



DiGS: Design for Storage Element Support

Project Title: DiGS

Document Title: DiGS: Design for Storage Element Support

Document Identifier: QCDGRID2-D6-3-DES 1.2

Document Filename: D6.3_design_for_srm_storage_support.doc

Distribution Classification: Public

Authorship: Radoslaw Ostrowski, Eilidh Grant, James Perry

Approval List: QCDgrid Project Management Board

Distribution List: Public

Document History:

<i>Personnel</i>	<i>Date</i>	<i>Summary</i>	<i>Version</i>
RHO, JTP, MGB	24/JAN/08	First release.	1.0
EJG, JTP, MGB	5/OCT/08	Updated design.	1.1
JTP, RHO, MGB	9/MAR/10	Added description of SRM adaptor	1.2

Contents

1	Introduction	3
2	Definition of a DiGS Storage Elements.....	5
2.1	Overview	5
2.2	Storage Element Interactions.....	5
2.3	Access control and user authentication.....	6
3	What is needed over and above this and why	7
3.1	How will we meet these extended requirements.....	7
3.2	Storage Element Interfaces	8
3.2.1	Interfaces in C	8
3.2.2	Error Codes.....	9
3.2.3	Checksum types	9
3.2.4	Atomic transactions.....	9
3.2.5	Client interface.....	10
3.2.6	Control thread interface	12
3.3	Implementation of the Interface for Globus SE.....	17
3.4	Implementation of the Interface for SRM	18
3.4.1	General Implementation and Dependencies	18
3.4.2	Sharing of GridFTP Code.....	18
3.4.3	Checksum Support	19
3.4.4	Inbox Support	20
3.4.5	Implementation of Data Transfer Methods	20
3.4.6	Tape Storage Considerations.....	21
3.4.7	Implementation of Other Methods.....	21
3.4.8	Security Implementation.....	22
3.4.9	SRM-specific Node Properties.....	23
3.4.10	Testing Strategy	23
4	Changes to setup of Inbox folder.....	24
5	Special Requirements for Storage Element Providers	26
5.1	Security Concerns.....	26
5.2	Which User Management System to choose	26
6	Conclusions.....	28
7	References.....	29
Appendix A	: Feedback from AAB Meeting (28/FEB/08).....	30
Appendix B	: High level functions that use interface	31

1 Introduction

The QCDgrid project [20] is a core activity of UKQCD [19], a collaboration of UK academics and researchers that aims to procure and jointly exploit computing facilities for lattice field theory (commonly referred to as Lattice QCD) calculations. The primary aim is to increase the predictive power of the *Standard Model of elementary particle interactions* through numerical simulation of *Quantum Chromodynamics*. Such numerical simulations produce significant amounts of data and the purpose of the QCDgrid project is to provide software and supporting infrastructure that simplifies the management, storage, and manipulation of this data.

In the first three years of the project (2002—2004), software engineers at EPCC developed QCDgrid—a data management system that combines the distributed resources of the collaborators into a robust facility called the *UKQCD Grid*. The result is a multi-terabyte storage facility over sites at: University of [Edinburgh](#) (including the UoE Advanced Computing Facility), University of [Liverpool](#), University of [Southampton](#), and University of Wales [Swansea](#). The University of [Glasgow](#) is also a member of the consortium.

The facility is based on commodity hardware and open-source software. The hardware consists primarily of high specification, PC-based servers running the Linux operating system and managing large RAID storage arrays. On top of this infrastructure, the QCDgrid software (built using components from the Globus Toolkit [13], EGEE application stack [11], and an XML database) provides *Datagrid* management and user functionality – furnishing a simple and intuitive environment that hides the complexities of the underlying grid and presents a standard file system to the user. It incorporates a robustness metric that automatically disperses datasets across the grid, providing a resilience that ensures data is not affected by the loss of one (or possibly more) storage nodes. Security is leveraged from the Globus Toolkit, based on X.509 digital certificates issued by an approved Certificate Authority. The result is a reliable and secure data management system.

At the time of writing, UKQCD is beginning to diversify the types of storage resource that it utilises, in order to meet an increasing demand for storage capacity that cannot be satisfied by existing UKQCD Grid infrastructure. The main contributor of additional storage for the foreseeable future will be GridPP. The capacity that GridPP intend to provide is different from existing UKQCD storage capacity in three key ways:

- The resources will be second generation WLCG/EGEE storage elements. Access to these storage elements must be implemented via the Storage Resource Manager (SRM) protocol (Version 2.2) [21].
- The storage capacity will consist of a mixture of tape-based and disk-based storage. Both of these storage types can be managed through the SRM interface. However, from the user perspective, one needs to be aware that a significant latency may be present for operations that act on tape-based storage.
- The resources will provide a standard WLCG/EGEE interface to the user. There will be no opportunity for UKQCD to install DiGS-specific components onto these resources.

This document fulfils three objectives:

1. It provides a review of existing storage elements deployed on UKQCD Grid.
2. It investigates the necessary changes that have to be carried out in order to address the on-going data management requirements of UKQCD.
3. It describes a plan and proposes a design for extending DiGS to fulfil these changes.

This document will be of interest to developers (for example, who are modifying or extending the software) and to system administrators for troubleshooting.

Acknowledgements

The QCDgrid team would like to acknowledge the contribution of Albert Anthony, who considered options for adding SRM functionality to DiGS in his MSc dissertation [2], which contains an alternative approach to creating an SRM-compliant DiGS storage element.

2 Definition of a DiGS Storage Elements

2.1 Overview

The *Storage Element* (SE) is the component in DiGS that is used to store user datasets. It is a Unix/Linux server that provides an interface to a storage system such as a RAID (Redundant Array of Independent Disks) or SAN (Storage Area Network)¹. A DiGS-powered data grid typically has multiple SEs at geographically dispersed locations.

On each SE, a directory is provided in which all user data (designated for that particular SE) is stored. In versions of DiGS up and including 2.0, this data directory is named `data` and resides within the DiGS software installation path. Typically, it is a symbolic link to the mount point for the particular managed storage system of the SE.

Recent versions of DiGS (Version 1.2 and higher) support the definition of multiple data directories, which are named `data`, `data1`, `data2`, and so on. This allows multiple storage systems to be managed from a single server.

On a SE, each user dataset is recorded in a file, which DiGS regards as an atomic unit. This means that DiGS places no restriction on, and makes no assumptions about, the contents of each user file. DiGS does support scientific mark-up of datasets, though this is handled in a separate component, called the *Metadata Catalogue*, which does not impact on the SE.

Multiple copies of user datasets (that is, files) may be held on a DiGS system. Duplication of data is intended, providing better availability of data and fulfilling a basic, backup functionality.

Each dataset is identified by a unique and persistent label called the *Logical Filename* (LFN). The mapping between LFN and the possibly multiple physical copies of a file is managed outside of the SE, by a service called the *File Catalogue*.

The complexity introduced by having multiple copies of each file is hidden from the user, who always uses an LFN to identify a dataset.

2.2 Storage Element Interactions

Within a DiGS-powered infrastructure, two different components are required to interact with an SE, as follows:

- A management process, called the *Control Thread* (CT), queries the SE for status information, such as: the amount of free space; a list of files stored on the node; and so on.
The CT also queries the SE for more detailed status information pertaining to individual files, such as: file size; checksum; Unix permissions; and so on.
- *Clients* (and the CT) contact the SE to initiate file transfers (to/from SE). In certain cases, the Client may also request that a file be deleted.

On the SE, these requests are handled by DiGS-specific services that are built on top of the Globus Toolkit [13] – specifically GRAM, GridFTP, and GASS services.

¹ Versions of DiGS up to and including 2.0 only support *online* (that is, disk-based) storage resources. They do not support the use of *offline* (that is, tape-based) storage systems.

2.3 Access control and user authentication

DiGS Version 2.0 introduces an access control mechanism to help protect datasets that are held on the data grid. Full details are provided in [17]: below we summarise the pertinent points.

All users of a particular data grid need to be registered as members of a virtual organisation (VO) – which is normally, though not necessarily, managed by VOMS [1]. The VO is partitioned into subgroups, with each individual member being allocated to exactly one of the available subgroups. Within the data grid, datasets are classified as either "Private" or "Public": a dataset that is classified as Public can be read by anyone who is a member of the VO; a dataset that is classified as Private can only be read by other members of the same subgroup as the original submitter.

User authentication is achieved using the Grid Security Infrastructure (GSI) [14] and implemented using the *grid-mapfile* mechanism. The VO structure described above is mirrored by a configuration of Unix user accounts and groups on each SE. Specifically, each SE has a grid-mapfile that maps a user (who is registered in the VO) to a local Unix account, which belongs to a Unix group corresponding to the VO subgroup of the user.

3 What is needed over and above this and why

In connection to the UKQCD use case, there are several limitations in the current version of the DiGS software that prevent the collaboration from exploiting the full range of available storage resources. These are, as follows:

1. There is a lack of support for SEs other than those that expose GridFTP/GRAM/GASS for data management.

The DiGS source code includes hard-wired use of Globus services such as GridFTP, GRAM and GASS. Since the initial implementation of QCDgrid, an established standard interface for dealing with SEs called Storage Resource Manager (SRM) has become available. Moreover, UKQCD have been allocated storage space by GridPP on SRM-compliant storage elements. Given these developments, it is clear that DiGS would be improved by support for SRM-based SEs. Additionally, SRM support in DiGS is also important from the perspective of the International Lattice Data Grid (ILDG), who have nominated SRM as their preferred storage element interface. UKQCD, as a member of ILDG, should employ the SRM interface for greater interoperability with other collaborators.

2. Lack of support for off-line storage.

The current version of DiGS is intended for use with disk-based storage. However, offline storage, such as tape, has been historically used as the principal method of mass storage: mainly, because it is more cost effective than disk. As the resources provided to UKQCD by GridPP may include offline storage, support for tape is another consideration for DiGS.

3. Requirement for DiGS-specific software components to be deployed and maintained on SEs.

The task of maintaining a DiGS SE is made more complicated by the fact that DiGS-specific components have to be set up and configured on the SE. Additionally, even small configuration changes require the attention of SE's administrators, potentially leading to delays and unforeseen configuration problems. At the time of writing, if UKQCD were offered a SE but had no way of installing the DiGS software on it, then the SE could not be utilised. Therefore, the amount of DiGS software needed on SE should be significantly reduced or, better still, eliminated completely. This should make both the initial setup and the on-going maintenance of SEs easier.

3.1 How will we meet these extended requirements

The first problem that needs tackling is support for other types of SEs, such as SRM. A sensible way to generalise the support for SE types is to explicitly define the interface between the SE and other DiGS components. This effectively decouples the type of the storage element used from the rest of the DiGS software. Then, integrating a new type of SE, from the point of view of DiGS, would only require the implementation of the interface with the type-specific commands. This approach should allow flexible adoption of not only SRM but also other types of SEs.

Only having solved the first problem, can we move on to look at the lack of support for offline storage. As a matter of fact, the aforementioned SRM interface (for a subset of implementations, e.g. dCache [22]) is capable of employing tape storage resources. So, extending DiGS to support SRM will implicitly improve offline storage capabilities. Unfortunately, there are some further issues with this. As DiGS is not able to serve

asynchronous transfer operations, it cannot access files that firstly need to be staged from the tape. A possible solution to this is to treat the offline storage in the same way as the disk-based, except with longer timeouts.

In order to solve the third problem, we need to consider the DiGS operations that are actually implemented locally on an SE. Then, possible replacement commands that achieve the same result but that can be run remotely should be investigated. One needs to be aware that remote procedures can introduce a significant overhead in performance for an operation, and take this into consideration when selecting the appropriate operations. Section 3.2 enumerates the different operations that should be supported by SEs. Using only remote calls to implement these operations is our goal in this report.

3.2 Storage Element Interfaces

Programming to an interface effectively decouples (or modularises) the components of a software application. Moreover, it allows them to be substituted flexibly. The interfaces described in this section, will allow DiGS to use the two types of the SEs that are of immediate interest (Globus-based and SRM), and will provide a platform for adoption of new SE types in the future. A successful implementation of these interfaces should grant interoperability with any other type as well.

The interfaces are documented in the C programming language, as this is the language used to code most of the DiGS software.

3.2.1 Interfaces in C

As C is not an object-oriented language, interfaces aren't a standard feature. The interfaces will be implemented using function pointers as follows.

The interface for `digs_free_string` is given as an example here. The operations structure has a function pointer to the `digs_free_string` method.

```
/* Storage element interface. */
struct storageElement {
    char name[30];
    int is_globus;
    digs_error_code_t (*digs_free_string)(char **string);

    /* Other methods here*/
} edinburghNode;
```

Two concrete versions of `digs_free_string` for Globus and SRM are defined:

```
digs_error_code_t digs_free_string_globus(char **string);
digs_error_code_t digs_free_string_srm(char **string);
```

The function pointers are then initialised to point at the appropriate implementation:

```
// Initialise the interface to either globus or srm implementation.
if (is_globus) { //globus
    edinburghNode.digs_free_string =
        digs_free_string_globus;
} else { //srm
    edinburghNode.digs_free_string = digs_free_string_srm;
}
```

The `digs_free_string` method can be called elsewhere in the code without specifying the Globus or SRM implementation:

```
// Call digs_free_string using function pointer.
edinburghNode.digs_free_string(pointer_to_string);
```

3.2.2 Error Codes

To facilitate fault diagnosis, all of the interface methods should return one of the error codes belonging to enumeration named `digs_error_code_t` defined below. This enumeration should be extended as required. Insertion of new error codes could affect existing implementation of functions. The interface version could be increased whenever new error codes are added.

```
typedef enum
{
    DIGS_SUCCESS,                /* Operation succeeded */
    DIGS_NO_SERVICE,            /* No instance of service found */
    DIGS_NO_RESPONSE,          /* Request to service timed out */
    DIGS_NO_SERVICE_PROXY,     /* No service proxy or expired */
    DIGS_NO_USER_PROXY,        /* No user proxy or expired */
    DIGS_AUTH_FAILED,          /* Authorisation denied */
    DIGS_NO_CONNECTION,        /* Unable to connect */
    DIGS_UNSPECIFIED_SERVER_ERROR,
        /* An unspecified error from server. */
    DIGS_FILE_NOT_FOUND,       /* File not found */
    DIGS_INVALID_CHECKSUM,     /* Invalid checksum */
    DIGS_UNSUPPORTED_CHECKSUM_TYPE,
        /* This type of checksum is not
        supported.*/
    DIGS_FILE_IS_DIR,          /* Expected file but found directory. */
    DIGS_NO_INBOX,             /* This storage element does not have
    an inbox.*/
    DIGS_UNKNOWN_ERROR        /* Operation failed for an
    unknown reason */
} digs_error_code_t;
```

3.2.3 Checksum types

To accommodate different checksum operations that could be used in DiGS in the future a `digs_checksum_type_t` enumeration is defined below. It should be extended as required by introducing other types of checksums.

```
typedef enum
{
    DIGS_MD5_CHECKSUM,          /* Message-Digest algorithm 5 */
    DIGS_CRC_CHECKSUM          /* Cyclic Redundancy Check */
} digs_checksum_type_t;
```

3.2.4 Atomic transactions

There are several operations presented in the interface, described below, which can leave the grid in the inconsistent state, if they are interrupted. Specifically, such operations may leave the contents of an SE out of sync with the information held by the CT (for example, in the file catalogue). This is clearly undesirable and, wherever possible, an operation should be implemented as an *atomic transaction*: that is, an operation that can return in one of two known states for success and failure.

Unfortunately, there are several operations that are not easily implemented as atomic transactions. These are transfer and recursive operations. Treatment of these operations is considered individually:

- **Transfer operations** `digs_startGetTransfer`, `digs_startPutTransfer`, `digs_startThirdPartyTransfer`, `digs_updateTransfer` and `digs_cancelTransfer` should be implemented in such a way that they return appropriate error codes, which describe the state of the system at the time of failure e.g. `DIGS_INVALID_CHECKSUM` – would mean that a file has been transferred over, but has

been corrupted or truncated and therefore should be deleted and the whole operation restarted; `DIGS_FILE_NOT_FOUND` – would mean that there was no file to be copied over or the file wasn't copied, so nothing to be deleted there, and so on. Provided that these failure modes are enumerated completely, then the CT can perform appropriate consistency checks to re-synchronise its status information. Additionally, `digs_updateTransfer` should return a status of the transfer as defined below:

```
typedef enum
{
    DIGS_TRANSFER_IN_PREPARATION, /* Transfer is being prepared */
    DIGS_TRANSFER_PREPARATION_COMPLETE,
                                /* Transfer has been prepared */
    DIGS_TRANSFER_IN_PROGRESS, /* Transfer is progressing */
    DIGS_TRANSFER_DONE,        /* Transfer completed successfully */
    DIGS_TRANSFER_FAILED,      /* Transfer failed */
    DIGS_TRANSFER_CLEANUP      /* Post-transfer cleanup */
} digs_transfer_status_t;
```

- Operation `digs_mv` could be replaced by a call to any of the transfer methods and atomic `digs_rm`.
- The recursive commands could be replaced by a recursive call to their atomic equivalents e.g. `digs_rmdir` or `digs_rm` and `digs_mkdir`, respectively.

3.2.5 Client interface

This interface is very simple, as clients only interact with an SE when downloading files from the data grid or uploading new data to the Inbox folder (more in Section 4) of an SE.

```
/*
 * Starts a get operation (pull) of a file (SURL) from a
 * remote SE (hostname) to local filespace (localPath). A handle is
 * returned to uniquely identify this transfer.
 */
digs_error_code_t digs_startGetTransfer(char *errorMessage, const
char *hostname, const char *SURL, const char *localPath, int
*handle);

/*
 * Copies a file (localPath) on a client to an Inbox folder on SE.
 *
 * This method is effectively a wrapper around
 * digs_startPutTransfer() that firstly resolves the target SURL
 * based on the specific configuration of the SE (identified by its
 * hostname). Not all SEs have Inboxes. If this function is called
 * on an SE that does not have an Inbox, then an appropriate error
 * is returned.
 *
 * A handle is returned to uniquely identify this transfer.
 */
digs_error_code_t digs_copyToInbox(char *errorMessage, const char
*hostname, const char *localPath, const char *lfn, int *handle);
```

```
/*
 * Starts a put operation (push) of a local file to a remote SE,
 * identified by hostname, with a specified remote location (SURL)
 * A handle is returned to uniquely identify this transfer.
 *
 * If a file with the chosen name already exists at the remote
 * location, it will be overwritten without a warning.
 */
digs_error_code_t digs_startPutTransfer(char *errorMessage, const
char *hostname, const char *localPath, const char *SURL, const char
*lfn, int *handle);
```

```

/*
 * Moves a file (identified by remoteFilename) from the Inbox
 * folder to permanent storage on an SE (identified by hostname),
 * with a specified destination file path.
 *
 * The function is effectively a wrapper to digs_mv operation.
 */
digs_error_code_t digs_copyFromInbox(char *errorMessage, const char
*hostname, const char *lfn, const char *targetPath);

/*
 * Checks a progress of a transfer, identified by a handle, by
 * returning its status. Also returns the percentage of transfer
 * that has been completed.
 */
digs_error_code_t digs_monitorTransfer_globus(char *errorMessage,
int handle, digs_transferStatus_t *status, int *percentComplete);

/*
 * Cancels and cleans up a transfer identified by a handle. Removes
 * partially transferred files.
 */
digs_error_code_t digs_cancelTransfer(char *errorMessage, int
handle);

/*
 * Cleans up a transfer identified by a handle. Checks that the file
 * has been transferred correctly (using checksum) and removes the
 * LOCKED postfix if it has. If the transfer has not completed
 * correctly, removes partially transferred files.
 */
digs_error_code_t digs_endTransfer(char *errorMessage, int handle);

```

3.2.6 Control thread interface

This interface exists between the CT and an SE. It includes a more extensive range of functionalities, which are organised into three sets.

Methods responsible for obtaining properties of the files:

```

/*
 * Checks if a file/directory exists on a node.
 */
digs_error_code_t digs_doesExist(char *errorMessage, const char
*filePath, const char *hostname, int *doesExist);

/*
 * Checks if a file on a node is a directory.
 */
digs_error_code_t digs_isDirectory(char *errorMessage, const char
*filePath, const char *hostname, int *isDirectory);

```

```
/*
 * Gets a size in bytes of a file on a node.
 *
 * Returned size is stored in variable fileLength
 */
digs_error_code_t digs_getLength(char *errorMessage, const char
*filePath, const char *hostname, long long int *fileLength);

/*
 * Gets a 16 characters MD5 checksum of a file on a node.
 * Gets a checksum of file on node using the defined checksum type.
 *
 * Hex digits will be returned in uppercase.
 * Returned checksum is stored in variable fileChecksum and its
 * type is stored in checksumType. Note that fileChecksum should be
 * freed accordingly to the way it was allocated in the
 * implementation! Use digs_free_string to free it.
 */
digs_error_code_t digs_getChecksum(char *errorMessage, const char
*filePath, const char *hostname, char **fileChecksum,
digs_checksum_type_t *checksumType);

/*
 * Gets an owner of the file on a node.
 *
 * Returned owner name is stored in variable ownerName. Note that
 * ownerName should be freed according to the way it was allocated
 * in implementation! Use digs_free_string to free it.
 */
digs_error_code_t digs_getOwner(char *errorMessage, const char
*filePath, const char *hostname, char **ownerName);

/*
 * Gets a group of the file on a node.
 *
 * Returned group name is stored in variable groupName. Note that
 * groupName should be freed accordingly to way it was allocated in
 * the implementation!
 */
digs_error_code_t digs_getGroup(char *errorMessage, const char
*filePath, const char *hostname, char **groupName);

/*
 * Gets the permissions of a file on a node.
 *
 * Returned permissions are stored in variable permissions, in a
 * numeric representation. For example 754 would mean:
 *   owner: read, write and execute permissions,
 *   group: read and execute permissions,
 *   others: only read permissions.
 */
digs_error_code_t digs_getPermissions(char *errorMessage, const
char *filePath, const char *hostname, int *permissions);

/*
 * Returns the modification time of a file located on a node.
 */
digs_error_code_t digs_getModificationTime(char *errorMessage,
const char *filePath, const char *hostname, time_t
*modificationTime);
```

```

/*
 * Free the string according to the correct implementation.
 */
digs_error_code_t digs_free_string(char **string);

/*
 * Free an array of strings according to correct implementation.
 */
digs_error_code_t digs_free_string_array(char ***arrayOfStrings);

```

Modifying the properties of files/file transfers:

```

/*
 * Starts a get operation (pull) of a file (SURL) from a
 * remote hostname to local destination file path. A handle is
 * returned to uniquely identify this transfer. The file will have
 * postfix LOCKED until endTransfer has been called.
 *
 */
digs_error_code_t digs_startGetTransfer(char *errorMessage, const
char *hostname, const char *SURL, const char *localPath, int
*handle);

/*
 * Starts a put operation (push) of a local file to a
 * remote hostname with a specified remote location (SURL)
 * A handle is returned to uniquely identify this transfer.
 *
 * If a file with the chosen name already exists at the remote
 * location, it will be overwritten without a warning. The file
 * will have the postfix LOCKED until endTransfer has been called.
 */
digs_error_code_t digs_startPutTransfer(char *errorMessage, const
char *hostname, const char *localPath, const char *SURL, int
*handle);

/*
 * Copies a source file from source node to a destination node with
 * specified destination file path and name. Returns handle to this
 * transfer. To be left unimplemented for now
 *
 */
digs_error_code_t digs_startThirdPartyTransfer(char *errorMessage,
const char *filePathFrom, const char *nodeFrom, const char
*filePathTo, const char *nodeTo, int *handle);

/*
 * Checks a progress of a transfer, identified by a handle, by
 * returning its status. Also returns the percentage of transfer
 * that has been completed.
 */

digs_error_code_t digs_monitorTransfer_globus(char *errorMessage,
int handle, digs_transferStatus_t *status, int *percentComplete);

```

```
/*
 * Cancels and cleans up a transfer identified by a handle. Removes
 * partially transferred files.
 */
digs_error_code_t digs_cancelTransfer(char *errorMessage, int
handle);

/*
 * Cleans up a transfer identified by a handle. Checks that the file
 * has been transferred correctly (using checksum) and removes the
 * LOCKED postfix if it has. If the transfer has not completed
 * correctly, removes partially transferred files.
 */
digs_error_code_t digs_endTransfer(char *errorMessage, int handle);

/*
 * Moves/Renames a file on an SE (hostname).
 */
digs_error_code_t digs_mv(char *errorMessage, const char *hostname,
\
const char *filePathFrom, const char *filePathTo);

/*
 * Deletes a file from a node.
 */
digs_error_code_t digs_rm(char *errorMessage, const char *hostname,
const char *filePath);

/*
 * Deletes a directory from a node.
 */
digs_error_code_t digs_rmdir(char *errorMessage, const char
*hostname, const char *filePath);

/*
 * Deletes a directory with subdirectories (recursively) from a
 * node.
 */
digs_error_code_t digs_rmr(char *errorMessage, const char
*hostname, const char *filePath);

/*
 * Creates a directory on a node.
 * Will return an error if the directory already exists.
 */
digs_error_code_t digs_mkdir(char *errorMessage, \
const char *hostname, const char *filePath);
```

```

/*
 * Creates a tree of directories at one go on a node
 * (e.g. 'mkdir lnf:/ukqcd/DWF')
 *
 */
digs_error_code_t digs_mkdirtree(char *errorMessage, const char
*hostname, const char *filePath);

/*
 * Sets up/changes a group of a file/directory on a node.
 * Globus implementation relies on chgrp command existing on the
 * node.
 *
 */
digs_error_code_t digs_setGroup(char *errorMessage, const char
*filePath, const char * hostname, const char *group);

/*
 * Sets up/changes permissions of a file/directory on a node.
 *
 */
digs_error_code_t digs_setPermissions(char *errorMessage, const
char *filePath, const char *hostname, const char *permissions);

```

Other:

```

/*
 * Pings node and checks that it has a data directory.
 *
 */
digs_error_code_t digs_ping(char *errorMessage, const char *
hostname);

/*
 * Gets a list of all the DiGS file locations. Recursive directory
 * listing of all the DiGS data files
 * Setting the allFiles flag to zero will hide any temporary
 * (locked) files.
 *
 * Returned an array of the file paths is stored in variable list.
 * Note that list should be freed accordingly to the way it was
 * allocated in
 * the implementation! Use digs_free_string_array.
 */
digs_error_code_t digs_scanNode(char *errorMessage, const char
*node, char ***list, int *listLength, int allFiles);

/* digs_error_code_t (*digs_scanInbox)(char *errorMessage,
 * const char *hostname, char ***list, int *listLength, int
 * allFiles);
 *
 * Gets a list of all the DiGS file locations in the inbox.
 * Recursive directory listing of all the files under inbox.
 * Setting the allFiles flag to zero will hide any temporary
 * (locked) files.
 *
 * Returned an array of the file paths is stored in variable list.
 * Note that list should be freed accordingly to the way it was
 * allocated in the implementation! Use digs_free_string_array.

```

```
digs_error_code_t diggs_scanInbox(char *errorMessage,
const char *hostname, char ***list, int *listLength, int allFiles);
```

3.3 Implementation of the Interface for Globus SE

Some of the methods have been already implemented and can be found in source file `gridftp.c`. This section is divided into parts depending on the Globus tool used to implement the methods.

Methods that have GridFTP equivalents:

```
digs_doesExist()
- int doesItExist(char *host, char *filename)
- globus_ftp_client_exists

digs_isDirectory()
- globus_ftp_client_mlst ? (Type=file Type=dir Type=(c|p)dir
Type=link)

digs_getLength()
- long long getRemoteFileLengthFullPath(char *host, char *filename)
- globus_ftp_client_size
-globus_ftp_client_mlst - (Size=1024990; for files)

digs_getChecksum()
- globus_ftp_client_cksm -- good as it only supports MD5

digs_getPermissions()
- globus_ftp_client_mlst - (Perm=r -- readable by current user;
Perm=el - dir can be entered and listed;)
-- maybe difficult
- globus_ftp_client_list ? -- depends on the LIST output

digs_getOwner()
- globus_ftp_client_list ? -- depends on the LIST output

digs_getGroup()
- globus_ftp_client_list ? -- depends on the LIST output

digs_cp()
- int thirdPartyTransfer(char *sourceHost, char *sourceFile, char
*destHost, char *destfile)
- int copyToLocal(char *remoteHost, char *remoteFile, char
*localFile)
- int copyFromLocal(char *localFile, char *remoteHost, char
*remoteFile)

digs_mv()
- int renameRemoteFileFullPath(char *host, char *oldName, char
*newName)
- globus_ftp_client_move

digs_rm()
- int deleteRemoteFileFullPath(char *host, char *filename)
- globus_ftp_client_delete

digs_rmdir()
- globus_ftp_client_rmdir

digs_rmr()
- globus_ftp_client_rmdir
```

```
digs_mkdir()  
- int makeDirectory()  
- globus_ftp_client_mkdir  
  
digs_setPermissions()  
- globus_ftp_client_chmod  
  
digs_scanNode()  
- globus_ftp_client_list - maybe difficult
```

Remaining methods (which might need to use GRAM):

```
digs_getFileAccessTime()  
  
digs_mkdirtree()  
  
digs_chgrp()  
  
digs_touch()  
  
digs_getFreeSpace()
```

3.4 Implementation of the Interface for SRM

3.4.1 General Implementation and Dependencies

The SRM support will be implemented as an adaptor conforming to the DiGS Storage Element Interface, as is the current Globus SE support. It will be written in C and will use gSOAP with the gsoap-gsi plugin to access SRM services. These libraries can be statically linked to the adaptor so there will be no additional library dependencies except when compiling from source.

Access-control management for an SRM server typically requires the extended attribute set of a VOMS proxy certificate. Because of this, client systems need to provide a `voms-proxy-init` command or equivalent. In addition, DiGS Client Tools that manage proxy generation need to be updated to produce VOMS-style proxies. A VOMS-style proxy should be backward compatible with a GSI-style proxy. Therefore, components that query/access GSI-generated proxies should continue to work without modification.

3.4.2 Sharing of GridFTP Code

SRM uses gsiftp as its default data transfer protocol. gsiftp is already used in the Globus SE adaptor so there is potential for sharing code here. There are 3 options:

1. No code sharing. Duplicate required gsiftp code in the new SRM adaptor
2. Call the existing Globus adaptor methods from the SRM adaptor
3. Move common gsiftp code into a separate module which can be called from both adaptors

Option 1 would make the code less maintainable and should be avoided. Option 2 is also undesirable; the existing Globus adaptor does not expose an interface to gsiftp at an appropriate level, and even if this could be made to work it could easily be broken by any changes to the Globus adaptor. Option 3 would be preferable and should be feasible.

To create the common module, the following internal functions will be moved from the Globus adaptor (`gridftp.c`), as writing in DiGS Version 3.0, into a new module that so that they are available to both adaptors:

```
acquireTransactionListMutex
releaseTransactionListMutex
newFtpTransaction
destroyFtpTransaction
getErrorAndMessageFromGlobus
waitOnFtp
waitOnFtpLong
isTransactionDone
isTransactionDoneLastCheck
completeCallback
replicateCompleteCallback
dataCallback
startFtpWrite
startFtpWriteBlock
startFtpRead
startFtpReadBlock
startFtpReadToBuffer
```

Global variables relating to the transaction list and its mutex will also be moved.

(Note: some of these functions are not required by the SRM adaptor, but they are being moved for consistency as they are very similar to functions that do need to be moved).

Apart from moving the functions and importing them from the new module, no other changes to the existing Globus adaptor will be required, minimising the risk of regressions.

3.4.3 Checksum Support

The DiGS SE interface includes a method for obtaining a checksum of a file (so far this is always an MD5 checksum – the interface supports other checksum types but the calling code may not). This is not easy to implement on SRM. Although the "detailed list" operation may optionally return a checksum for each file, many SRM servers (including DPM and StoRM) do not implement this.

This could be worked around by requesting the transfer URL for a file and then using the `gridftp` method to obtain its MD5 checksum, or by copying the file to the local system and checksumming it there. (The first workaround is preferable as it is much more efficient but the second would be a possibility if in future a different data transfer protocol was to be used). However both of these methods could be problematic if the data is stored on tape, which is not well suited to random access patterns such as the DiGS control thread periodically checking files.

The checksum method is currently used for 3 different purposes:

1. Periodically checking the integrity of files on the grid
2. Obtaining the checksum of each copy of the file in the `digs-check-lfn` tool
3. Obtaining checksums for files without a checksum attribute when the replica catalogue is rebuilt

The first of these (control thread's periodic check) has proven useful in the past for detecting corrupted data so it should be maintained for SRM storage if possible. It is proposed that a "last checksummed" time attribute be added to the replica catalogue for each copy of each file and kept up to date when this check is performed. This would provide the information needed to implement a more sophisticated checking algorithm if it proves necessary in future (for example, checking files on tape-based SEs less frequently). In addition, a per-node flag will

be added to `mainnodelist.conf` allowing the periodic integrity checks to be disabled for a particular storage element, in case they become problematic on some systems.

The second and third uses (`digs-check-lfn` and catalogue rebuild) only happen comparatively rarely so using the workaround here is unlikely to pose a problem. However it is proposed that the catalogue rebuild algorithm, which only requires a checksum from one copy of a file, be updated to take the checksum from a replica on a GridFTP storage element rather than an SRM storage element when possible, minimising the need to checksum files on tape.

In addition to the periodic checksums, files in DiGS are integrity checked whenever they are transferred. This presents no problems for SRM-based systems; when a file is transferred the GridFTP transfer URL is already available so it is easy to call the GridFTP checksum method on it. However it may be more difficult if other transfer protocols are used in the future.

3.4.4 Inbox Support

Inbox support is an optional feature which storage elements do not have to implement. However there must be at least one inbox available on the grid, so it would be advantageous to implement this feature in the SRM adaptor if possible, potentially allowing a DiGS grid to be run with only SRM storage elements.

It appears that the SRM security mechanisms will be sufficient for implementing the DiGS inbox functionality.

The Inbox directory needs to:

- be owned by the control thread certificate
- be group writable (by any groups that can add data) and group readable
- NOT be world-writable or readable

This could either be set up manually by a system administrator, or via a DiGS tool.

3.4.5 Implementation of Data Transfer Methods

The DiGS SE interface data transfer methods are used as follows:

- call `digs_startPutTransfer`, `digs_startGetTransfer`, or `digs_startCopyToInbox` to initiate a transfer. These methods return a handle which can then be passed to the other methods to refer to the transfer
- call `digs_monitorTransfer` periodically to check on the transfer's progress until it completes
- call `digs_endTransfer` to clean up resources after a transfer has completed
- `digs_cancelTransfer` can be used to abort a transfer that is in progress

The SRM data transfer methods work like this:

- call `srmPrepareToPut` or `srmPrepareToGet`. These methods return a request token
- call `srmStatusOfPutRequest` or `srmStatusOfGetRequest` periodically, passing in the token. Once the preparation is complete, these methods will return a transfer URL
- transfer the file to or from the transfer URL using `gridftp`
- call `srmPutDone` to notify the server that the upload is complete (put transfer only)
- `srmAbortRequest` is used to cancel a request that is in progress

The following mapping is proposed:

- `digs_startPutTransfer` and `digs_startCopyToInbox` should internally call `srmPrepareToPut`. The token returned by SRM will be associated with the DiGS handle
- similarly, `digs_startGetTransfer` should call `srmPrepareToGet` and associate the SRM token with the DiGS handle
- `digs_monitorTransfer` will behave differently depending on whether the actual transfer has started.
 - if the transfer has not yet started, it will call `srmStatusOfPutRequest` or `srmStatusOfGetRequest` to check if the transfer URL is available yet. If it is, the actual gridftp transfer will be started
 - if the transfer has started, it will monitor the gridftp transfer progress similarly to the Globus adaptor
- `digs_endTransfer` will perform any necessary internal clean up, and for put requests will call `srmPutDone`
- `digs_cancelTransfer` will stop the gridftp operation and/or call `srmAbortRequest` (depending on what stage the transfer is at)

The DiGS SRM adaptor will need to be able to keep track of information including the SRM token, file name, whether the actual transfer has started, and whether it is a put or get transfer, and associate this with the transfer handle.

Although gsiftp is the default transfer protocol for SRM and the only one that needs to be supported at present, it would be wise to restrict any gsiftp-specific code to a well-defined set of functions, to make it as easy as possible to add alternative transfer protocols should they ever be required. This set should include functions for starting, monitoring and cancelling transfers. It should also include a checksum function as this must be handled by the transfer protocol due to lack of support in the SRM service itself.

3.4.6 Tape-Storage Considerations

SRM provides a uniform interface to both tape and disk storage systems. The main difference between them is that for files stored on tape, there is likely to be a much longer delay between calling `srmPrepareToGet` and the gsiftp transfer URL becoming available. This will not cause any problems for the DiGS SRM adaptor itself, but the calling code should be aware of transfers from tape storage nodes potentially taking a very long time.

It should be possible to accommodate this with the existing time-out mechanism. All parts of the DiGS code that transfer files either have no time-out (for example, replications) or use the node-specific file transfer time-out defined in the main node list file (for example, client gets and puts). This time out value could be increased as necessary for any nodes with tape storage.

3.4.7 Implementation of Other Methods

Most of the general DiGS SE interface methods can be mapped onto SRM methods in a straightforward manner:

<code>digs_doesExist</code>	- <code>srmLs</code>
<code>digs_isDirectory</code>	- <code>srmLs, detailed, check file type</code>
<code>digs_getOwner</code>	- <code>srmLs, detailed</code>

digs_getGroup	- srmLs, detailed
digs_getPermissions	- srmLs, detailed
digs_getModificationTime	- srmLs, detailed
digs_getLength	- srmLs
digs_getChecksum	- Unimplemented for tape. Get transfer URL and call gridftp checksum function for disk.
digs_setGroup	- srmSetPermission
digs_setPermissions	- srmSetPermission
digs_mkdir	- srmMkdir
digs_mkdirtree	- srmMkdir and srmLs
digs_rmdir	- srmRmdir
digs_rmr	- srmRmdir, recursive
digs_mv	- srmMv
digs_copyFromInbox	- srmCopy
digs_ping	- srmPing
digs_housekeeping	- srmLs and srmRm, to remove any old files from the inbox
digs_rm	- srmRm
digs_scanNode	- srmLs, recursive
digs_scanInbox	- srmLs

3.4.8 Security Implementation

The existing Globus SE adaptor uses standard UNIX-style file-system permissions, manipulated by GridFTP functions, to implement the DiGS security model. This works as follows:

- All files stored on the SE are owned by the same user (the DiGS Package Account), to which the control thread's certificate is mapped in the grid-mapfile
- The file's group is set to whichever group owns the file (at present, only one privileged group is used). The DiGS Package Account must be a member of all the groups in use.
- All files are readable and writable by their owner, and readable by their group
- Public files are also world readable. Private files are not
- `digs_setGroup` and `digs_setPermissions` are used to manage permissions on the files

The inbox has its own security requirements:

- It must be writable by members of all groups that can write to the grid (so called Privileged Groups).
- It must be owned by the DiGS Package Account, so that the Control Thread can delete files from there.
- It (or the files in it) should not be world-readable, as this would potentially allow unauthorised access to private files.
- Files in the Inbox must however be group-readable so that the Control Thread can read them.

Most of the Inbox permissions are set by the system administrator as part of storage element setup and are not modified programmatically by DiGS.

SRM's group and permission system appears to be a superset of that used in the Globus SE adaptor, so it should be possible to implement this security model on SRM with no problems. In SRM (DPM at least), file owners are defined directly by their certificate subject and not by

a UNIX username, and groups are defined at the virtual-organisation level. It is possible, in SRM, to set access permissions on a file for multiple users and groups, although this is not necessary for the DiGS Version 3 model.

At present, a DiGS user can specify their group by setting an environment variable, or alternatively the control thread will find their default group from the `group-mapfile`. However, it would be preferable for the control thread to extract the user's group from their VOMS proxy where possible, only falling back to the environment variable or default group when the VOMS information is not present (e.g. a non-VOMS proxy is being used).

3.4.9 SRM-specific Node Properties

The DiGS `mainnodelist.conf` file provides a primitive way of storing properties for each storage element. For the SRM adaptor some additional properties will be required:

- the SRM endpoint URL
- the base URL of the storage managed by DiGS on this SE

At present, the supported property types are hard-coded in `node.c`, and this would need to be done for any properties added, even if they are only relevant for one type of SE. To avoid the need to put SE-specific code in `node.c`, it is proposed that a hashtable structure be added to the storage-element data structure. Any extra properties for the node will then be added to the hashtable by `node.c` when it parses the main node list, and the SE-specific code can query the hashtable to find any properties it requires.

Additionally, it may be desirable to move the main node list to an XML-based format. This would allow for more robust parsing and error checking, better extensibility and the possibility of more structured node information, but at the expense of adding some complexity to the code and adding a dependency on `libxml2` (a standard library for working with XML in C/C++) when compiling DiGS from source code.

3.4.10 Testing Strategy

A very comprehensive test suite already exists for the Globus SE adaptor, built on the CuTest framework. It appears to be mostly generic enough to work on any SE adaptor that conforms to the interface, however it contains several hard-coded paths and makes some assumptions that may not be true for SRM storage elements. This suite should be updated to be able to test the SRM adaptor (and any future adaptors) as well as the Globus adaptor.

There is also a script for testing DiGS installations, `digs-test-setup`. This currently only tests the local machine on which it is run, and the control node. It does not access the storage elements and so will not need to be updated for SRM.

4 Changes to setup of Inbox folder

In versions of DiGS, up to and including Version 2.0, an Inbox folder must exist on each SE. It is used as a temporary store for files that are going to be inserted into the grid. Users, who have write access to the grid, can copy files into this folder, implicitly following the sequence below:

1. A client copies a file into the Inbox directory on an SE. The SE is chosen according to the `nodeprefs.conf` file, which contains the list of SEs ordered by preference. The location of the Inbox is expected to be `$DIGS_HOME/data/NEW`.
2. Client sends a message to CT, advising it that there is a file awaiting insertion. This message contains basic file information, including file name, SE's name and file attributes.
3. CT – having received the signal – stores the information about the file and scans the suggested SE's `NEW` folder.
4. If CT finds a file that has the corresponding attributes, it copies the file to the permanent grid storage (by invoking GRAM call to `cp`).
5. If this operation succeeds, CT registers the file's new location and its attributes with RLS and deletes the temporary file from `NEW`.

This approach has some disadvantages.

- Firstly, it implies a requirement on (potentially third-part) SE providers. It requires them to set up an appropriate folder with the exact security restrictions enforced. If they failed to do so, confidentiality of newly added data could be compromised.
- Secondly, only one group can write to `NEW` directory in current implementation. This inhibits the use of multiple groups in DiGS.
- Thirdly, there shouldn't be any folders like `NEW` on tape-based SEs, as write times are likely to be prohibitively slow.

To overcome these disadvantages, several changes to the Inbox mechanism are to be implemented for DiGS version 3, as follows:

- Firstly, the availability of an Inbox on a SE is optional. An attribute will be added to the `mainodelist.conf` configuration file, which indicates the presence/absence of an Inbox on each SE.
- Greater flexibility is provided with regard to the choice of location for the Inbox. The full path to the Inbox is specified in the `mainodelist.conf` configuration file. In versions of DiGS up to and including Version 2.0, the Inbox was assumed to exist within `$DIGS_HOME/data/NEW`. The change described here removes this restriction.

The attribute specified in the previous item will likely be the location of the Inbox, with an empty field indicating that no Inbox is present.

At least one Inbox must be present on a data grid. If no Inbox is specified in the `mainodelist.conf` file, then the Control Thread will generate an error and halt.

- As described in Appendix B, the client suggests the group to which new data should be assigned, and the CT validates the suggestion. This allows multiple user groups to submit files to the data grid.
- A CT management operation is to be implemented that periodically checks the contents of the Inbox on each SE and removes any files that are older than a preconfigured time – likely to be ~1 week. This operation is added to help ensure that an Inbox does not

become 'full' in the event of a problem being encountered with a particular storage element or otherwise.

The access permissions of the each Inbox folder need to be configured appropriately, in order to enable different groups to write to the same Inbox. This should be done such that a new file cannot be read by members of groups other than that of the original submitter. For example, the NEW folder could allow read, write and execute permissions for everyone (drwxrwxrwx), but each new file would be readable and writable only by the user who submitted it, and readable by his or her group (rw-r-----).

5 Special Requirements for Storage Element Providers

5.1 Security Concerns

As a DiGS-powered grid supports a group-based authorisation model (see [17] for more information), appropriate access control needs to be set up on each SE. In order to achieve this, files should be protected using their UNIX (or other SE-specific mechanism) file permissions, to fulfil the following requirements:

- A (Unix) group needs to be created for each group that has access to the data grid.
 - Two types of groups are envisaged:
 - *Experiment group* – users from such a group are allowed to submit files to the grid, as well as retrieve them.
 - *Guest group* – users from such a group are only allowed to retrieve files that have been specifically made world-readable. Users from guest groups may not submit files to the grid.
- All files should be owned by a special user (typically, the `qcdgrid` user) and belong to one of the experiment groups.
- Only members of experiment groups should have permission to write to the NEW folder.
- Users of an experiment group should be able to read files belonging to their own group, as well as files from other groups that have been classified as world-readable.
- Users of an experiment group should be allowed to list the content of any data storage directory, except for the NEW folder.
- Members of an experiment group should not be able to read a file belonging to a different experiment group, unless the file is classed as world-readable.
- Users belonging to guest groups should have not have write permissions to anything, by default.

When a new file is moved onto an SE, it should be configured with appropriate permissions. It should be:

- owned by the special user.
- readable by one of the experiment groups (same group as the user who submitted the file).
- Not available to anyone from outside the aforementioned group.

5.2 Which User Management System to choose

At the time of writing, the information about UKQCD/ILDG users is hosted on a VOMS² system. On each SE, this information is downloaded and processed into a GSI³ *grid-mapfile*, by a small component of DiGS software. This approach has two disadvantages:

- Firstly, any changes to the list of *special entries* in the *grid-mapfile* – for example, the host certificate of the Control Node – require modifications to be made manually on every SE. This is because special entries are not recorded alongside VO data (in VOMS).

² Virtual Organisation Management System.

³ Grid Security Infrastructure.

- Secondly, if there is a SE that doesn't allow any software to be installed, this approach cannot be used.

It is therefore recommended to use the Grid User Management System (GUMS) [8] as the user management system alongside DiGS. GUMS is a web service that contains configurable rules of identity mappings. It can be set up to regularly download users data from a VOMS system, and can also be used to produce grid-mapfile access control lists if required.

As it is a centralised service, it is the only place where any modifications have to be made. Those changes will then propagate to all SEs without further administrator attention. Additionally, GUMS can be integrated with dCache (through gPlazma module) in such a way that there is no need for either a grid-mapfile or any grid-mapfile generation software to be installed on the SE.

6 Conclusions

Having taken into account the future data management requirements of UKQCD, we have reconsidered the existing setup of Storage Elements. The investigation presented in this document has identified several compatibility issues between DiGS Version 2.0 and future UKQCD storage requires. These incompatibilities have been addressed by a high-level design, which also promises improved robustness for and lower complexity administration of DiGS-powered infrastructures.

7 References

- [1] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell’Agnello, Á. Frohner, A. Gianoli, K. Lörentey, and F. Spataro, VOMS, an Authorization System for Virtual Organizations, Technical Report from INFN, Italy.
- [2] A. Anthony, *Adding SRM Functionality to QCDGrid*. MSc Dissertation, University of Edinburgh (2007).
- [3] Apache Project, *Axis SOAP Environment*. Project homepage at <http://jakarta.apache.org/tomcat>.
- [4] Apache Project, *Tomcat Web Application Container*. Project homepage at <http://jakarta.apache.org/tomcat>.
- [5] Apache Project, *XIncid XML Database*. Project homepage at <http://xml.apache.org/xindice/>.
- [6] D. Baker, *The UK Grid Integration Test Script – GITS*. Project homepage at <http://www.soton.ac.uk/~djb1/gits.html>.
- [7] M.G. Beckett and J.T. Perry, *D1.4.2: QCDgrid Software Review 2*, Project deliverable (September 2006).
- [8] Brookhaven National Laboratories (BNL), *GUMS Overview*, Online documentation available from <https://www.racf.bnl.gov/Facility/GUMS/1.2/index.html> (November 2007).
- [9] D.J. Byrne, *QCDGrid2 Software Review 1*. QCDgrid project deliverable (March 2005).
- [10] EGEE, *Enabling Grid for E-Science*. Project homepage at <http://public.eu-egee.org/> (2006).
- [11] EGEE, *gLite – Lightweight Middleware for Grid Computing*. Project homepage at <http://glite.web.cern.ch/glite/> (2006).
- [12] eXist, *eXist – An Open Source Native XML Database*. Project homepage at <http://exist-db.org/> (2006).
- [13] Globus Alliance, *Globus Toolkit*. Project homepage at <http://www.globus.org/> (2006).
- [14] Globus Alliance, *Overview of the Grid Security Infrastructure* <http://www.globus.org/security/overview.html>.
- [15] International Lattice Datagrid, *ILDG Virtual Organisation*. Service hosted at <https://grid-voms.desy.de:8443/voms/ildg/> (2006).
- [16] International Lattice Datagrid. Project homepage at <http://www.lqcd.org/ildg/tiki-index.php>.
- [17] R.H. Ostrowski and M.G. Beckett, *WP4.3 ILDG File Catalogue Security*, QCDgrid project deliverable (February 2007).
- [18] QCDgrid Project, *QCD Software*, hosted by NeSCForge at <http://forge.nesc.ac.uk/projects/qcdgrid/> (January 2008),
- [19] UKQCD Collaboration. Homepage at <http://www.ph.ed.ac.uk/ukqcd/> (2006).
- [20] UKQCD Collaboration, *QCDgrid Project*. Homepage at <http://www.gridpp.ac.uk/qcdgrid>.
- [21] SRM Working Group, <http://sdm.lbl.gov/srm-wg/> (2006).
- [22] dCache. Homepage at <http://www.dcache.org/>