

Accessing external data resources from DEISA Grid

Document Title	Accessing external data resources from DEISA Grid
Authorship	
Document Filename	GlobalDatabaseAccessHandout.doc
Document Version	1.0
Distribution Classification	Public

Document History

<i>Personnel</i>	<i>Date</i>	<i>Comment</i>	<i>Version</i>
EPCC	9/OCT/07	<i>First release version.</i>	<i>1.0</i>

Purpose of this Document

This document is a companion to the DEISA training session presentation that explains how to access an external database from within the DEISA Grid. The approach that is described herein uses a grid technology called OGSA-DAI.

The source code that is reproduced in this document can be downloaded from <http://www.deisa.org/training/Bologna2007/>.

Contents

Accessing external data resources from DEISA Grid.....	1
Purpose of this Document.....	2
Contents.....	2
List of Figures.....	3
Related documents	3
1 What we mean by ‘access to external data resources’	4
2 How to access a database from DEISA?	6
3 Prerequisites.....	8
4 Enabling Access To An External Database.....	8
4.1 Install the OGSA-DAI server and register database	9
4.2 Install the OGSA-DAI client on DEISA.....	10
4.3 Run an OGSA-DAI client.....	11
4.4 Write you own OGSA-DAI client.....	13
5 Summary	16
Appendix A Confirming your execution site supports external database access.....	17
A.1 Test for Java on execution site.....	17
A.2 Test the network connection	18

List of Figures

Figure 1: Traditional flow of data in an HPC application.	4
Figure 2: Data held in databases cannot be staged into an HPC computation in the same manner as used for file-based datasets.....	5
Figure 3: A DEISA job accessing a remote database over the internet.....	6
Figure 4: Global Database Access using OGSA-DAI.....	7
Figure 5: Sample OGSA-DAI resource descriptor for a MySQL database	9
Figure 6: Sample OGSA-DAI login descriptor file.....	9
Figure 7: Script to install the OGSA-DAI client runtime.....	10
Figure 8: SAGA job script to submit the OGSA-DAI client runtime installation script to the execution site.	10
Figure 9: Process to execute the OGSA-DAI client runtime installation scrip via the DESHL	11
Figure 10: Script that executes a sample MySQL query on execution site.....	12
Figure 11: SAGA job script to submit the OGSA-DAI client to the execution site.....	12
Figure 12: Sample command line session to run a simple MySQL query.	13
Figure 13: Replacing repeated calls to the SQL client with a simple Java client, which runs for the duration of the HPC job.....	14
Figure 14: A simple OGSA-DAI workflow incorporating the ‘Tee’ activity.	15
Figure 15: Example of how one might insert one’s own activity into a workflow.	15
Figure 16: Script to retrieve the version for a particular Java installation.	17
Figure 17: SAGA job script to submit the Java installation test script to the execution site.....	17
Figure 18: Process to execute the Java installation test script via the DESHL.	18
Figure 19: Java program to test the connection from the execution site to the OGSA-DAI server.	19
Figure 20: Script which executes the connection test program.	20
Figure 21: SAGA job script to submit the connection test program to the execution site.	20
Figure 22: Log of command line session used to submit the Connection Test job to execution site....	21

Related documents

- [1] [The OGSA-DAI Project website](http://www.ogsadai.org.uk/) - <http://www.ogsadai.org.uk/>
- [2] [DESHL](http://forge.nesc.ac.uk/projects/deisa-jra7/) - <http://forge.nesc.ac.uk/projects/deisa-jra7/>
- [3] [MySQL](http://www.mysql.com/) - <http://www.mysql.com/>
- [4] [SAGA](https://forge.gridforum.org/projects/saga-rg/) - <https://forge.gridforum.org/projects/saga-rg/>
- [5] [Axis](http://ws.apache.org/axis/) - <http://ws.apache.org/axis/>
- [6] DEISA Primer - <http://www.deisa.org/userscorner/primer/primer.php>.
- [7] OGSA DAI Version documentation - <http://www.ogsadai.org.uk/documentation/ogsadai3.0/>.
- [8] BioJava Website – <http://biojava.org/>.

1 What we mean by ‘access to external data resources’

From a traditional viewpoint, a high performance computing (HPC) application has a very simple composition in terms of data and computations. Figure 1 illustrates the flow of data that one normally associates with an individual run of an application (usually referred to as a *job*). Firstly, input data (stored in files) is transferred from the user’s client machine onto the target HPC resource. Then, the computational body of the job is executed. Finally, the output data (again, stored in files) is retrieved to the user’s client machine.

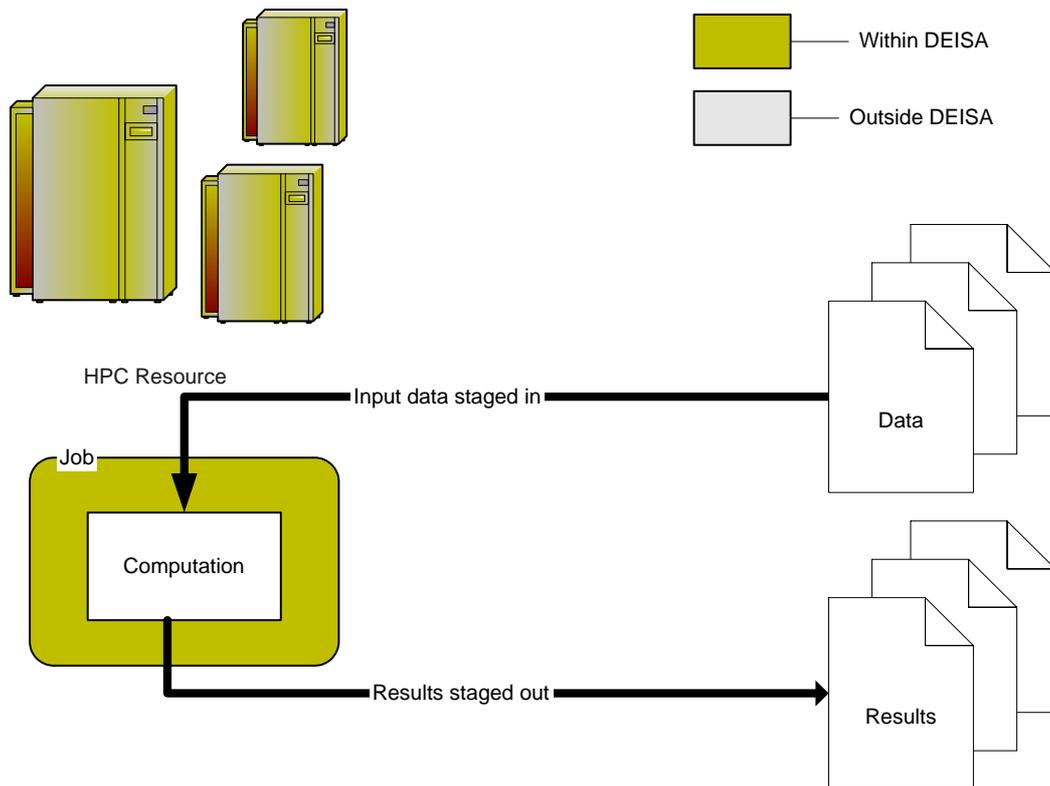


Figure 1: Traditional flow of data in an HPC application.

This workflow suggests a clean delineation between the provision of data, the computational kernel, and the retrieval of results. However, the validity of this separation of the three steps has been challenged by some of the projects with which the DEISA activity has engaged – scientific applications that have not historically been regarded as users of HPC.

As an example, we consider life science applications. These applications are commonly subject to a very different model for data management, with a need for more dynamic access to both input and output data sources – which are often provided as databases – during the body of the computation. Unlike file-based datasets, it is generally not possible to stage whole databases onto/out of the HPC resource at the beginning/end of a computation (see Figure 2).

The DEISA Grid offers a range of facilities over and above those typically encountered in an HPC environment, which can alleviate this problem. From the perspective of file management, the provision of the GPFS file system delivers a significant improvement over the traditional model, allowing a single instance of data files to be accessible from multiple sites, including both the execution site and the user’s home site. This, in turn, means that commonly used, public databases (from life sciences) can be hosted on the DEISA GPFS file system, and made directly accessible to jobs on compute resources.

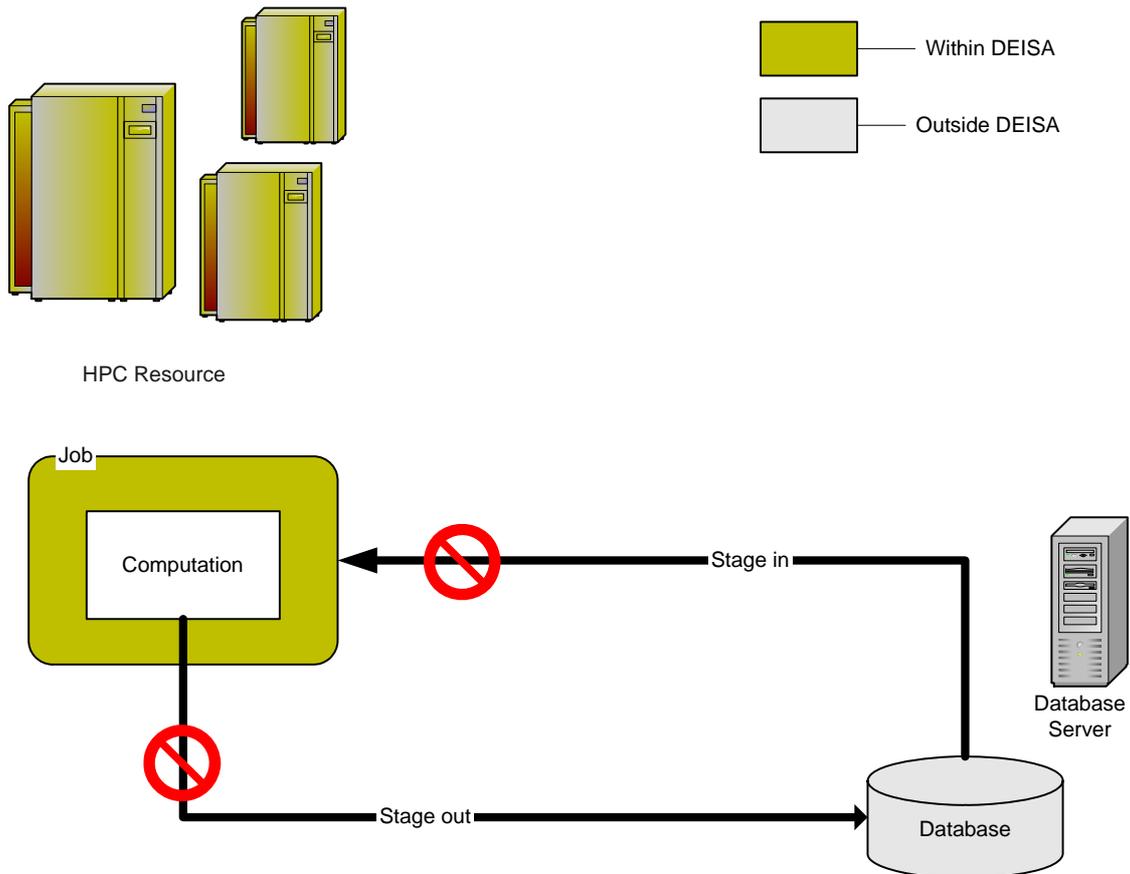


Figure 2: Data held in databases cannot be staged into an HPC computation in the same manner as used for file-based datasets.

However, this does not accommodate the case when an application relies on one database (or possibly multiple databases) that cannot be hosted on the DEISA file system: for example, a database that is proprietary to a particular institution or that receives contributions from multiple projects, including some not supported by DEISA. For such cases, the only viable approach is to set up a dynamic connection from the DEISA compute resource to the remote database (see Figure 3).

In this document, and the accompanying tutorial material, we describe a method for accessing databases, outside the DEISA infrastructure, from within a job that is running on a DEISA super-computing resource. The intention might be to: retrieve data from the database, update data in the database, or perhaps both. Such a mechanism, supporting interaction with external data sources from within the DEISA infrastructure, has the potential to make a vast amount of existing scientific data available to applications that run on DEISA Grid.

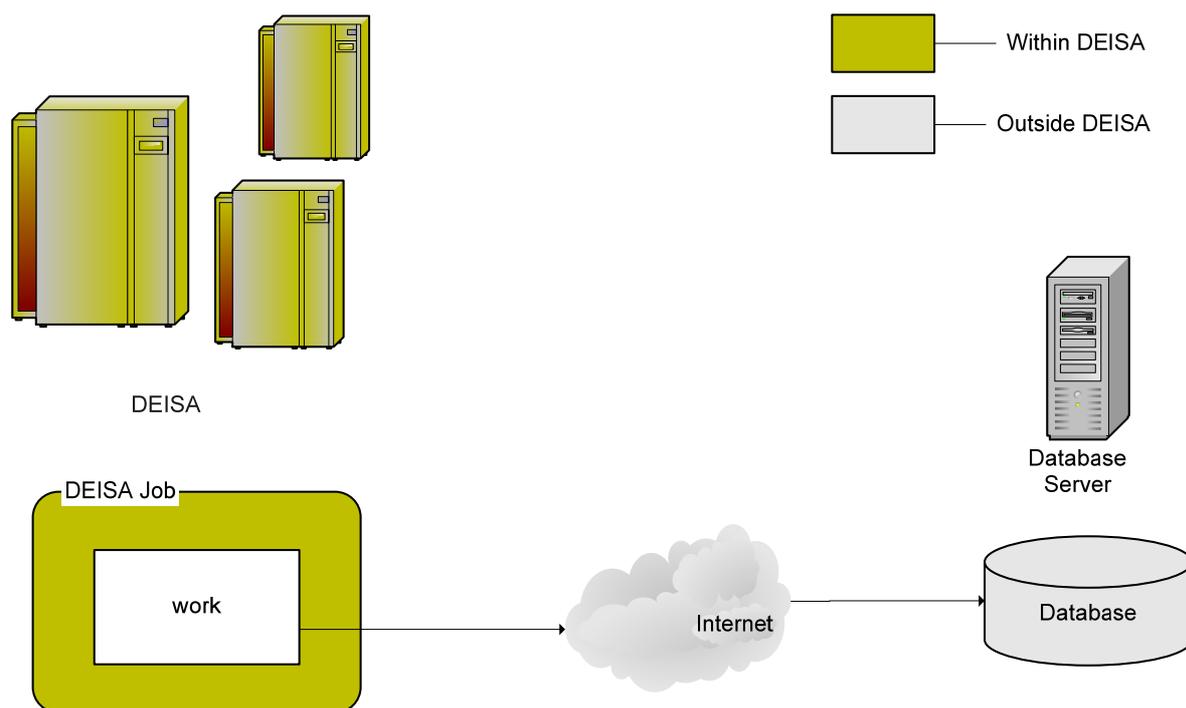


Figure 3: A DEISA job accessing a remote database over the internet.

To support our description, we refer to a real DEISA application, which requires external database access. The GTG project, which has the subtitle “Organisation of protein sequence space for efficient database searching”, uses a new paradigm for protein sequence database searching that is based on a pre-processing of the sequence data to promote high-order transitive searches in the most promising directions. Leveraging DEISA resources, the GTG team have planned to update an important database, called PairsDB, which is used to drive bioinformatics applications being developed by other activities within the parent research group. The PairsDB database is hosted on a MySQL server, which is installed at the home institution of the project team.

The computational core, of the GTG application, involves the completion of BLAST¹ searches for each of a large collection of non-redundant peptide sequences. The task equates to approximately 22,000 jobs; each job running a batch of 50 searches. It is the results from these searches that are stored in the PairsDB database.

The most efficient way to store these results is to encapsulate the database update operations into the actual job that runs on the HPC resource (which in the case of the GTG project is the execution site at IDRIS). This is the approach that is explained in the remainder of this report.²

2 How to access a database from DEISA?

Databases are commonly classified according to their types. For example, there are relational databases (such as MySQL and Oracle), and XML database (such as eXist and Xindice). Each particular database is supported by a set of client tools and libraries, which can be used to access and manipulate the data contained within.

One possible way to access an external database would be to deploy the appropriate client tools and libraries onto the DEISA execution site, so that they are readily available to the user’s job. However, this approach introduces a number of potential problems:

¹ BLAST (Basic Local Alignment Search Tool) is a set of similarity search programs designed to explore all of the available sequence databases for both proteins and DNA.

² The use case of this report has been developed in collaboration with the GTG project team. Unfortunately, in actuality, the PairsDB updates that are described above have been completed outside of the DEISA infrastructure; a path necessitated by their project deadlines.

- Given the range of different databases that are available, attempting to provide client tools for even the most commonly used, could easily become an onerous task.
- Database clients are intended to be run on desktop machines, rather than super-computers. Trivial requirements, such as availability of standard Perl/Java libraries, are likely to be non-trivial to satisfy on an HPC platform.
- Clients typically talk to remote databases on a specific port – for example, by default a MySQL server listens for client requests on port number 3306. Each different database is likely to use a different port number, and the choice of port is determined at the database host site, implying that the end-user will have little or no control over the decision. In order that a client can talk to the remote database on the preferred port, the firewalls at both the HPC resource and the database site may need to be modified. This is, at least, an inconvenience and may even be prohibited by security policy.
- For security reasons, database servers typically only accept access requests from a defined set of client machines. Given that a user job, submitted to the DEISA Grid, can run on one of a huge number of nodes, it may be problematic to correctly configure the database server's access list to support all possible request sources.

The solution that we propose aims to support as many databases as possible, within realistic effort constraints and without potentially compromising the security of the system. The solution utilises a technology called OGSA-DAI (Open Grid Services Architecture - Data Access and Integration), which provides both a client toolkit and extensible server for building data-centric workflows and which supports many popular databases.

Figure 4 shows a DEISA job accessing a remote database over the internet using OGSA-DAI technology. An OGSA-DAI server, which is installed in a web services container, such as Tomcat/Axis [5], is set up on or near to the database server, and configured to allow an OGSA-DAI client, running inside the DEISA job, to connect to the remote database.

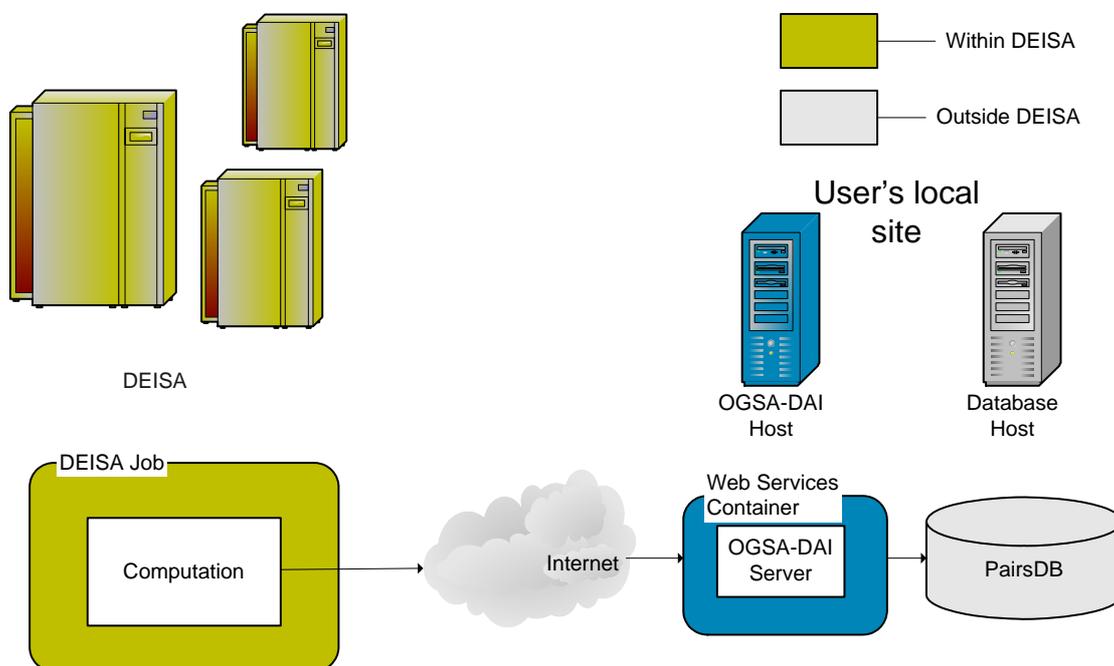


Figure 4: Global Database Access using OGSA-DAI.

The advantages of using OGSA-DAI to interface between the DEISA compute resource and the remote database are, as follows:

- Only one client – specifically, the OGSA-DAI client – needs to be made available at the DEISA execution site, irrespective of the type of target database³. The specifics of communicating with any particular database are handled by the OGSA-DAI server.
- The OGSA-DAI client is a Java application, which simply requires a Java Virtual Machine (JVM). A JVM is part of the DEISA Common Production Environment (DCPE), so is generally available on DEISA compute resources.
- The OGSA-DAI client and server can communicate on a user-specified port (port 8080 is a common choice). This makes firewall configuration much simpler, at both ends.
- The remote database receives communications from the DEISA resource via the OGSA-DAI host only. Therefore, only this one host needs to be added to the list of valid client machines in the database configuration.

OGSA-DAI also supports flat file databases. However, using it in this way may not be the most efficient means of access data in remote files.

3 Prerequisites

The approach to external database access that is described in this report assumes a number of prerequisites, on the part of the user, as described below.

- It is assumed that you are a DEISA user and can submit jobs to DEISA using either the UNICORE client or the DESHL. In this report, we use the DESHL⁴ [2] and you should consult the DEISA Primer [6] to confirm that your user environment is set up correctly.
- It is assumed that you have selected a DEISA execution site that satisfies the requirements for the OGSA-DAI client. Specifically:
 - The execution site provides a Java Virtual Machine – consult the DEISA Primer for instructions on how to set up your environment for Java.
 - A route exists from the compute nodes of the execution site to the internet (that is, to the OGSA-DAI server host). This is the case for the majority of sites, though not for all.

In this example, we will assume that IDRIS is the intended execution site. Refer to Appendix A for instructions on how to test that your execution site supports OGSA-DAI.

- The database you wish to access must be one of the OGSA-DAI supported databases⁵. For this example we use a MySQL database (Version 5.0), which is set up at EPCC on a dedicated machine called `pairsdb.epcc.ed.ac.uk` and which is hosting the PairsDB database.
- The solution proposed in this example requires an OGSA-DAI server to be installed on or close to the database host – by ‘close to’, we mean within the same administrative domain.⁶ The host needs to meet the OGSA-DAI server prerequisites [7]. In this example, we use the server distributed as part of OGSA-DAI Version 3.0, which we host on a system in EPCC called `ogsadai.epcc.ed.ac.uk`.

4 Enabling Access To An External Database

Satisfied that you can fulfil the requirements itemised in Section 3, there are two separate tasks to complete in order to actually enable access to an external database:

1. You need to set up your OGSA-DAI server (on your chosen host) and register your database as a resource of the server.

³ Provided that the target database is supported by OGSA-DAI.

⁴ Specifically, the tutorial has been developed using DESHL Version 4.1.

⁵ OGSA-DAI supported databases – see

<http://www.ogsadai.org.uk/documentation/ogsadai-wsrf-2.2/doc/dataresources/index.html>.

⁶ The proximity between the database and the OGSA-DAI server is advantageous as it eliminates firewall issues.

2. You need set up the OGSA-DAI client and to incorporate the necessary database query operations into your job code. This could be done, for example, during the enabling process.

In this section, we summarise how to complete both of these steps, using the GTG project as our example. The type of database access operation that is implemented in Step 2 is very much application-specific. As a simple illustration, we use a pre-packaged OGSA-DAI client, called 'SQLClient', to run a simple SQL SELECT query on our target database and print out the results. We also discuss how to write you own OGSA-DAI client, and highlight other features of OGSA-DAI that might be of use to you as a DEISA user.

4.1 Install the OGSA-DAI server and register database

Detailed instructions on how to install an OGSA-DAI server are available from the OGSA-DAI project website [1]. In essence, one needs to deploy the server bundle into a web services container, running on your chosen host machine.

Having installed the OGSA-DAI server, you can register your database with it. This involves creating a *descriptor file* that describes your database and exposes it on the OGSA-DAI server. An example descriptor file for exposing a MySQL database is presented in Figure 5.

```
id=PairsDB
type=uk.org.ogsadai.DATA_RESOURCE
creationTime=null
terminationTime=null
PROPERTIES
uk.org.ogsadai.resource.dataresource.product=MySQL
uk.org.ogsadai.resource.dataresource.vendor=MySQL
uk.org.ogsadai.resource.dataresource.version=5.0
END
CONFIG
dai.driver.class=org.gjt.mm.mysql.Driver
dai.data.resource.uri=jdbc:mysql://pairsdb.epcc.ed.ac.uk:3306/pairsdb
dai.login.provider=uk.org.ogsadai.LOGIN_PROVIDER
END
ACTIVITIES
uk.org.ogsadai.SQLQuery=uk.org.ogsadai.SQLQuery
uk.org.ogsadai.SQLUpdate=uk.org.ogsadai.SQLUpdate
END
dataResourceClass=uk.org.ogsadai.resource.dataresource.jdbc.JDBCDataResource
```

Figure 5: Sample OGSA-DAI resource descriptor for a MySQL database

In the first line of the OGSA-DAI descriptor file, the resource is given a resource identifier – in this case “PairsDB”. This is a meaningful name for the database so that it can be referenced by an OGSA-DAI client. The database username and password are stored in a separate file called “logins.txt” (see Figure 6).

```
id=PairsDB
userID=musername=defaultUser
password=<password>
LOGIN-END
END
```

Figure 6: Sample OGSA-DAI login descriptor file

Finally, in order to access the MySQL database we need to install a JDBC database driver onto the OGSA-DAI server. Detailed instructions on how to do this are provided in the OGSA-DAI documentation – see the OGSA-DAI project website [1].

Having set up an OGSA-DAI server and registered the target database, one may choose to verify that a connection can be made from the DEISA execution site to the OGSA-DAI server. Details of how to do this are provided in Appendix A.2.

4.2 Install the OGSA-DAI client on DEISA

The next task is to install the OGSA-DAI client onto your DEISA site. This is easily done using the *OGSA-DAI extended runtime bundle*, which contains all of the components that are required. Again, this can be downloaded from the OGSA-DAI project website [1].⁷

The installation of the runtime bundle involves uploading the package to the execution site and unpacking it. To do this, we write a shell script, as presented in Figure 7. Notice that the client is unpacked into your home directory.

```
#!/bin/bash
module load deisa

# Move to home directory.
cd $DEISA_HOME

# Unzip the runtime (in this case, OGSA-DAI client for Axis/Tomcat container).
unzip ogsadai-3.0-axis-1.4-bin-extended-jre.zip
```

Figure 7: Script to install the OGSA-DAI client runtime

The DESHL submission script to action this installation, needs only to copy the install script into the job space and execute it (Figure 8).

```
#!/bin/bash
#
# SAGA JobDefinition based directives:
#$ SAGA_JobName = Install_OGSADAI
#$ SAGA_FileTransfer =file:///install_ogsadai.sh#DEISA_HOME > install_ogsadai.sh
#$ SAGA_JobCmd = install_ogsadai.sh
#$ SAGA_JobArgs =
#$ SAGA_WallClockSoftLimit=600
```

Figure 8: SAGA job script to submit the OGSA-DAI client runtime installation script to the execution site.

In Figure 9, we present a sample command line session to illustrate the use of the DESHL to submit the job and fetch the results. Some details, such as the DESHL job identifier, will vary for different sessions. Furthermore, the details of the unpack operation have been abbreviated.

⁷ Be sure to download the runtime appropriate to your OGSA-DAI server. For example, an OGSA-DAI server hosted in a Tomcat/Axis web services container requires the OGSA-DAI Axis extended runtime, namely ogsadai-3.0-axis-1.4-bin-extended-jre.zip.

```

> deshl copy -f ogsadai-3.0-axis-1.4-bin-extended-jre.zip \
  IDRIS/home/ ogsadai-3.0-axis-1.4-bin-extended-jre.zip

> deshl copy -f install_ogsadai.sh IDRIS/deisa_home/install_ogsadai.sh

> deshl submit -q IDRIS install_ogsadai_submit.sh
Your job: IDRIS%2F-1647524339, has been successfully submitted.

> deshl status IDRIS%2F-1647524339
Job: IDRIS%2F-1647524339, has status: Done, Unicore:COMPLETED

> deshl fetch IDRIS%2F-1647524339

> cat IDRIS%2F-1647524339.out

This script was created and executed by Unicore
UNICORE - start of user output on stdout

Archive:  ogsadai-3.0-axis-1.4-bin-extended-jre.zip
  creating: ogsadai-3.0-axis-1.4-bin-extended-jre/
  creating: ogsadai-3.0-axis-1.4-bin-extended-jre/lib/
  creating: ogsadai-3.0-axis-1.4-bin-extended-jre/licences/
  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/client-config.wsdd
  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/lib/activation.jar
  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/lib/addressing-1.0.jar
  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/lib/axis-ant.jar
  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/lib/axis.jar

...

  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/licences/xerces.LICENSE
  inflating: ogsadai-3.0-axis-1.4-bin-extended-jre/licences/xmlDb.LICENSE

```

Figure 9: Process to execute the OGSA-DAI client runtime installation scrip via the DESHL

4.3 Run an OGSA-DAI client

Having installed the client bundle, as described in Section 4.2, we are ready to run a database query. To illustrate a simple use of the OGSA-DAI client, we will execute an SQL query from the DEISA execution site (IDRIS), using the `SQLClient` class that is shipped with the OGSA-DAI client bundle. The `SQLClient` takes the following arguments (see OGSA-DAI documentation for more details [1]):

- An OGSA-DAI service base URL (in this case, `http://ogsadai.epcc.ed.ac.uk:8080/dai/services/`);
- a `DataRequestExecutionResourceID` (the default, which is `'DataRequestExecutionResource'`);
- a MySQL `DataResourceID` (in this case, `'PairsDB'`, as defined in Figure 5);
- an SQL query to execute.

Next we write a simple shell script, `ogsadai_demo.sh` which calls the `SQLClient` class with the correct parameters (Figure 10). Note that we need to set up the Java Class Path for the OGSA-DAI client.

```
#!/bin/bash
module load deisa

export OGSADAI_RUNTIME_HOME=$DEISA_HOME/ogsadai-3.0-axis-1.4-bin-extended-jre

export OGSADAI_CLASSPATH=\
  $OGSADAI_RUNTIME_HOME/:\
  $OGSADAI_RUNTIME_HOME/lib/activation.jar:\
  $OGSADAI_RUNTIME_HOME/lib/addressing-1.0.jar:\
  ...
  $OGSADAI_RUNTIME_HOME/lib/xml-apis.jar:\
  $OGSADAI_RUNTIME_HOME/lib/xmlParserAPIs.jar

java -cp $OGSADAI_CLASSPATH:. \
  uk.org.ogsadai.client.toolkit.example.SQLClient \
  -u http://ogsadai.epcc.ed.ac.uk:8080/dai/services/ \
  -e DataRequestExecutionResource \
  -d PairsDB \
  -q "SELECT * FROM nrdb40"
```

Figure 10: Script that executes a sample MySQL query on execution site.

To action the query script, we create a DESHL submission script (reproduced in Figure 11) to copy the query script into the job space and execute it. In this case, ‘SAGA_NumCpus’ and ‘SAGA_NumTasks’ directives are specified, as we require this job to run on the compute nodes of the execution site.

```
#!/bin/bash
# Test job script for DESHL using SAGA.
#
# SAGA JobDefinition based directives:
#$ SAGA_JobName = OGSADAI_Demo
#$ SAGA_FileTransfer = file:///ogsadai_demo.sh#DEISA_HOME > ogsadai_demo.sh
#$ SAGA_JobCmd = ogsadai_demo.sh
#$ SAGA_NumCpus = 1
#$ SAGA_NumTasks = 8
#$ SAGA_WallClockSoftLimit=600
```

Figure 11: SAGA job script to submit the OGSA-DAI client to the execution site.

In Figure 12, we present a sample command line session to illustrate the use of the DESHL to submit the job and fetch the results. Again, some details will vary for a particular session.

```

> deshl copy -f ogsadai_demo.sh IDRIS/deisa_home/

> deshl submit -q IDRIS ogsadai_demo_submit.sh
Your job: IDRIS%2F-1647524734, has been successfully submitted.

> deshl status IDRIS%2F-1647524734
Job: IDRIS%2F-1647524734, has status: Done, Unicore:FINISHED

> deshl fetch IDRIS%2F-1647524734

> cat IDRIS%2F-1647524734.out

This script was created and executed by Unicore
UNICORE - start of user output on stdout

DRER ID: DataRequestExecutionResource
Data Resource ID: PairsDB
Base Services URL: http://ogsadai.epcc.ed.ac.uk:8080/dai/services/
SQLQuery: "SELECT * FROM nrdb40"
Client toolkit-server
class:uk.org.ogsadai.client.toolkit.presentation.axis.AxisServer@5f31d21f
- Initializing JAX-RPC handler
org.apache.axis.message.addressing.handler.AxisClientSideAddressingHandler.
..
- Initializing JAX-RPC handler
org.apache.axis.message.addressing.handler.AxisClientSideAddressingHandler.
..
Server version:3.0 [Axis 1.4]
- Initializing JAX-RPC handler
org.apache.axis.message.addressing.handler.AxisClientSideAddressingHandler.
..
- Initializing JAX-RPC handler
org.apache.axis.message.addressing.handler.AxisClientSideAddressingHandler.
..
uk.org.ogsadai.resource.request.status.COMPLETED
| one | two      | three      | four | five | six      | seven |
| 371 | 2982826 | -232.5211  | 1    | 712  | +396-2+109-3+207 | 1     |
| 371 | 2982827 | -236.5611  | 1    | 712  | +396-2+108-3+207 | 1     |
| 371 | 2982828 | -252.5611  | 1    | 712  | +396-2+209-3+207 | 1     |
| 371 | 2982829 | -232.5611  | 1    | 712  | +395-2+109-3+207 | 1     |

and so on ...

UNICORE - end of user output on stdout

```

Figure 12: Sample command line session to run a simple MySQL query.

4.4 Write you own OGSA-DAI client

The example presented in Section 4.3 is very simple, and barely touches on the range of functionalities offered by OGSA-DAI (for all types of user from a novice to an experienced user). Here we note three possible enhancements of the use case.

The first and most simple enhancement involves eliminating the repeated calls to the `SQLClient` command, by creating a simple (Java) client that runs for the duration of the HPC job and inserts datasets, into PairsDB, as they are generated (Figure 13). The advantage of this approach is that it

eliminates the overhead of repeatedly invoking a JVM (the SQLClient is a Java application), for each result set generated.

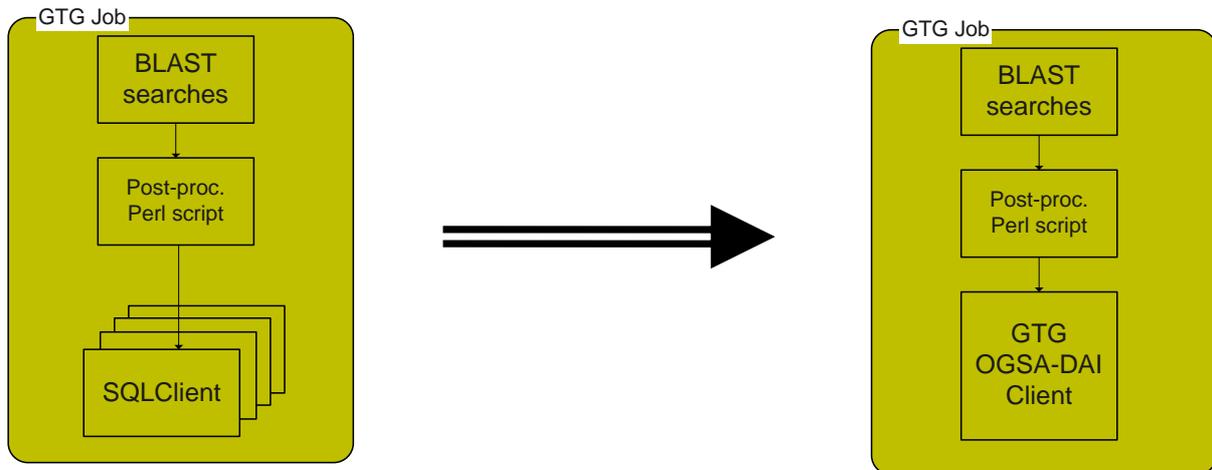


Figure 13: Replacing repeated calls to the SQL client with a simple Java client, which runs for the duration of the HPC job.

A second and slightly more advanced extension involves the creation of a simple OGSA-DAI workflow for the results. For example, one could use the OGSA-DAI ‘Tee’ activity to duplicate the results, directing one instance to the PairsDB database (as before) and directing the other instance to a log file (as illustrated in Figure 14). One could then inspect the log file, in real-time, to monitor progress of the HPC job or check the sanity of the results.

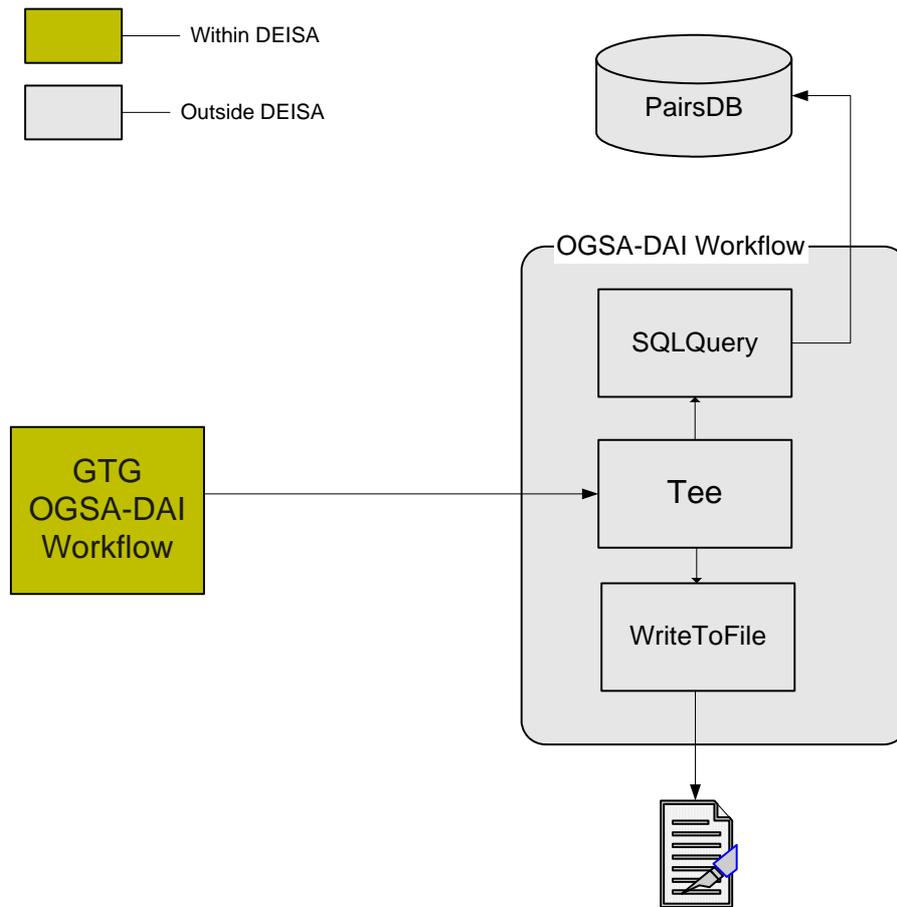


Figure 14: A simple OGSA-DAI workflow incorporating the ‘Tee’ activity.

A third and more sophisticated extension would be to create one’s own OGSA-DAI activity to manipulate the results. For a life scientist, it might be useful to incorporate the BioJava [8] library into the OGSA-DAI workflow (as illustrated in Figure 15). For example, if appropriate, one might wish to convert the BLAST results from protein sequences into nucleotide sequences.

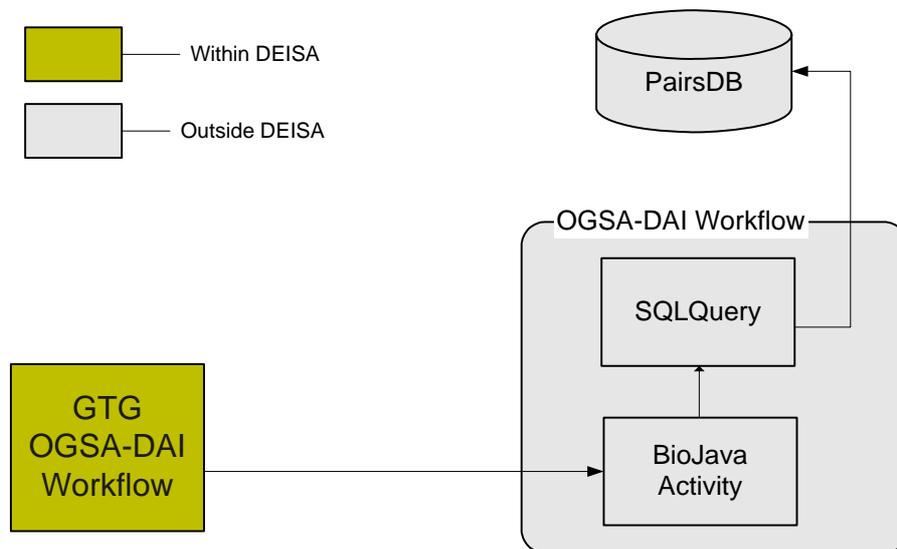


Figure 15: Example of how one might insert one’s own activity into a workflow.

For more information about OGSA-DAI, including: a tutorial on how to write your own client; full documentation; and details of up-coming training events, visit the OGSA-DAI Project website [1].

5 Summary

In this tutorial, we have motivated the need for HPC applications to access external data held on remote databases, using the actual example of the DEISA GTG project. We have demonstrated a suitable mechanism for achieving this, using OGSA-DAI, and have summarised the requirements for doing this. We have walked through a simple example using a MySQL database, which should be straightforward to reproduce, and which does not require significant programming experience on the part of the reader. Finally, we have highlighted additional functionalities of OGSA-DAI that could be used to enhance the use case.

Appendix A Confirming your execution site supports external database access

As not all DEISA sites satisfy the prerequisites for external database access (with OGSA-DAI), we provide a short summary – in this appendix – of how one might validate an individual DEISA execution site. This is a two step process that involves: checking that a Java Runtime Environment (JRE) is available, and confirming that a route can be found from the compute nodes of the execution site to your OGSA-DAI server.

A.1 Test for Java on execution site

To run an OGSA-DAI client on the execution site, a Java Runtime Environment⁸ is required. This is generally the case, as Java is part of the DEISA Common Production Environment (DCPE). One should consult the DEISA Primer [6], for information about the configuration of Java on a particular execution site.

Assuming that the environment variable JAVA_HOME has been assigned to the location of the Java installation on the target execution site, then the short script in Figure 16 will report the Java version for the installation.

```
#!/bin/bash
module load deisa java

# Check the version of java which is found.
echo "Java version"
java -version
```

Figure 16: Script to retrieve the version for a particular Java installation.

To action the script in Figure 16 on your DEISA site, we create a DESHL submission script as presented in Figure 17. Note that, in order to ensure that our job executes on the compute nodes, we request eight processors (using the SAGA flag ‘#\$ SAGA_NumTasks = 8’).

```
#!/bin/bash
#
# SAGA JobDefinition based directives:
#$ SAGA_JobName = ConnectionTest
#$ SAGA_FileTransfer = file:///connection_test.sh#DEISA_HOME >
connection_test.sh
#$ SAGA_FileTransfer = file:///ConnectionTest.class#DEISA_HOME >
ConnectionTest.class
#$ SAGA_JobCmd = connection_test.sh
#$ SAGA_JobArgs =
#$ SAGA_NumCpus = 1
#$ SAGA_NumTasks = 8
#$ SAGA_WallClockSoftLimit=600
```

Figure 17: SAGA job script to submit the Java installation test script to the execution site.

The process for uploading the script, launching the job, and retrieving the results is illustrated in Figure 18. We start by uploading the job script “java_test.sh” using the DESHL ‘copy’ command. The job is then submitted using the DESHL ‘submit’ command. The ‘-q’ flag is used to set the execution site. The location of the JVM on the execution site is specified with the DESHL ‘-v’ flag (assign the

⁸ Specifically, OGSA-DAI Version 3.0 requires at least Java Version 1.4.

location to the JAVA_HOME environment variable). When the job completes, the results are fetched using the DESHL 'fetch' command. This generates two files on the local file system, named "<job-id>.out" and "<job-id>.err". The JVM prints version information on the error stream, so to view the results we list the contents of "<job-id>.err" file. In this case, the JVM is version 1.4.2.

```
> deshl copy -f java_test.sh $DEISA_SITE/deisa_home/java_test.sh

> deshl submit -q $DEISA_SITE java_test_submit.sh
Your job: IDRIS%2F-1647524810, has been successfully submitted.

> deshl status IDRIS%2F-1647524810
Job: IDRIS%2F-1647524810, has status: Done, Unicore:FINISHED

> deshl fetch IDRIS%2F-1647524810

> cat IDRIS%2F-1647524810.err

This script was created and executed by Unicore
UNICORE - start of user output on stderr

java version "1.4.2"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
Classic VM (build 1.4.2, J2RE 1.4.2 IBM AIX build ca142-20060421 (SR5) (JIT
enabled: jitc))

UNICORE - end of user output on stderr
UNICORE EXIT STATUS 0 +
```

Figure 18: Process to execute the Java installation test script via the DESHL.

A.2 Test the network connection

The second test involves confirming the connection to the OGSA-DAI server from your DEISA execute node. To do this, we write a simple Java program that connects to the OGSA-DAI server and downloads the service description. This Java program is reproduced in Figure 19. We compile the program and create a wrapper script, called "connection_test.sh", which calls our client and passes the location of the OGSA-DAI server to it as an argument (Figure 20).

```

import java.io.*;
import java.net.*;

public class ConnectionTest {
    public static void main(String[] args) {
        if (args.length > 0) {
            String ogsadaiServer = args[0];
            String ogsadaiServicesVersionWSDL = ogsadaiServer + "/Version?wsdl";

            try {
                final URL urlWSDL = new URL(ogsadaiServicesVersionWSDL);

                try {
                    InputStreamReader input = new InputStreamReader(urlWSDL.openStream());
                    StringBuffer buf = new StringBuffer();
                    char[] charArray = new char[1024];
                    int readBytes;
                    while ((readBytes = input.read(charArray)) >= 0) {
                        buf.append(charArray, 0, readBytes);
                    }
                    System.out.println("Successfully retrieved the OGSA-DAI server WSDL.\n");

                } catch (UnknownHostException e) {
                    System.err.println("Failed to retrieve Version WSDL for the OGSA-DAI service.");
                    System.err.println("Check that the OGSA-DAI Service URL, "
                        + ogsadaiServicesVersionWSDL + ", is a valid URL.");
                    System.err.println("Possible reason: " + e.getMessage() + "");
                }

                } catch (ConnectException e) {
                    System.err.println("Failed to retrieve Version WSDL for the OGSA-DAI service.");
                    System.err.println("Check that OGSA-DAI Server is running at following port, "
                        + ogsadaiServer + ".");
                }

                } catch (FileNotFoundException e) {
                    System.err.println("Failed to retrieve Version WSDL for the OGSA-DAI service.");
                    System.err.println("Check that the OGSA-DAI Server is running at, "
                        + ogsadaiServer + " is running in an Tomcat/Axis server.");
                }

                } catch (IOException e) {
                    System.err.println("Failed to retrieve Version WSDL for the OGSA-DAI service.");
                    System.err.println("Check that the OGSA-DAI Service URL, "
                        + ogsadaiServicesVersionWSDL + ", is a valid URL.");
                    System.err.println("Possible reason: " + e.getClass().getName()
                        + ", " + e.getMessage() + "");
                }

            } catch (MalformedURLException e) {
                System.err.println("Failed to retrieve Version WSDL for the OGSA-DAI service.");
                System.err.println("Check that OGSA-DAI Service URL, "
                    + ogsadaiServicesVersionWSDL + ", is a valid URL.");
                System.err.println("Possible reason: " + e.getMessage() + "");
            }
        } else {
            System.err.println("Expected OGSA-DAI server URL to be passed as an argument. \n");
            System.exit(1);
        }
    }
}

```

Figure 19: Java program to test the connection from the execution site to the OGSA-DAI server.

```
#!/bin/bash

java ConnectionTest http://$OGSADAI_HOST:$OGSADAI_PORT/dai/services
```

Figure 20: Script which executes the connection test program.

To submit this job to the execution site, create a DESHL submission script, as illustrated in Figure 21, which we call “connection_test_submit.sh”.

```
#!/bin/bash
#
# SAGA JobDefinition based directives:
#$ SAGA_JobName = ConnectionTest
#$ SAGA_FileTransfer = file:///connection_test.sh#DEISA_HOME >
connection_test.sh
#$ SAGA_FileTransfer = file:///ConnectionTest.class#DEISA_HOME >
ConnectionTest.class
#$ SAGA_JobCmd = connection_test.sh
#$ SAGA_JobArgs =
#$ SAGA_NumCpus = 1
#$ SAGA_NumTasks = 8
.. - - - - -
```

Figure 21: SAGA job script to submit the connection test program to the execution site.

A sample command line session that submits this job and retrieves the results is presented in Figure 22. If the connection test fails for you, then firstly double-checked that the OGSADAI host and port number are correct. If they are, then there are two other likely reasons why you may not be able to connect. The first is that the DEISA site has a firewall that is blocking outgoing traffic. The second reason is that there is no route from the DEISA sites execute nodes to the internet, as the execute nodes are on a different subnet to the head node. In either of these cases, please contact DEISA User Support at your home site.

```
> deshl copy -f connection_test.sh $DEISA_SITE/deisa_home/connection_test.sh

> deshl copy -f ConnectionTest.class $DEISA_SITE/deisa_home/ConnectionTest.class

> deshl submit -q $DEISA_SITE connection_test_submit.sh
Your job: IDRIS%2F-1647524897, has been successfully submitted.

> deshl status IDRIS%2F-1647524897
Job: IDRIS%2F-1647524897, has status: Done, Unicore:FINISHED

> deshl fetch IDRIS%2F-1647524897

> cat IDRIS%2F-1647524897.out

This script was created and executed by Unicore
UNICORE - start of user output on stdout

Successfully retrieved the OGSA-DAI server WSDL.

UNICORE - end of user output on stdout

> cat IDRIS%2F-1647524897.err

This script was created and executed by Unicore
UNICORE - start of user output on stderr

Failed to retrieve the Version WSDL for the OGSA-DAI service.
Check that the OGSA-DAI Service URL,
http://$OGSADAI_HOST:$OGSADAI_PORT/dai/services/Version?wsdl, is a valid URL.

UNICORE - end of user output on stderr
UNICORE EXIT STATUS 1 +
```

Figure 22: Log of command line session used to submit the Connection Test job to execution site.