper is implemented on top of the `RandomAccessFile` mechanism. This can be accomplished by implementing read and write operations that use the `RandomAccessFile` ones but allow accessing a sequence of bytes directly. An example of how this might work is given below for reading a block of bytes from a file:
raf = new RandomAccessFile(file, "r");
raf.seek(offset);
raf.read(block_buffer);

The user in this case only needs to specify the starting offset of the block that will be read from the file and this is read directly and stored inside a buffer. The file pointer "raf" is moved only once per block access, resulting in fewer system resources being used and better performance. Obviously, the applications that would benefit most from the existence of a buffered wrapper are the ones that access data that are not very random, but stored in nearby locations in memory, like the one presented here for retrieving an image in contiguous chunks of bytes.

Another advantage of the Random ByteIO implementation compare to what OGSA-DAI provides at the moment, is that data need only be read once from the database. As soon as these are stored inside a Random ByteIO resource (i.e. a file in the server), one or more clients may request accessing them multiple times in the future without needing to contact the database at all. This has the potential of increasing the performance of many applications, since accessing data from a database can be much slower than accessing data from files.

Regarding the number of chunks an image is being retrieved in when using the Random ByteIO, table [5.1] shows that the performance downgrades while this is increased. This is expected, since messages are subject to the latency of the network and a remote operation adds overheads for processing the request and response messages as well.

In the current Random ByteIO implementation data are being transferred from the server to the client inside the body of the SOAP message by encoding these into "Base64" text format. This is not regarded as an efficient means of transferring large binary data as it was described in subsection [2.2.1]. However, it should be noted here that one can easily plug into the implementation more efficient transfer mechanisms, like MTOM[20] or DIME[21] that transfer data as MIME attachments and avoid the "Base64" text encoding, resulting in reduced size of data and increased processing speed. This is a major advantage of the Random ByteIO implementation compare to the existing access mechanisms of OGSA-DAI. The reasons have also been discussed in subsection [2.2.1] of the background.

The discussion of the results proves that the Random ByteIO implementation can easily be extended to achieve better performance by plugging in more efficient transfer mechanisms and optionally implementing a buffered wrapper on top of the Random Access Files mechanism. This has been left as future work, since it was not a primary aim of the project and time was not enough. The intention here was to show where the Random ByteIO implementation currently stands in respect to performance. In no way however was expected that the implementation would outperform OGSA-DAI in the simple case of reading contiguous blocks of data and transferring these to the client inside SOAP messages that was investigated. The real benefit of using the Random ByteIO implementation comes from the ability to access a data resource in parts and in random ways,
a functionality which was not supported by OGSA-DAI.

5.2.2 Insert into Database

Inserting data into a database using the Random ByteIO interface can be seen as the reverse operation of querying these. In a simple case scenario, a client is expected to upload data to a Random ByteIO data resource in chunks of bytes using the operations of the interface and request that the whole data is transferred to the database in the end. The implementation of the Random ByteIO interface allows appending or replacing existing blocks of data in a Random ByteIO resource while leaving the rest of the data unaffected. This can be very useful in cases when the data of a resource are very large in size and the user may only need to process a small chunk of bytes. Combining this with the asynchronous query scenario that was discussed in the previous section, the Random ByteIO implementation allows for processing blocks of bytes “in place” inside the Random ByteIO resource, while avoiding unnecessary data transfers and hence maximizing the overall performance.

Two ways of inserting data into a Random ByteIO resource have been investigated and implemented by this project. In the first case the user requests that data are transferred from its local memory into the database in chunks of bytes, whereas in the second the data are supplied from a third party provider.

Simple Insert

Similarly to the asynchronous query, the process of inserting data into a database using the Random ByteIO service and the OGSA-DAI middleware consists of multiple steps. These are described below:

1. The client requests that a new Random ByteIO resource is created by the OGSA-DAI server.

2. The client copies data into the resource by invoking operations of the Random ByteIO service directly.

3. The client requests that the data of the Random ByteIO resource are copied into a database.

The client will first request that a new Random ByteIO resource is constructed in the server-side for storing the data that are to be inserted into the database inside. It does that by submitting a data workflow constructed from the CreateRandomByteI0DataResource activity, which in turn sends back the id of the newly created resource. Alternatively, the client may use an existing resource for the same purpose and avoid the creation of a new one. Following this action, the client can retrieve the stubs of the Random ByteIO service dynamically and set the resource id inside the SOAP message header so
that it can invoke operations on the service that will target the specific resource. The interface provides two operations for writing data into the resource, namely the *write* and *append* operations, described in chapter 3. These can be invoked directly for uploading chunks of bytes to the resource in an asynchronous manner. Upon the completion of any data transfer to the Random ByteIO resource, the client can request that the whole data of the resource is transferred inside a database by submitting another data workflow to the OGSA-DAI server that is described in more detail later in the section.

Figure 5.5 illustrates how a client might use the Random ByteIO implementation and OGSA-DAI to insert data into a database.

![Diagram of inserting data into a database using the Random ByteIO](image)

Figure 5.5: Inserting data into a database using the Random ByteIO.

To illustrate the process of writing chunks of bytes into a Random ByteIO resource, a use case has been implemented, where data are read from a local file in the client and appended to a Random ByteIO resource in chunks.

The client can insert data in chunks by executing a for-loop that reads a block of bytes from a file and appends this to the resource by invoking the *append* operation of the Random ByteIO service at every iteration. This is illustrated in figure 5.6.
Figure 5.6: Writing data into a Random ByteIO resource in blocks of bytes.

Following the recording of the data into the Random ByteIO resource, the client can request that these are copied to a database table by executing an OGSA-DAI data workflow that will stream the data inside the database. The data workflow will need to be constructed from the following activities:

**ReadFromByteIO** The activity allows to read data from the resource-specific file and forward these to another activity. The client only needs to specify the id of the resource that is to be read.

**TupleForm** This activity will be used for creating a tuple object that will contain the binary data returned by the ReadFromFile activity. The reason is that OGSA-DAI needs data to be wrapped inside a tuple object for storing these into a database table.

**SQLParameterizedUpdate** The output of the TupleForm activity will need to be connected to the input of this activity so that the tuple that contains the data can be stored inside the database. Furthermore, since data will be streamed inside the database, the client will need to request that the first block replaces any previous data inside the table (e.g. by executing an SQL update), while the next blocks

---

3This activity has not been implemented, but the existing ReadFromFile activity of OGSA-DAI was used instead, since the data of the resource were stored inside files. It would be beneficial however to support this in a future release of the implementation, since it would allow hiding some of the complexity of the implementation from the user.

4Unfortunately the activity TupleForm does not exist in the current version of OGSA-DAI, but this is planned to be implemented in the near future.
will need to be inserted into the table without replacing the previously inserted ones (e.g. by executing an SQL insert).

The above discussion shows that by using core activities of the OGSA-DAI middleware it is possible to copy the data of a Random ByteIO resource into a database table. However, this has not been implemented in practice, since the activity TupleForm was missing from the new version of OGSA-DAI 3.0. Implementing a core activity of OGSA-DAI for transforming data into tuples has been out of project’s scope. What is more, an implementation of such an activity will need to consider every possible type of input data and not only binary data, as it is the case here.

**Complex insert**

Data that are very large in size are often stored in a different machine than that the client runs on. Accessing the data and requesting that blocks of these are inserted into a database via OGSA-DAI would have resulted in low performance if these had to go through the client. What is more, it is possible that the machine the client will be running on will not have adequate memory to temporarily store the data before these are copied to the OGSA-DAI server.

These issues can be overcome if the data were transferred directly from the provider to the OGSA-DAI server. This can be achieved following the steps below:

1. Client submits a "complex-insert" request to the OGSA-DAI server.
2. OGSA-DAI server becomes a client, reading data from a Random ByteIO resource that is located in the provider’s machine.
3. OGSA-DAI server writes the returned data inside its local Random ByteIO resource.
4. OGSA-DAI transfers data from the Random ByteIO resource to the database.

For implementing this use case, a second Random ByteIO server was required to act as the data provider. This was set up on a second machine that the OGSA-DAI team uses for running tests. The machine had a static IP and this made it possible to access the service via the OGSA-DAI server that was running on the laptop. The URL of the second Random ByteIO service was:


Figure[5.7] illustrates how a complex-insert can be accomplished with the new functionality that has been added to OGSA-DAI.
1. Request (Pipeline of activities)

2. read(resourceEPR):byte[]

3. read():byte[]

4. write():void

Figure 5.7: Inserting data into a Random ByteIO resource that originate from a third party data provider that supports the Random ByteIO interface.

Figure 5.8 in the next page illustrates a snippet of a client-side code that can be used for building the request for the complex-insert scenario. The data workflow that will be submitted to OGSA-DAI to perform the complex-insert scenario is constructed from the activities ReadFromRemoteRandomByteIOResource and WriteToRandomByteIOResource, described in more detail in section 4.3. The first allows reading data from a remote Random ByteIO resource by accessing the service that manages this and is located in the data provider side. The output of the first activity will need to be connected to the input of the second so that the retrieved data can be recorded into the Random ByteIO resource that is located in the OGSA-DAI server. Since the data are being stored inside the resource using the Write operation of the Random ByteIO implementation, the client can supply additional information about how the data will be stored inside the resource (i.e. starting offset, size of blocks and stride).

The scenario has been implemented up until the point where OGSA-DAI writes the data inside its local Random ByteIO resource. The data have not been recorded inside the database, since the activity TupleForm that was described in the previous section was missing.

In conclusion, the "complex-insert" scenario that was presented here will allow for more flexibility and enhanced performance, since data will be moved and processed without the client’s interference. Furthermore, the mechanism will allow OGSA-DAI to fetch data from remote Random ByteIO resources, which is important since the Random ByteIO is expected to become a standard interface for exposing data in the Grid and most data providers are going to support it.
5.2.3 Copying data between databases

The Grid is a hardware and software infrastructure that spans thousands of computing machines and storage resources. In this context, data resources can become unavailable at any time, due to a network, server, or any other kind of failure (e.g. resource destruction, corruption, and so forth). Maintaining multiple copies of data resources is of major importance for the success of the Grid, since this will allow for high availability, enhanced performance and fault tolerance. In case a data resource fails, clients can target an available copy of it for accomplishing their tasks. Data can be moved near to the users that needs these, thus reducing the latency of fetching them\(^5\). Finally, backups of data collections can be made, so that these can be secured and tested that they always contain correct data.

Replicating data in the Grid using the Random ByteIO interface allows for more flexibility and enhanced performance. Users will be able to select parts of data to replicate.

\(^5\)This is termed as data caching.
from a data resource, thus avoiding unnecessary transfers of unwanted data. This can be especially useful when the data are very large in size and storing these in intermediate nodes (i.e. servers) may be difficult or even impossible due to memory constraints. What is more, it will be made possible via the Random ByteIO interface operations to incorporate any parts of data in existing different data resources.

A use case scenario has been implemented for copying data from one database to another using the OGSA-DAI middleware and the Random ByteIO implementation. Each database was interfaced with a different OGSA-DAI server that contained a resource for accessing this. Figure 5.9 illustrates the steps that needed to be performed for transferring a chunk of data from one database to another.

![Diagram of copying data between databases](image)

**Figure 5.9: Copying data between databases.**

The copy can be achieved by combining use cases that have been discussed in the previous sections. The client will first need to submit a data workflow to the OGSA-DAI server that provides a connection to the database that contains the original data to request that these are extracted and copied into a new Random ByteIO resource "A". The workflow will need to be constructed from the same activities that were described in section 5.2.1 for retrieving a database binary large object (BLOB). Next, the client will need to instruct the second OGSA-DAI server to read parts of data from the Random ByteIO resource "A" and store these inside a new or existing local Random ByteIO data
resource "B". This is accomplished by submitting a data workflow that is constructed from the activities described in section [5.2.2] for writing data into a Random ByteIO resource that originate from a third party provider. Finally, the second OGSA-DAI server is requested to read the data from its local Random ByteIO resource "B" and store these inside the target database. This is achieved by submitting another data workflow, described in section [5.2.2] for inserting data into a database.
Chapter 6

A provisional real use case scenario implementation

The previously described use cases demonstrated the functionality that has been added to OGSA-DAI by the Random ByteIO interface alone. Numerous benefits can derive from utilizing this in conjunction with OGSA-DAI’s existing mechanisms. This will allow to access and integrate subsets of data from multiple data resources, which is considered a substantial asset for the development of an efficient and flexible Grid infrastructure. A provisional real use case scenario that demonstrates the potentials of the combined functionality is discussed next.

6.1 Medical Processing

Applications that deal with medical research and clinical care are often good candidates for the Grid. The size of data they access is vast and covers lots of diverse file formats, including images, videos, text, volumes and so forth. What is more, data are distributed across hundreds of databases in different geographical locations and these will need to be integrated efficiently and analyzed.

Suppose that a patient has different types of data stored inside distinct databases and a medical research centre wishes to retrieve and process parts of these. This can easily be achieved by submitting a request to an OGSA-DAI server for performing the following operations:

1. Contact a metadata store to find out the locations of the patient’s data.
2. Execute a cross join operation across databases to retrieve the desired data.

A cross join allows to combine each row of the first table with the that of the second table and include these in the result table.
3. Filter the retrieved large binary data (i.e. the X-Ray images) by extracting subsets of these using the Random ByteIO interface.

4. Transfer, analyze or replicate the desired subsets of data.

The first operation is known as data discovery. A user does not need to specify the exact location or name of the data he wishes to access, but these can be discovered using metadata attributes that point to them. Having found the location of the data and the virtual data resources that make these available, OGSA-DAI can execute a cross join built-in activity to collect all data that are associated with the particular patient from the distinct databases. Large binary data (i.e. videos, images) can then be stored inside Random ByteIO resources on the OGSA-DAI server and be made available via endpoint references, while text data, such as the patient’s name, age or genre can be returned inside the response message when the operation completes. This can be achieved by forwarding the output of the "cross join" activity to the TupleSplit activity that allows separating the values of the columns and outputting these to different activities. The values of the columns that contain large binary data can this way be forwarded to the appropriate Random ByteIO resources using the AppendToRandomByteIOResource activity. The activities have been described in subsection 5.2.1.

Following the storing of the data inside the Random ByteIO resources, these can be filtered before they are transferred elsewhere. The ReadFromRandomByteIO activity can be executed (see subsection 5.2.2) for extracting the required subsets of the binary data (i.e. a part of an image) and forwarding these to any other transformation activities. Subsets of data resources can this way stream through a workflow of activities and be manipulated in various ways. For instance, a part of an image can be illuminated and later compressed and stored inside a different Random ByteIO resource. What is more, the transformed subsets of data can easily be replicated as it has been described in subsection 5.2.3 to ensure that these will be available in other sites too.

Given the functionality that has been described here, the medical centre can process subsets of a patient’s distributed data on the server-side and request that the results are retrieved or transferred to other sites. Data need not necessarily be delivered to the client application, but this can access these (e.g. for monitoring purposes) asynchronously at any time in the future via their endpoint references, as it has been described in subsection 5.2.1. Moreover, the Random ByteIO implementation on the server is expected to provide efficient transfer mechanisms for copying the results to the medical centre or any other site, resulting in increased performance.
Chapter 7

Testing

Testing has been crucial for the success of the project, since this allowed finding and controlling defects in the code. Given the amount of software developing that has been carried out, testing made it possible to manage the risks that were associated with it and increase the chances that the project would finally meet its goals. What is more, it facilitated changes to the code, since it constituted a way of assuring that these would not break the existing code.

The chapter describes the unit and system tests that have been developed to assure that the implementation functions correctly. Both tests have been created using the JUnit testing framework of the Java programming language. Interoperability tests have also been carried out, but these are discussed in the next chapter.

7.1 Unit tests

Unit testing serves as an informal way of testing individual parts of source code and assuring that these function correctly \[12\]. The test cases that have been created cover most paths of accessing the parts of code they correspond to. Each operation of the Random ByteIO interface has been unit tested to make sure that this works properly. The tests check for correctness, boundary conditions (e.g. negative arguments) and throws of the appropriate exceptions in every case. Furthermore, in order to test the operations in isolation from the OGSA-DAI framework, mock Random ByteIO resources are constructed manually and these are tested instead.

The new activities have also been unit tested by checking that these function as they are expected to. Again, in order to isolate the activity code from the rest of OGSA-DAI and test this individually, mock objects have been utilized that would fake the input and output pipes of the activities. These were provided by the OGSA-DAI project team, which also uses them for its own unit tests.

It should be noted that although a substantial effort was made to include as many test
cases as possible, it is possible that not every possible input to the program has been identified and tested. Unit tests may reveal errors in a code, but they cannot prove that there are no errors at all. Tests evolve while code progresses and new issues come to light. What is more, unit tests are applied to small independent parts of a program and therefore may not catch integration or other system wide errors. For the latter purpose, system tests have also been developed. These are described next.

7.2 System tests

System tests have been developed for assuring that the implementation of the Random ByteIO interface meets its specified requirements. These test the complete system from the user’s perspective and they have no knowledge of the inner parts (i.e. details) of the implementation.

Each operation that the Random ByteIO service exposes to its clients is tested by invoking it and checking that this functions as expected. Abnormal use has also been considered. This is called destructive testing and the system tests are ideal for this purpose. For instance, operations on missing resources are tested, or for unsupported transfer mechanisms. The new activities are also system tested by submitting the appropriate data workflows to OGSA-DAI and checking the results. The values of resource properties are queried and tested too.

Finally, it should be noted that the client application that has been developed also serves as a tool for running system tests.
Chapter 8

Additional work

A primary goal of web services is the ability to talk to each other across different platforms, operating systems and programming languages. A service can have many potential clients, based on different implementations of the standards. Moreover, a service may even become a client itself connecting to another service. Ensuring that all clients can connect to a web service and understand any messages exchanged is critical for the service’s success [10].

In the case of the Random ByteIO, the need for interoperability is even more imperative, since this is expected to become a standard interface for accessing parts of data from data sources in the Grid, where the heterogeneity in both hardware and software is great. Moreover, OGSA-DAI is an application that provides access to data sources in the Grid and as such, it is important that it can interoperate with arbitrary client implementations of the interface.

For ensuring the interoperability of all implementations of the ByteIO interface, the ByteIO Working Group has set up a WIKI page where developers can post the endpoints (i.e. URLs) to their services. Developers can then use their clients to access the services of other developers and test whether these can interoperate. The results are listed in the same page and in case of failures, these can be further investigated. The aim of the interoperability fiesta is that all implementations will finally reach consensus and any issues will be identified and dealt with.

The Random ByteIO implementation that has been discussed in this document took part in the interoperability fiesta that had been organized by the ByteIO Working Group. This added more testing to the implementation for assuring its fine quality. Next, I describe the work that has been carried out for running the interoperability tests and finally discuss the results.

1The WIKI page can be found at https://forge.gridforum.org/sf/wiki/do/viewPage/projects.byteio-wg/wiki/HomePage
8.1 Interoperability testing

The ByteIO Working Group defined a set of tests\(^2\) to ensure that the messages that are exchanged by client and server implementations can be processed correctly. According to the specification, each service that wishes to participate in the tests would have to satisfy a number of criteria. These are summarized below and some notes about these are given in chapter D of the Appendix:

1. The Port Type must be bound to the SOAP 1.1 communication protocol.
2. The binding must use the "document/literal" style.
3. The implementation must support a standard factory service that is defined inside the specification.
4. No security mechanisms should be used (i.e. no encoded messages).
5. Each resource needs to be initialized with a specific string of bytes, described in the specification.
6. The simple transfer mechanism will only be tested. That is, data will be transferred inside the body of SOAP messages.

Next follows the description and implementation of the standard factory service\(^3\).

8.1.1 The factory service

ByteIO resources can be constructed directly by contacting the ByteIO standard factory service. This implements the following interface:

**CreateResource()** Allows creating and initializing new ByteIO resources

**DeleteResource(EndpointReferenceType epr)** Deletes an existing ByteIO resource

The implementation of the factory service uses mechanisms of the OGSA-DAI middleware to create, initialize and destroy Random ByteIO resources. That is, the factory service that was delivered acts only as a frontend for creating new ByteIO resources and hides the OGSA-DAI specific factory mechanisms (i.e. the activity framework). Upon receiving a request for creating a new Random ByteIO resource, the service will act as a client and use the CreateRandomByteIOResource activity that was described in section 4.3 to construct an OGSA-DAI data workflow that will be submitted to the Data Request Execution Resource. A unique resource id will be returned to the factory service and this will be used next together with the AppendToRandomByteIOResource activity.

\(^2\)The specification is available at [19].

\(^3\)Note that this service has been proposed by the ByteIO working group only for the purpose of running the interoperability tests.
activity to construct another workflow that will let initializing the resource with the required data. The latter can be read from a file located inside a temporary directory in the server and recorded into the new Random ByteIO resource. The process is illustrated in figure 8.1.

Figure 8.1: Creation of a Random ByteIO resource for the purpose of the interoperability tests.

As soon as the Random ByteIO resource is created and initialized, the service will need to construct an endpoint reference that addresses the new resource. Unfortunately, it was not possible to use the OGSA-DAI’s built-in method `resolve()` to retrieve the resource’s EPR, since the endpoint reference type that was used by the ByteIO specification was incompatible with that of OGSA-DAI. In particular, the namespaces of the two addressing schemas were as follows:

- **ByteIO** [http://www.w3.org/2005/08/addressing](http://www.w3.org/2005/08/addressing)
- **OGSA-DAI** [http://schemas.xmlsoap.org/ws/2004/03/addressing](http://schemas.xmlsoap.org/ws/2004/03/addressing)

Changing the ByteIO implementation to use the OGSA-DAI’s EPRs was not possible, since this would have resulted in a non-interoperable ByteIO factory service implementation. The solution that was finally chosen was to construct the “ByteIO-type” of

---

*Note that the figure does not contain much detail about how the OGSA-DAI activities work. These have been described in section 4.3. The goal here was to illustrate the number of steps that are performed by the Create operation of the standard Random ByteIO factory service implementation.*
EPR and initialize it with the information OGSA-DAI expects to find inside request messages.

Endpoint references have already been introduced in section [5.1]. These provide a standard mechanism to encapsulate information about specific endpoints that a simple URL cannot provide. Information is stored inside elements of an XML document of particular structure (XML Schema). Usually, in order to address a WS-Resource, all we need is the URL of the instance service that manages the resources and the resource id. Furthermore, all EPR information will be finally copied into the Header of the SOAP message that will be sent to the recipient service.

Using the OGSA-DAI’s addressing schema, a resource can be referenced by placing its unique identifier inside the element ReferenceProperties of the endpoint reference XML. The resource id is wrapped inside a child element, named ResourceID and associated with the http://ogsadai.org.uk namespace. Upon receiving a client request, the OGSA-DAI server will look inside the header of the SOAP message to find the particular prefixed element ResourceID and retrieve its value, which should be the resource identifier.

In the case of the ByteIO’s addressing schema, there is no ReferenceProperties element inside the EPR’s XML. Information such as the resource id is stored inside a different element, named ReferenceParameters. However, the endpoint reference information will still be included inside the header of the SOAP message and ensuring that this is the one that OGSA-DAI expects, then there should be no problem processing the SOAP message whatsoever.

Figure 8.2 illustrates a response from the factory service that has been implemented when the createResource() operation was invoked. Notice the structure of the EPR’s XML document and that the resource id is stored inside the ReferenceParameters element.

```
<createResourceResponse xmlns="http://schemas.ggf.org/byteio/2006/07/interop">
  <ns1:EndpointReference xmlns:ns1="http://www.w3.org/2005/08/addressing">
    <ns1:Address>http://localhost:5555/dai/services/RandomByteIOService</ns1:Address>
    <ns1:ReferenceParameters>
      <ns2:ResourceID xmlns:ns2="http://ogsadai.org.uk">ogsadai-113d02cdc02</ns2:ResourceID>
    </ns1:ReferenceParameters>
  </ns1:EndpointReference>
</createResourceResponse>
```

Figure 8.2: Response message of a "CreateResource" operation as it was recorded by the TCPMonitor tool.

Figure 8.3 illustrates the header of the SOAP message that was sent to the Random ByteIO service for performing a Read operation on the resource that was constructed
before. Note the ResourceID element that has been added inside it.

<table>
<thead>
<tr>
<th>Request Message of a Read operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="">soapenv:Header</a></td>
</tr>
<tr>
<td>&lt;wsa:MessageID soapenv:mustUnderstand=&quot;0&quot;&gt;uuid:b9df4d40-33c7-11dc-9a9b-9e907b74e5a3&lt;/wsa:MessageID&gt;</td>
</tr>
<tr>
<td>&lt;wsa:To soapenv:mustUnderstand=&quot;0&quot;&gt;<a href="http://localhost:5555/dai/services/RandomByteIOService">http://localhost:5555/dai/services/RandomByteIOService</a>&lt;/wsa:To&gt;</td>
</tr>
<tr>
<td>&lt;ns1:ResourceID xmlns:ns1=&quot;<a href="http://ogsadai.org.uk">http://ogsadai.org.uk</a>&quot; soapenv:mustUnderstand=&quot;0&quot;&gt; ogsadai-113d02cdc02 &lt;/ns1:ResourceID&gt;</td>
</tr>
<tr>
<td>&lt;/soapenv:Header&gt;</td>
</tr>
</tbody>
</table>

Figure 8.3: SOAP Header of a Read request message as it was recorded by the TCP-Monitor tool.

The second operation that the factory service supports allows for removing a Random ByteIO resource from the server. Upon receiving the client’s request, the service will extract the resource ID from the endpoint reference that was passed as an argument to the operation and delete the resource by removing it from the Resource Manager of the OGSA-DAI server and erasing its data file as well. Any future attempts to access the deleted resource are expected to raise a ResourceUnkownFault.

The factory service can be deployed into the Tomcat web services container following the same procedure that has been used for the Random ByteIO service and was described in section 4.4. Its schema was placed inside the folder schema/ogsadai/axis-specific-wsdl/rbyteio-factory and the "build" files were edited to include the new service. The implementation class of the service was placed inside the package uk.org.ogsadai.service.axis.rbyteio_interop and was named AxisRandomByteIOFactoryService.

8.1.2 Results

The implementation of the Random ByteIO service was tested together with three other implementations. Since this had been the first interoperability test for every implementation, lots of issues arose that had merely to do with inconsistencies in the interfaces of the Random ByteIO and the factory service (i.e. the WSDLs) and different interpretations of these by the developers. The issues were recorded and solutions proposed for tackling them.
Some of these are listed below:

1. The WSDL of the Random ByteIO interface was declaring wrong data contents when the simple transfer mechanism was used. The data were recorded inside a `byteio:any` XML element instead of a `byteio:data` one.

2. It was not clear in the WSDL of the standard factory service how the `createResource` operation should be invoked. Some developers were adding data in the request message, whereas others did not.

3. Some developers interpreted the truncate-append operation wrong and did not allow truncating the resource if there was no data to append.

The client that was described in section 5.1 was used for accessing the implementations of other developers’ services. Two more commands were added to the user interface that are shown below:

**create-interop**  The command can be used for requesting that a new Random ByteIO resource is created for executing the interoperability tests.

**delete-interop**  Deletes the resource that was created for the interoperability tests.

Following the creation of a new resource, the user can invoke any of the commands the interface of the client supports already for accessing the data of the resource via the Random ByteIO service that manages this directly.

Due to the problem with the `createResource` operation that was listed before, it was not possible to create Random ByteIO resources when contacting one of the other three services. Resources were being created successfully in the case of the other two services. However the `Read` operation was failing due to unknown server-side errors. The rest of the operations appeared to work correctly since no exception was raised.

The current implementation was also tested by client programs of other developers. Two of them found that the Random ByteIO resources were being constructed correctly by the factory service. However, only one could successfully invoke the `Read` operation to read data from inside the resources. The rest operations were failing due to errors that had to do with the serialization and deserialization of the request and response messages. It needs to be made clear however that these errors appeared due to interoperability issues and not because the current implementation was faulty.

The conclusion here is that the interoperability tests can show issues that are not visible when working with the same implementation of a client and server. Although an implementation can function correct on its own, this may not interoperate with other implementations of the same standards. Achieving interoperability is an iterative process that involves lots of communication among the developers and precise descriptions of the specifications until consensus is reached at all levels and the tests are successful. The solutions that have been proposed at the end of the first interoperability fiesta have not been incorporated in the current implementation of the Random ByteIO interface, but this has been left as future work. It is also expected that as more and more users will
start accessing the Random ByteIO implementation with their own client implementa-
tions, additional interoperability issues will arise. The first fiesta proved that standard
mechanisms need some time to "mature" before these can fulfil their aims.
Chapter 9

Conclusions

With a view to providing efficient and flexible mechanisms for accessing and integrating data over the Grid, the current project implemented and connected the Random ByteIO interface to OGSA-DAI 3.0, the key features of which are as follows:

- This is a standard interface that has been specified by the ByteIO working group of the Global Grid Forum.
- This allows partial random access to data resources over the Grid.
- This maps easily to existing APIs, such as POSIX.
- This provides efficient mechanisms for transferring binary data.

The interface was mapped by the ByteIO authors to a WSRF-compliant web service and this was implemented on top of OGSA-DAI and deployed to the Tomcat web services container. Access to the Random ByteIO resources can be accomplished in two ways. A user can either contact the Random ByteIO service directly and invoke one of the operations that the interface provides, or alternatively request that the resources are accessed via specific OGSA-DAI activities. The first is possible since any remote Random ByteIO resource can be made publically available via a unique endpoint reference. This also allows clients to access Random ByteIO resources in standard ways, while being ignorant of the underlying OGSA-DAI managing middleware. The second way is considered OGSA-DAI specific, since it implies that the client needs to construct an OGSA-DAI data workflow and submit this to a data request execution resource.

A number of activities have been developed that allow accessing the data of the Random ByteIO resources via OGSA-DAI workflows. Due to the large-scale, distributed nature of data in the Grid, this is considered beneficial for numerous applications in the scientific and industrial spectrums. Using the combined functionality of OGSA-DAI and the Random ByteIO interface, scientists can request that complex data workflows are executed on the server-side that will operate on subsets of data resources, resulting in enhanced performance. Some of these applications have been discussed in this document.
Use case scenarios of the Random ByteIO interface have been implemented to demonstrate the potentials of the new functionality. These involve basic operations for querying, updating, inserting and copying data from structured collections (i.e. databases) over the Grid. In more detail, the following scenarios have been investigated:

1. Query random parts of binary data from a remote database BLOB in an asynchronous manner.
2. Update random parts of a database BLOB by inserting data to the latter that originate from the requesting client or a third party data provider.
3. Copy random parts of binary data from a database BLOB that is managed by one OGSA-DAI server into a database BLOB that is managed by a different OGSA-DAI server.

Unfortunately, copying the data of a Random ByteIO resource into a database has not been possible. The reason was that a core activity of OGSA-DAI that would have permitted to convert the data of the resource (bytes) into a format that would be suitable for storing these inside a database was missing. However, what the Random ByteIO interface concerns, the use cases have been successfully implemented. Given the missing activity that is scheduled to be included in the next releases of OGSA-DAI, it can be argued that a database is updated and random parts of data are copied from one database to another successfully.

For implementing the above use cases and allowing a user to utilize the new Random ByteIO functionality in general, a client application has been developed. This provides an interactive interface for acquiring the user’s requests and converting these into appropriate operations for executing on the server-side. The capabilities of the client are defined by separate classes that serve as plug-ins. These encapsulate functionality for building particular user requests and submitting these to the server. In this sense, the client application is considered extensible software. Currently, the client supports plug-ins for invoking operations of a Random ByteIO service directly and for constructing and submitting a number of OGSA-DAI data workflows.

The importance of interoperability was made apparent during the course of the project. Code has been developed so that the Random ByteIO implementation can participate in the interoperability tests that are organized by the ByteIO working group. This involved the creation of a standard factory service for Random ByteIO resources, which was defined inside the specifications document of the interoperability tests. The service managed to partly interoperate with other implementations in the first interoperability fiesta. The attempt is considered successful however, since achieving interoperability has not been in the initial plan of this project and also this requires a lot of communication among the participating developers and refinements of the initial specifications, which could not fit inside the time slot that was available.

In summary, a new access mechanism has been added to OGSA-DAI that provides standard, flexible and efficient ways for accessing remote data sources over the Grid. This has been the aim of the project and was successfully accomplished.
9.1 Future Work

The current Random ByteIO implementation only supports the basic transfer mechanism, which delivers data inside the body of SOAP messages. This is a highly inefficient way to transport large binary data, merely due to the "Base64" text format encoding. The implementation can be extended by implementing and connecting to it more efficient transfer mechanisms like MTOM and DIME that transfer data inside MIME attachments and avoid the "Base64" encoding. This is expected to increase the achieved performance significantly when large binary data are accessed.

The operations of the Random ByteIO interface have been implemented on top of the Random Access Files I/O mechanism of Java. This is unbuffered by default and can therefore result in low performance when accessing data that are stored close to each other in memory. The current implementation can be extended by implementing a buffered wrapper on top of the Random Access Files I/O. This has been discussed in subsection 5.2.1.

The completion of the interoperability tests have also been left as future work. It is expected that this will not be a difficult task to accomplish given the implementation of the standard factory service that the specifications document defined.

Finally, the Streamable ByteIO interface will need to be implemented to complete the implementation of the ByteIO specification. The procedure of implementing and connecting this to OGSA-DAI is not expected to be very different from that of the Random ByteIO.
Appendix A

Project Management

For keeping the project on track and managing the risks that were associated with it, it was important that a good process would be maintained. The process was applied early from start of the project by assessing risks and creating a work plan. This chapter explains the changes that were made to the initial plan and the new risks that have been assessed and controlled.

A.1 Project Plan

The project lasted 16 weeks and was initially broken down into the following work packages (tasks):

- Background Reading.
- Implementation of the Random ByteIO interface for file access.
- Connection of the implementation to OGSA-DAI.
- Development of client.
- Execution of use case scenarios.
- Dissertation writing

Each work package represents a subset of the project and encapsulates a specific functionality that needs to be implemented. Chapters 3, 4, 5 and 6 of the dissertation describe the implementations of some of the work packages listed above. What is more, all work packages were successfully accomplished.

During the course of the project a new work package was added that corresponded to the interoperability tests and was not in the initial plan. This was made possible, since some of the initially described tasks were accomplished faster than it had been estimated. However, there was a major risk associated with this decision. It was difficult to
estimate the size of work that would be needed for executing the interoperability tests. In chapter it was shown that there was a difficulty handling the "ByteIO" type of endpoint references and this needed considerable time to investigate and solve. What is more, testing for interoperability is an iterative process that involves lots of communication among the participating developers and time. Some issues that had been identified early were fixed, but not all of them. This would have taken more time than was initially assigned for the interoperability tests and would have thus put the project in risk of not hitting its deadline. Nevertheless, for the reasons that are described in chapter it is believed that the attempt has been successful.

Some risks that were assessed and controlled during the project are given below:

**Many OGSA-DAI distributions** OGSA-DAI 3.0 was released the same month that the project commenced. As such, it was very likely that the initial version of the middleware would be updated (e.g. for correcting bugs) during the course of the project. Indeed, the OGSA-DAI team released three updates until the deadline of the project. Each time a new release was made available, the implementation of the Random ByteIO interface needed to be incorporated in the former, and this could take a significant amount of time to accomplish. The risk was mitigated by creating shell scripts that would automate the process of retrieving and installing the new OGSA-DAI distribution and then copying the implementation of the interface to it. Also, a history was kept with the exact edits that needed to be made to the build files of OGSA-DAI to include the new Random ByteIO code.

**No OGSA-DAI documentation** There was no documentation on the new version of OGSA-DAI 3.0 until the end of June. This was a major risk that could have resulted in project failure. The risk was eliminated by changing the work plan and overlapping the tasks of implementing the Random ByteIO interface and connecting it to OGSA-DAI. This way, it was made possible to start investigating OGSA-DAI 3.0 earlier, while the operations of the interface were being implemented. The decision was correct, since it was found that significant amount of time was needed to figure out how the new OGSA-DAI worked, involving project meetings and discussions with the OGSA-DAI development team. If these have not been made early, the project would have been stalled at the "connection" stage of the interface and put in risk of not hitting its schedule.

It should also be noted that key role for the success of the project and the mitigation or elimination of the risks played the regular project meetings. These served as technical reviews, involving code reading and inspections that helped detecting and controlling defects upstream.
Appendix B

Java I/O

These mechanisms can be grouped into the following three main classes:

- Java I/O Streams
- Random Access Files
- Java NIO

The first class of I/O mechanisms allows accessing data from memory in a serial fashion. Data flow inside a stream that connects the consumer with the provider in FIFO order, starting at a particular offset (memory address) that represents the start of the stream. What is more, the stream can only be accessed forwards, eliminating the possibility of reading data backwards when using one stream only.

Java supports various types of streams for different kind of data. Byte streams can be used to access one byte of data at a time and are considered low-level I/O, since all other Java streams are implemented on top of them. Character streams allow accessing character data while helping developers write portable codes by transferring the burden of adapting to the local character set to the Java core. Java also supports data and object streams. The first allow accessing primitive data types (e.g. boolean, int, long, float, etc..), whereas the second ones can be used for more complicated, user-defined types.

The streams described so far are considered unbuffered. That is, only one byte, character or any other kind of data type is accessed at a time from within the stream and this operation is handled directly by the OS. However, memory accesses (especially disk accesses) are very costly and can downgrade the performance of the code significantly, especially when large size of data is required by the application. A solution to this problem is to store a larger chunk of data temporarily inside buffers close to the CPU (e.g. in cache). This can minimize the amount of disk accesses (and TLB misses) required and increase the number of cache hits, resulting in enhanced performance. Buffered streams are built on top of regular streams by using wrapper classes that implement the buffers and provide methods to access these.
The second I/O mechanism allows non-sequential access to files, as its name suggests. In this case the data is accessed directly from a file residing in the local file system, using a totally different approach from that of streams. This supports the notion of a file pointer that can be shifted to any given position into the file to mark the starting offset of a read or write operation. Unwanted data can be this way skipped and the file may be accessed both forwards and backwards in a totally random manner.

The last I/O mechanism of Java that has been investigated is called New I/O (NIO) and was introduced in Java 1.4. The NIO packages were mainly created for use in HPC, where memory accesses can become a significant bottleneck in performance. Data flow inside streams again and the goal here is to maximize the amount of data accessed, while minimizing any CPU loss. Factors such as memory alignment and page characteristics are taken into consideration for increasing the number of cache hits and threads are employed to perform non-blocking I/O. The latter is also necessary for constructing scalable servers that will be able to satisfy more than one client requests at a time (i.e. allowing simultaneous reads in different threads).

Table B.1 summarizes some of the pros and cons of the Java I/O mechanisms that were described above.

<table>
<thead>
<tr>
<th>Java I/O</th>
<th>Buffering</th>
<th>Random Data Access</th>
<th>Non-Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java I/O Streams</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Random Access Files</td>
<td>possible</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Java NIO</td>
<td>X</td>
<td>-</td>
<td>X</td>
</tr>
</tbody>
</table>

Table B.1: Characteristics of several Java I/O mechanisms
Appendix C

Client

C.1 User Interface

The user can interact with the Random ByteIO resources via a simple client that provides a command line interface for acquiring the user’s requests. These are translated to specific operations that must be executed on the server side. The following commands are supported by the client:

- **create [resource id]**
  A user can request that a new Random ByteIO resource is created by executing the `create` command. This takes as an optional argument the id that the new resource should have. It has already been noted that resource creation is not defined inside the Random ByteIO specification and this is handled by the OGSA-DAI server via activities. The client constructs an OGSA-DAI workflow by adding the `CreateRandomByteIOResource` activity to it and submits this to the Data Request Execution Resource. Upon successful completion of the operation, the resource id is returned to the client and this is stored inside a variable for referencing the resource in future operations.

- **copyfile**
  A user can request that data are copied from some file residing in the server’s file system into the Random ByteIO resource by executing the `copyfile` command. The client constructs an OGSA-DAI workflow that reads data from a file and appends these into the Random ByteIO resource in a stream-like fashion. Details such as the location of the OGSA-DAI server and the file are hardcoded in the client can be changed by editing its source code. Moreover, the id of the resource that the "copy" operation concerns is automatically loaded by the client from the variable that this is stored.

- **writeblob**
  A user can request that data are extracted from an SQL database and stored inside the Random ByteIO resource by executing the `writeblob` command. Again, this
operation is executed by submitting a workflow of activities to OGSA-DAI, that allow fetching the data from the remote data resource (i.e. database), and storing these inside the Random ByteIO resource.

- **complexInsert**
The complexInsert command allows inserting data into a Random ByteIO resource by submitting a workflow of activities to the Data Request Execution Resource. The data may be read from any Random ByteIO resource, local or remote, that its endpoint reference is known, and these are stored inside the target resource using the implementation of the Write operation of the Random ByteIO interface, described in section 3.

- **read <startOffset> <bytes per block> <numBlocks> <stride>**
The read command allows accessing the standard interface of the Random ByteIO service to invoke its read operation for fetching chunks of data from the resource. The user must provide here all arguments that the Read operation expects, as these were described in section 3.

- **write <startOffset> <bytesPerBlock> <stride> [<data>]**
The command allows invoking the Write operation of the Random ByteIO service for inserting data into a Random ByteIO resource. The user must provide all arguments that the operation expects. Data can be added to the request message in two ways here. A user can provide the data from the command line as a single word (*This is used merely for testing purposes.*) , or load these from some local file. The location of the file is hardcoded in the client and for this to be read the user must not pass any data from the command-line.

- **append [<data>]**
The append command allows invoking the corresponding operation of the Random ByteIO service to append data to the resource. Again, data can be either given from the command-line as a single word, or loaded from a local file.

- **truncappend <offset> [<data>]**
Similarly to the previous commands, truncappend allows invoking the TruncateAppend operation of the Random ByteIO standard interface to truncate and optionally append data to the resource. The user must provide the offset to which to truncate the resource and optionally a single word that contains the data that will be appended. Specifying no data is allowed, since then the resource should only be truncated according to the Random ByteIO specification and its implementation described in section 3.

- **create-interop**
Requests that a new Random ByteIO resource is created for running interoperability tests by contacting the standard factory services.

- **delete-interop**
Requests that the Random ByteIO resource that was previously created for running interoperability tests is deleted in the server.
• quit
  Exits the client.

C.2 Adding a new plug-in to the Client

New plug-ins can be easily added to the client following the steps below:

1. Add a new command to the interaction loop of the Client class. The command will be entered by the user to request that some method of the new plug-in is invoked.

2. Create the class of the plug-in, its constructor and the method that will be called when the user enters the above command.

3. Add code inside the interaction loop for creating an instance of the plug-in class upon entering its corresponding command. Then invoke the method of the plug-in. Optionally, it is possible to pass arguments to the plug-in that may derive from previous client executions, such as the resource ID, the resource EPR, returned data from a Read operation and so on.
Appendix D

Notes on the interoperability requirements

The WSDL document that the ByteIO authors have published does not define how the messages that will be exchanged between clients and servers will be represented on a particular communication protocol (i.e. SOAP). However, both ends will need to agree on a specific message representation if they are to interoperate. For the purpose of executing the interoperability tests, it was decided that all implementations should use the "document/literal" message encoding and version 1.1 of SOAP. The Random ByteIO implementation of this project has been using the recommended style, as it was described in section 3.7.

Another issue that was encountered had to do with the creation of the ByteIO resources. The ByteIO specification does not define any particular interface of a service that creates ByteIO resources but leaves this to the implementors to decide, subject to their needs and the existing technology the ByteIO service will be connected to. However, for the purpose of simplifying the process of carrying out any interoperability testing, a standard ByteIO factory service was needed that would allow clients to create or destroy ByteIO resources in a uniform way.
Bibliography


[14] Global Grid Forum Web Site
http://www.ogf.org/ [July 2007]


[16] Web Services Interoperability Organization website,
http://www.ws-i.org/ [July 2007]

[17] WSDL Bindings Explained,

[18] OGSA ByteIO Working Group: Use Cases Document,


http://www.w3.org/TR/soap12-mtom/ [August 2007]

[21] Direct Internet Message Encapsulation (DIME)
http://msdn.microsoft.com/msdnmag/issues/02/12/DIME/default.aspx

[22] Portable Operating System Interface
http://standards.ieee.org/regauth/posix/ [December 2005]

[23] The OGSA-DAI project
http://www.ogsadai.org.uk/ [August 2007]


http://www.w3.org/Protocols/rfc959/ [October 1985]

[26] Simple Mail Transfer Protocol (SMTP) - RFC
http://www.faqs.org/rfcs/rfc2821.html [April 2001]

[27] Web Services Description Language (WSDL) 1.1 - RFC
http://www.w3.org/TR/wdsl [March 2001]

http://www.globus.org/wsrf/ [April 2006]
[29] Borja Sotomajor: GT4’s programmer tutorial
   http://gdp.globus.org/gt4-tutorial [November 2005]

[30] CERN Grid Cafe
   http://gridcafe.web.cern.ch/gridcafe/ [July 2007]

[31] Apache Tomcat
   http://tomcat.apache.org/ [August 2007]