



IST-2001-32133

GridLab - A Grid Application Toolkit and Testbed

## Deliverable D8.4 - Low level data access services

---

Author(s):	Andrei Hu tanu, Andre Merzky, Brygg Ullmer
Document Filename:	
Work package:	WP 8 - Data Handling and Visualization
Partner(s):	
Lead Partner:	ZIB
Config ID:	GridLab-08-DATA-0004-M18
Document classification:	Internal

---

**Abstract:** This document describes the APIs for the low level data access services and libraries, with prototypes deployed on the GridLab testbed.



## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Streaming library</b>	<b>2</b>
2.1	Stream producer API . . . . .	2
2.1.1	Constructor . . . . .	2
2.1.2	WriteBuffer . . . . .	2
2.1.3	ProbeWrite . . . . .	3
2.2	RequestBuffer . . . . .	3
2.3	CommitBuffer . . . . .	3
2.4	Stream consumer API . . . . .	4
2.4.1	Constructor . . . . .	4
2.4.2	Read . . . . .	4
2.4.3	Probe . . . . .	4
2.4.4	Skip . . . . .	4
2.5	Deployment . . . . .	5
<b>3</b>	<b>File browsing service</b>	<b>5</b>
3.1	File browsing API . . . . .	5
3.1.1	ConnectedList . . . . .	5
3.1.2	StopList . . . . .	5
3.1.3	non-chached listing . . . . .	5
3.1.4	Ping . . . . .	6
<b>4</b>	<b>Improvements to previously delivered software</b>	<b>6</b>
<b>5</b>	<b>Summary</b>	<b>6</b>

## 1 Introduction

This document describes the API of the GridLab low level data access libraries, in association with their first prototype release (streaming library), and the remote file browsing service. The data movement service, initially planned to be released together with this document was already delivered and documented in the previous report.

## 2 Streaming library

The streaming library provides means to create and use reliable simplex data streams. The central element of our proposed streaming scheme builds upon the GridFTP protocol and the GridFTP server side data processing feature.

The server side processing feature allows specification of *custom* operations on remote data. These operations are performed by corresponding plugins to the GridFTP server. We use this feature to implement the streaming server which is the GridFTP server which communicates using shared memory with the stream producer(s).

This gives the following limitation: to be able to create a data stream, one needs to have the GridFTP server with the GridLab streaming plugin running on a known port on the local machine.

For the first prototype we provide a C++ API for both the stream producer and the stream consumer.

### 2.1 Stream producer API

#### 2.1.1 Constructor

```
StreamProducer::StreamProducer(int const zone_size, int const port,  
mode_t const perms, char const* filename);
```

This creates a new instance of the `stream` class. The stream will be mapped to a local file to ease the implementation of a security policy. `zone_size` is the size of the shared memory zone that will be used to communicate with the GridFTP server. This value will limit the maximum number of bytes that can be written to the stream using a single write call. `port` is the port where the GridFTP server is listening on the local machine. `perms` are the access permissions to the newly created stream, which are usual file permissions. (values similar to the ones given to `creat()`) `filename` is the name of the file that will be created with the given permissions, name that needs to be referred by the streaming clients that access the stream.

#### 2.1.2 WriteBuffer

```
int StreamProducer::WriteBuffer(void const * data, int const size,  
unsigned char last_chunk);
```

Writes `size` bytes from the `data` pointer to the stream. First, waits for that amount of memory to be free. This might mean waiting until previous data was read by a stream consumer. `last_chunk` signalizes that this call will be the last write call so the stream should be closed after writing the bytes.

Returns : 0 = Success, -1 = Failure.

### 2.1.3 ProbeWrite

```
globus_bool_t StreamProducer::ProbeWrite(int const size) const;
```

Verifies if `size` bytes can be immediately written to the stream (no waiting for a consumer). Returns : boolean, 1 = true, the bytes can be written immediately.

## 2.2 RequestBuffer

```
void* StreamProducer::RequestBuffer(int& size, int& id,  
    unsigned char last_chunk);
```

This method requests a contiguous memory zone of maximum `size` bytes that will be used to write “directly” into the stream. This method does not involve the memory copy between the stream producer and the GridFTP server used in the `WriteBuffer` call and is offered as a possibility to optimize the transfer. If at that particular moment a contiguous buffer of `size` bytes cannot be provided (end of shared memory zone), the maximum available buffer will be returned and `size` will contain its length. `id` is a number that is returned and that identifies the request and this needs to be used in the `CommitBuffer` call to free the requested (and locked) buffer. `last_chunk` - same as in `WriteBuffer`.

Returns : the allocated buffer if success, NULL at failure

```
void* StreamProducer::RequestVarBuffer(int& size, int& id,  
    unsigned char last_chunk);
```

This method adds new functionality to the `RequestBuffer` method in the sense that in the requested buffer, a different (lower) number of bytes than the requested `size` can be actually written. The number of written bytes needs to be specified in the `CommitBuffer` method. We call the returned buffer a *variable size buffer*. When a variable size transfer is in place (a variable size buffer was requested), all the following requests will have to wait for this transfer to end (the buffer to be committed) before they can be served.

Returns : the allocated buffer if success, NULL at failure.

## 2.3 CommitBuffer

```
int StreamProducer::CommitBuffer(int const id)
```

This method releases the buffer with the identifier `id` and copies the data to the stream. If the request was a variable size buffer request this will commit the exact number of bytes that were requested. The commit will wait if necessary for the buffer requested before `id` to be committed. Returns : 0 = Success, -1 = Failure

```
int StreamProducer::CommitBuffer(int const id, int const size,  
    unsigned char const last_chunk);
```

This method is useful when used in conjunction with the `RequestVarBuffer` method, to release the requested variable size buffer(s). The buffer identifier is `id`, the actual number of bytes written in the requested buffer and the number of bytes that will be submitted is specified as `size`. If this is the last write, this can be specified by setting `last_chunk` to 1.

Returns : 0 = Success, -1 = Failure

## 2.4 Stream consumer API

### 2.4.1 Constructor

```
StreamConsumer::StreamConsumer(char const* url);
```

This initialises the class for future operations with the data stream. The url is of the form 'gsiftp://<host>:<port>/<path\_to\_file\_and\_name>'. The host and the port determine the GridFTP server that was used to create the stream and the filename is the one specified in the constructor of the stream producer.

### 2.4.2 Read

There are two modes of reading. One mode is to read only portions from the data stream, and the second mode is to read the whole data from the stream. In the second mode one needs to make an extra call to issue the read command.

```
int StreamConsumer::start_read_all();
```

This call issues the command to the GridFTP server that he should only serve this connection for reading from the stream.

Returns : 0 = Success, -1 = Failure

```
int StreamConsumer::read(globus_byte_t* buffer, globus_size_t& size);
```

This call reads at most **size** bytes from the stream and puts the result in **buffer**. The call is blocking and the call will wait until the requested number of bytes are available or the stream is closed. If the stream is closed from the remote side (and no more bytes will be produced) and the number of available bytes is smaller than the requested size, only the available bytes will be read and their number is returned in **size**.

The call can be used after a **start\_read\_all** call, to read some more bytes from the opened stream or alone to read just a portion from the stream.

Returns : 0 = Success, -1 = Failure

### 2.4.3 Probe

```
int StreamConsumer::probe(globus_size_t const size, globus_bool_t& response);
```

This call tests to see if **size** bytes are available to be read from the stream immediately. The result is written to **response**. The call cannot be used after a **start\_read\_all**.

Returns : 0 = Success, -1 = Failure

### 2.4.4 Skip

```
int StreamConsumer::skip(globus_size_t const size);
```

This call dumps **size** bytes from the opened stream. It cannot be used in conjunction with **start\_read\_all**.

Returns : 0 = Success, -1 = Failure

## 2.5 Deployment

To be able to use the streaming producer, one needs to install the GridFTP server with server-side processing compiled together with the plugin provided by us. We will also provide the libraries implementing the presented API.

## 3 File browsing service

This is a SOAP based Web Service implemented using the Globus FTP library. The service is running on the testbed.

### 3.1 File browsing API

#### 3.1.1 ConnectedList

```
int ns__DATAConnectedList(char* in_URL, int verbose,
    struct List_response *resp);
struct List_response {
    char *retlist;
    char *response;
};
```

This is the main function of the browsing service. As parameters it takes the URL of the directory to be listed, a boolean parameter `verbose`, that when set to 1 will result in detailed listing of the given directory (`ls -l`). When set to 0, the method will return just a list of entries in that directory. The response has two components, first, the string containing the directory listing, and second a string containing a human-readable error code.

When called for the first time, this method initiates a cached connection to the GridFTP server. That connection will be used for all the following list operations until the `StopList` method is called. Also, the service will not cache more than a fixed number of connections (at this moment 100 connections) at one time.

#### 3.1.2 StopList

```
int ns__DATAStopList(void* dump, char** response);
```

This method deletes the client handle, cached for the user connecting to the service. This removes all cached connections for that user.

The first parameter is ignored, the second is filled with a human-readable string that contains the return message.

#### 3.1.3 non-cached listing

```
int ns__DATAListDefaults(char* in_URL,
    struct List_response *response);
```

This method returns the list of entries in the specified directory. The connection to the GridFTP server is not cached.

```
int ns__DATAListVerboseDefaults(char* in_URL,
    struct List_response *response);
```

This method returns the detailed list (ls -l) of entries in the specified directory. The connection to the GridFTP server is not cached.

```
int ns__DATAList(char* in_URL, int max_retries, long interval_sec,  
long deadline_sec, int verbose,  
struct List_response *response);
```

This method offers the possibility to select between detailed and non-detailed listing (using verbose) as well as the possibility to activate the globus ftp client restart plugin (using the max\_retries, interval\_sec and deadline\_sec parameters)

#### 3.1.4 Ping

```
int ns__getServiceDescription(void* dump, char** description);
```

Returns a human-readable string with the description of the service.

## 4 Improvements to previously delivered software

The replica management system and the data movement service now use predictions given by the adaptive service to select the best available replica and to tune the gridftp transfer.

## 5 Summary

This deliverable is preliminary, and describes work in progress.