GT 4.2.0 XIO : Developer's Guide

GT 4.2.0 XIO : Developer's Guide

Introduction

This guide contains information of interest to developers working with XIO. It provides reference information for application developers, including APIs, architecture, procedures for using the APIs and code samples.

Table of Contents

1. Before you begin 1
    1. Feature summary 1
    2. Tested platforms 1
    3. Backward compatibility summary 2
    4. Technology dependencies 2
    5. Security considerations for XIO 2
2. Using the Globus XIO API for IO operations within C programs 3
    1. Introduction 3
    2. Activate Globus 3
    3. Load Driver 3
    4. Create Stack 3
    5. Opening Handle 4
    6. Pay Off 4
3. Driver Quick Start 6
    1. Introduction 6
    2. Data Structures 6
    3. Attributes 6
    4. Target 9
    5. Handle 10
    6. IO Operations 10
    7. The Glue 11
4. Tutorials 12
5. Architecture and design overview 13
    1. Introduction 13
    2. Framework 13
    3. Drivers 14
    4. Example 14
    5. Driver Interface 16
6. XIO APIs 17
    1. Programming Model Overview 17
    2. Component API 17
7. Environment Variable Interface 18
    1. Environmental variables for XIO 18
8. Debugging 19
9. Troubleshooting 20
    1. Set GLOBUS_LOCATION correctly 20
    2. Errors 21
10. Related Documentation 24
A. Add-on 25
    1. Introduction 25
    2. UDT Reference Implementation Driver 25
    3. Bandwidth Limiting Driver 25
    4. Pipe Open Driver 25
    5. Bidirectional Driver 26
    6. Netlogger Driver 26
    7. Skeleton Driver 26

DRAFT

List of Figures

5.1. XIO Architecture 13
5.2. XIO Application 15

DRAFT

List of Tables

9.1. XIO Errors 22

DRAFT

Chapter 1. Before you begin

1. Feature summary

Features new in release 4.2.0

- Driver specific string attributes. Set values like *tcp buffer size* via string at runtime.
- UDT driver.
- Mode E Driver
- Telnet Driver
- Queuing Driver
- Ordering Driver
- Dynamically loadable drivers.

Other Supported Features

- Single API to swappable IO implementations.
- Asynchronous IO support.
- Native timeout support.
- Data descriptors for providing driver specific hints.
- Modular driver stacks to maximize code reuse.
- TCP, UDP, file, HTTP, telnet, mode E, GSI drivers.

Deprecated Features

- GSSAPI_FTP driver now distributed with the GridFTP Server

2. Tested platforms

Tested Platforms for XIO:

- Linux
  - Mandrakelinux release 10.1
  - SuSE Linux 9.1 (i586)
  - Debian GNU/Linux 3.1
  - Red Hat Linux release 9
- SunOS

DRAFT

    - SunOS 5.9 sun4u sparc SUNW,Sun-Fire-280R
- MacOS
    - Darwin Kernel Version 7.9.0
- Additionally, all platforms supported by GT 4.2.0 Platforms.

3. Backward compatibility summary

Protocol changes since GT version 4.0.x

- None.

API changes since GT version 4.0.x

- `globus_xio_stack_copy` added to the API. This allows a user to duplicated a configured stack.
- `globus_xio_driver_set_eof_received` added to the driver API. This function allows drivers to have multiple outstanding reads at one time.
- `globus_xio_driver_eof_received` added to the driver API. Working in conjunction with `globus_xio_driver_set_eof_received` to allow drivers to have multiple outstanding reads.
- Users can now pass in a NULL callback for timeouts and it is assumed that when time expires the user wants the operation to timeout. Previously a user callback was required where the user would decide if they wanted the timeout.

4. Technology dependencies

XIO depends on the following GT components:

- Globus Core
- Globus Common
- Globus GSSAPI

5. Security considerations for XIO

Globus XIO is a framework for creating network protocols. Several existing protocols, such as TCP, come built into the framework. XIO itself introduces no known security risks. However, all network applications expose systems to the risks inherent when outsiders can connect to them. Also included in the XIO distribution is the GSI driver, which provides a driver that allows for secure connections.

DRAFT

Chapter 2. Using the Globus XIO API for IO operations within C programs

1. Introduction

This Guide explains how to use the Globus XIO API for IO operations within C programs. Since Globus XIO is a simple API it is pretty straight forward. The best way to become familiar with it is by looking at an example. See globus_xio_example.c.

2. Activate Globus

Let's examine the case where a user wishes to use Globus XIO for reading data from a file. As in all globus programs the first thing that must be done is to activate the globus module. Until activated no globus_xio function calls can be successfully executed. It is activated with the following line:

```
globus_module_activate(GLOBUS_XIO_MODULE);
```

3. Load Driver

Let's examine the case where a user wishes to use Globus XIO for reading data. The next step is to load all the drivers needed to complete the IO operations in which you are interested. The function globus_xio_load_driver() is used to load a driver. In order to successfully call this function you must know the name of all the drivers you wish to load. For this example we only want to load the file IO driver. The prepackaged file IO driver's name is: "file". This driver would be loaded with the following code:

```
globus_result_t                     res;
globus_xio_driver_t                 driver;

res = globus_xio_driver_load(&driver, "file");
```

If upon completion of the above function call `res` returns GLOBUS_SUCCESS then the driver was successfully loaded and can be referenced with the variable "driver".

4. Create Stack

Now that globus_xio is activated and we have a driver loaded we need to build a driver stack. In our example the stack is very simple as it consists of only one driver, the file driver. The stack is established with the following code (building off of the above code snippets):

```
globus_xio_stack_t              stack;

globus_xio_stack_init(&stack);
globus_xio_stack_push_driver(stack, driver);
```

5. Opening Handle

Now that the stack is created we can open a handle to the file. There are two ways that a handle can be opened. The first is a passive open. An example of this is a TCP listener. The open is performed passively by waiting for some other entity to act upon it. The second is an active open. An active open is the opposite of a passive open. The TCP counter example for this is a connect. The users initiates the open. In our example we shall be performing an active open.

Before opening a handle it must be initialized. The following illustrates initialization for client side handles:

```
globus_xio_handle_t              handle;

res = globus_xio_handle_create(&handle, stack);
```

Server side handles are a bit more complicated. First we must introduce the data structure `globus_xio_server_t`. This structure shares many concepts with a TCP listener, mainly that it spawns handles ("connections") as passive open requests are made. If the user wishes to accept a new connection a call to globus_xio_accept() or globus_xio_register_accept() will initialize a new handle:

```
globus_xio_server_t          server;
globus_xio_handle_t          handle;
globus_result_t              res;

res = globus_xio_server_create(&server_handle, NULL, stack);
res = globus_xio_server_accept(&handle, server);
```

Once the handle is initialized it should be opened in order to perform read and write operations upon it. It the handle is a client then a "contact string" is required. This string represents the target that the user wishes to open:

```
globus_xio_attr_t            attr;
char *                       contact_string = "file:/etc/groups";

globus_xio_attr_init(&attr);
globus_xio_open(xio_handle, contact_string, attr);
globus_xio_attr_destroy(attr);
```

note: attrs can be used to color the behaviors of a handle. For a conceptual understanding, they are not important and a user is free to simple pass NULL wherever an attr is required.

Now that we have an open handle to a file we can read or write data to it with either globus_xio_read() or globus_xio_write(). Once we are finished performing IO operations on the handle globus_xio_close(handle) should be called.

6. Pay Off

This may seem like quite a bit of effort for simple reading a file, and it is. However the advantages become clear when exploring the swapping of other drivers. In the above example it would be trivial to change the IO operations from file IO to TCP, or HTTP, or ftp. All the the user would need to do is change the driver name string passed to glo-

DRAFT

bus_xio_load_driver() and the contact string passed to globus_xio_target_init(). This can easily be done at runtime, as the program globus_xio_example.c [1] demonstrates.

So the little program globus_xio_example.c [2] has the ability to be any reading client or server (HTTP, ftp, TCP, file, etc) as long as the proper drivers are in the LD_LIBRARY_PATH.

DRAFT

[1] globus_xio_example.c
[2] globus_xio_example.c

Chapter 3. Driver Quick Start

1. Introduction

This Guide explains how to create a transport driver for Globus XIO. For the purpose of exploring both what a transform driver is and how to write one this guide will walk through an example driver. The full source code for the driver can be found at Driver Example[1]. This example implements a file driver for globus_xio. If a user of globus_xio were to put this file at the bottom of the stack, they could access files on the local file system.

2. Data Structures

There are three data structures that will be explored in this example: attribute, target, and handle. The driver defines the memory layout of these data structures but the globus_xio framework imposes certain semantics upon them. It is up to the driver how to use them, but globus_xio will be expecting certain behaviors.

3. Attributes

Each driver may have its own attribute structure. The attribute gives the globus_xio user API an opportunity to tweak parameters inside the driver. The single attribute structure is used for all possible driver specific attributes:

1. Target attributes
2. Handle attributes
3. Server attributes

How each of these can use the attribute structure will be unveiled as the tutorial continues. For now it is simply important to remember there is attribute structure used to initiate of the driver ADTs.

A driver is not required to have an attribute support at all. However if the driver author chooses to support attributes the following functions must be implemented:

```
typedef globus_result_t
(*globus_xio_driver_attr_init_t)(
       void **                                              out_attr);

typedef globus_result_t
(*globus_xio_driver_attr_cntl_t)(
       void *                                                attr,
       int                                                      cmd,
       va_list                                               ap);

typedef globus_result_t
(*globus_xio_driver_attr_copy_t)(
       void **                                               dst,
       void *                                                 src);

typedef globus_result_t
```

[1] globus_xio_driver_example.c

```
(*globus_xio_driver_attr_destroy_t)(
       void *                                                    attr);
```

See driver API doc[2] for more information.

We shall now take our first look at the file Driver Example[3]. The file driver needs a way to provide the user level programmer with a means of setting the mode and flags when a file is open (akin to the POSIX function `open()`).

The first step in creating this ability is to a) define the attribute structure and b) implement the `globus_xio_driver_attr_init_t` function which will initialize it:

```
/*
 *  attribute structure
 */
struct globus_l_xio_file_attr_s
{
    int                                                                 mode;
    int                                                                 flags;
}

globus_result_t
globus_xio_driver_file_attr_init(
    void **                                                         out_attr)
{
    struct globus_l_xio_file_attr_s *            file_attr;

    /*
     *  create a file attr structure and initialize its values
     */
    file_attr = (struct globus_l_xio_file_attr_s *)
        globus_malloc(sizeof(struct globus_l_xio_file_attr_s));

    file_attr->flags = O_CREAT;
    file_attr->mode = S_IRWXU;

    /* set the out parameter to the driver attr */
    *out_attr = file_attr;

    return GLOBUS_SUCCESS;
}
```

The above simply defines a structure that can hold two integers, mode and flags, then defines a function the will allocate and initialize this structure. `globus_xio` hides much of the memory management of these attribute structures from the driver. However, it does need the driver to provide a means of copying them, and free all resources associated with them. In the case of the file driver example, these are both simple:

```
globus_result_t
globus_xio_driver_file_attr_copy(
        void **                                                     dst,
        void *                                                       src)
{
        struct globus_l_xio_file_attr_s *           file_attr;
```

[2] http://www.globus.org/api/c-globus-4.2.0/globus_xio_gridftp_driver/html/index.html
[3] globus_xio_driver_example.c

```
        file_attr = (struct globus_l_xio_file_attr_s *)
        globus_malloc(sizeof(struct globus_l_xio_file_attr_s));

        memcpy(file_attr, src, sizeof(struct globus_l_xio_file_attr_s));

        *dst = file_attr;

        return GLOBUS_SUCCESS;
}

globus_result_t
globus_xio_driver_file_attr_destroy(
        void *                                                          attr)
{
        globus_free(attr);

        return GLOBUS_SUCCESS;
}
```

The above code should be fairly clear. Obviously we need a method with which the user can set flags and mode. This is accomplished with the following interface function:

```
globus_result_t
globus_xio_driver_file_attr_cntl(
        void *                                                          attr,
        int                                                             cmd,
        va_list                                                         ap)
{
        struct globus_l_xio_file_attr_s *                   file_attr;
        int *                                                           out_i;

        file_attr = (struct globus_l_xio_file_attr_s *)attr;
        switch(cmd)
        {
                case GLOBUS_XIO_FILE_SET_MODE:
                        file_attr->mode = va_arg(ap, int);
                        break;

                case GLOBUS_XIO_FILE_GET_MODE:
                        out_i = va_arg(ap, int *);
                        *out_i = file_attr->mode;
                        break;

                case GLOBUS_XIO_FILE_SET_FLAGS:
                        file_attr->flags = va_arg(ap, int);
                        break;

                case GLOBUS_XIO_FILE_GET_FLAGS:
                        out_i = va_arg(ap, int *);
                        *out_i = file_attr->flags;
                        break;

                default:
```

```
                        return FILE_DRIVER_ERROR_COMMAND_NOT_FOUND;
                        break;
        }

        return GLOBUS_SUCCESS;
}
```

This function is called passing the driver an `initialized file_attr` structure, a command, and a variable argument list.

Based on the value of `cmd`, the driver decides to do one of the following:

- set flags or mode from the `va_args`
- return flags or mode to the user via a pointer in `va_args`

4. Target

A target structure represents what a driver will open. It is initialized from a contact string and an attribute. In the case of a file driver, the target simply holds onto the contact string as a path to the file.

The file driver implements the following target functions:

```
globus_result_t
globus_xio_driver_file_target_init(
        void **                                                              out_targ
        void *                                                                target_
        const char *                                                     contact_strin
        globus_xio_driver_stack_t                               stack)
{
        struct globus_l_xio_file_target_s *                 target;

        /* create the target structure and copy the contact string into it */
        target = (struct globus_l_xio_file_target_s *)
                    globus_malloc(sizeof(struct globus_l_xio_file_target_s));
        strncpy(target->pathname, contact_string, sizeof(target->pathname) - 1);
        target->pathname[sizeof(target->pathname) - 1] = '\0';

        return GLOBUS_SUCCESS;
        }

/*
 *  destroy the target structure
 */
globus_result_t
globus_xio_driver_file_target_destroy(
        void *                                                                target)
{
        globus_free(target);

        return GLOBUS_SUCCESS;
}
```

The above function handles the creation and destruction of the file driver's target structure.

Note

When the target is created, the contact string is copied into it. It is invalid to just copy the pointer to the contact string. As soon as this interface function returns, that pointer is no longer valid.

5. Handle

The most interesting of the three data types discussed here is the handle. All typical IO operations (open, close, read, write) are performed on the handle. The handle is the initialized form of the target and an attr. The driver developer should use this ADT to keep track of any state information they will need in order to perform reads and writes.

In the example case, the driver handle is fairly simple as the driver is merely a wrapper around POSIX calls:

```
struct globus_l_xio_file_handle_s
 {
        int                                  fd;
};
```

The reader should review the following functions in Driver Example[4] in order to see how the handle structure is used:

- `globus_xio_driver_file_open()`
- `globus_xio_driver_file_write()`
- `globus_xio_driver_file_read()`
- `globus_xio_driver_file_close()`

6. IO Operations

The read and write interface functions are called in response to a user read or write request. Both functions are provided with a vector that has at least the same members as the `struct iovec` and a vector length. As of now, the iovec elements may contain extra members, so if you wish to use `readv()` or `writev()`, you will have to transfer the `iov_base` and `iov_len` members to the POSIX iovec.

As with the open and close interface functions, if an error occurs before any real processing has occurred, the interface function may simply return the error (in a `result_t`), effectively canceling the operation. However, once bytes have been read or written, you must not return the error. You must report the number of bytes read/written along with the result.

When an operation is done, either by error or successful completion, the operation must be 'finished'. To do this, a call must be made to:

```
globus_result_t
globus_xio_driver_finished_read/write(
    globus_xio_driver_operation_t                    op,
    globus_result_t                                            res,
    globus_ssize_t                                             nbytes);
```

DRAFT

[4] globus_xio_driver_example.c

6.1. Blocking vs Non-blocking Calls

In general, the driver developer does not need to concern himself with how the user made the call. Whether it was a blocking or an asynchronous call, XIO will handle things correctly.

However the call was made, the driver developer can call globus_xio_driver_finished_{open, read, write, close} either while in the original interface call, in a separate thread, or in a separate callback kick out via the globus_callback API.

7. The Glue

Through a process not finalized yet, XIO will request the `globus_xio_driver_t` structure from the driver. This structure defines all of the interface functions that the driver supports. In detail:

```
/*
 *  main io interface functions
 */
globus_xio_driver_open_t                                                    open_func;
globus_xio_driver_close_t                                                   close_func;
globus_xio_driver_read_t                                                     read_func;
globus_xio_driver_write_t                                                     write_func
globus_xio_driver_handle_cntl_t                                        handle_cntl_func;

globus_xio_driver_target_init_t                                            target_init_fu
globus_xio_driver_target_destroy_t                                   target_destroy_finc

/*
 * target init functions.  Must have client or server
 */
globus_xio_driver_server_init_t                                            server_init_fun
globus_xio_driver_server_accept_t                                    server_accept_func;
globus_xio_driver_server_destroy_t                                   server_destroy_func;
globus_xio_driver_server_cntl_t                                            server_cntl_func

/*
 *  driver attr functions.  All or none may be NULL
 */
globus_xio_driver_attr_init_t                                                attr_init_f
globus_xio_driver_attr_copy_t                                              attr_copy_func
globus_xio_driver_attr_cntl_t                                                attr_cntl_fu
globus_xio_driver_attr_destroy_t                                         attr_destroy_fun

/*
 *  data descriptor functions.  All or none
 */
globus_xio_driver_data_descriptor_init_t                          dd_init;
globus_xio_driver_driver_data_descriptor_copy_t           dd_copy;
globus_xio_driver_driver_data_descriptor_destroy_t       dd_destroy;
globus_xio_driver_driver_data_descriptor_cntl_t              dd_cntl;
```

DRAFT

Chapter 4. Tutorials

- PowerPoint Tutorial[1]

DRAFT

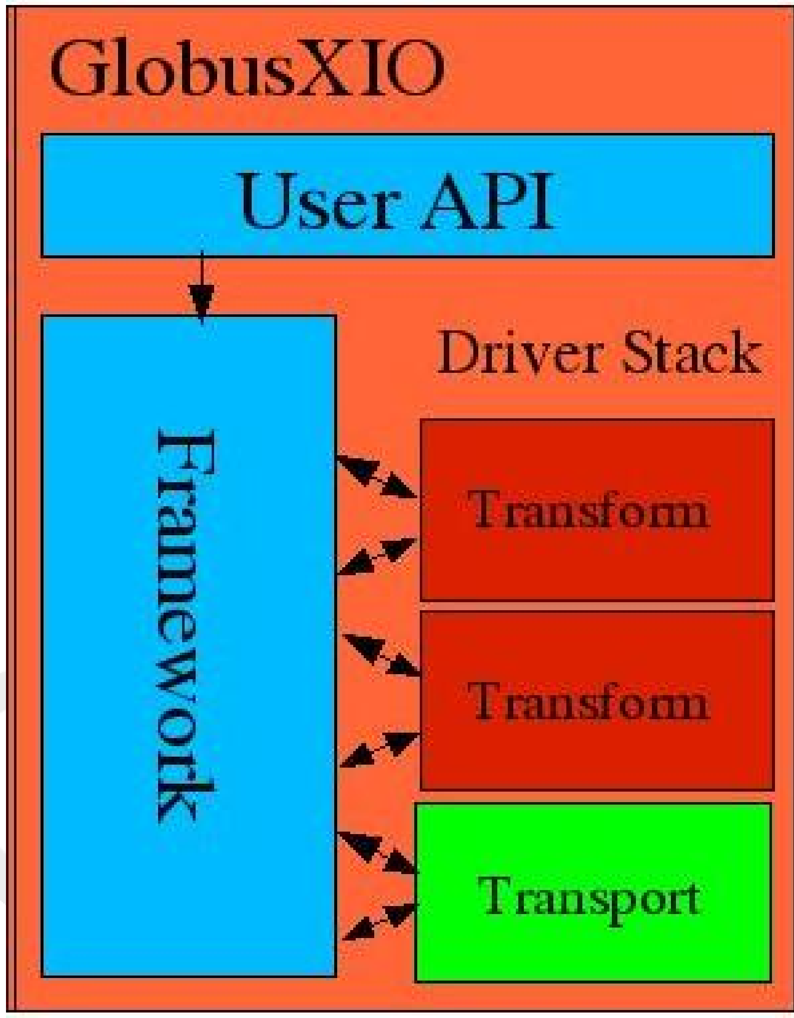
[1] xio.ppt

Chapter 5. Architecture and design overview

1. Introduction

This document shall explain the external view of the Globus XIO architecture. Globus XIO is broken down into two main components, framework and drivers. The following picture illustrates the architecture:

Figure 5.1. XIO Architecture



2. Framework

The Globus XIO framework manages IO operation requests that an application makes via the user API. The framework does no work to deliver the data in an IO operation nor does it manipulate the data. All of that work is done by the drivers. The framework's job is to manage requests and map them to the drivers interface. It is the drivers themselves that are responsible for manipulating and transporting the data.

3. Drivers

A driver is the component of Globus XIO that is responsible for manipulating and transporting the users data. There are two types of drivers, transform and transport. Transform drivers are those that manipulate the data buffers passed to it via the user API and the XIO framework. Transport drivers are those that are capable of sending the data over a wire.

Drivers are grouped into stacks, that is, one driver on top of another. When an IO operation is requested the Globus XIO framework passes the operation request to every driver in the order they are in the stack. When the bottom level driver (the transport driver) finishes shipping the data, it passes the data request back to the XIO framework. Globus XIO will then deliver the request back up the stack in this manner until it reaches the top, at which point the application will be notified that there request is completed.

In a driver stack there can only be one transport driver. The reason for this is that the transport driver is the one responsible for sending or receiving the data. Once this type of operation is performed it makes no sense to pass the request down the stack, as the data has just been transfered. It is now time to pass the operation back up the stack.

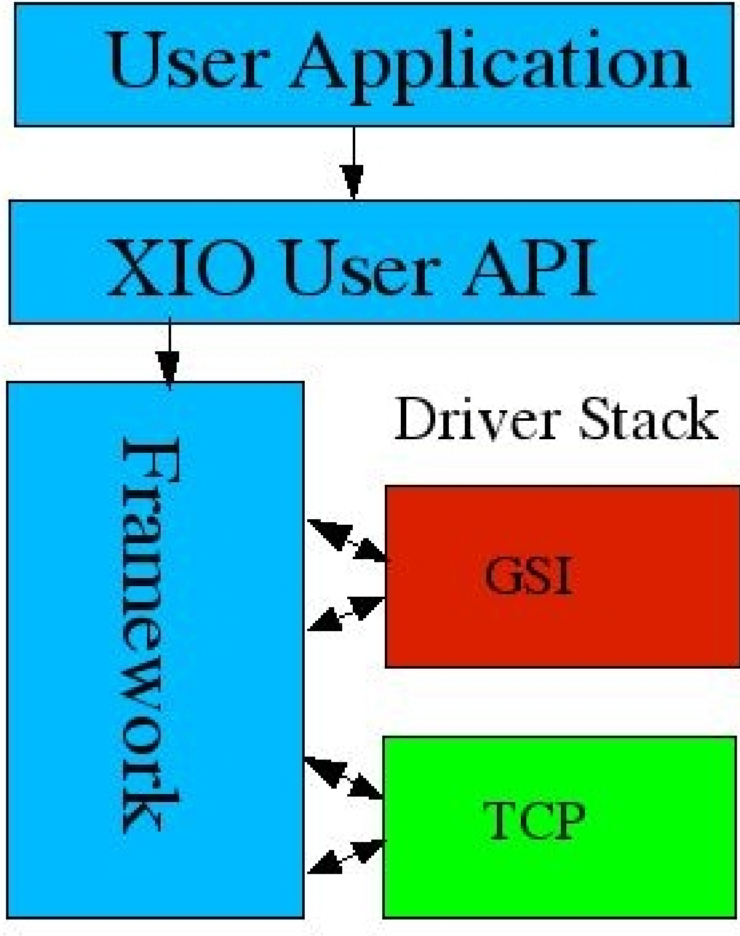
There can be many transform drivers in any given driver stack. A transform driver is one that can manipulate the requested operations as they pass. Some good examples of transform drivers are security wrappers and compression. However, a transport driver can also be one that adds additional protocol. For example a stack could consist of a TCP transport driver and an HTTP transform driver. The HTTP driver would be responsible for marshaling the HTTP protocol and the TCP driver would be responsible for shipping that protocol over the wire.

4. Example

In the following picture we illustrate a user application using Globus XIO to speak the GridFTP protocol across a TCP connection:

DRAFT

Figure 5.2. XIO Application



The user has built a stack consisting of one transform driver and one transport driver. TCP is the transport driver in this stack, and as all transport modules must be, it is at the bottom of the stack. Above TCP is the GSI transform driver which performs necessary messaging to authenticate a user and the integrity of the data.

The first thing the user application will do after building the stack is call the XIO user API function globus_xio_open(). The Globus XIO framework will create internal data structures for tracking this operation and then pass the operation request to the GSI driver. The GSI driver has nothing to do before the underlying stack has opened a handle so it simply passes the request down the stack. The request is thereby passed to the TCP driver. The TCP driver will then execute the socket level transport code contained within it to establish a connection to the given contact string.

Once the TCP connection has been established the TCP driver will notify the XIO framework that it has completed its request and thereby the GSI driver will be notified that the open operation it had previously passed down the stack has now completed. At this point the GSI driver will start the authentication processes (note that at this point the user does not yet have an open handle). The GSI driver has an open handle and upon it several sends and receives are performed to authenticate the connection. If the GSI driver is not satisfied with the authentication process it closes the handle it has to the stack below it and tells the XIO framework that it has completed the open request with an error. If it is satisfied it simply tells the XIO framework that it has completed the open operation. The user is now notified that the open operation completed, and if it was successful they now have an open handle.

Other operations work in much the same way. When a user posts a read the read request is first delivered to the GSI driver. The GSI driver will wrap the buffer and pass the modified buffer down the stack. The framework will then de-

liver the write request with the newly modified buffer to the TCP driver. The TCP driver will write the data across the socket mapped to this handle. When it finishes it notifies the framework, which notifies the GSI driver. The GSI driver has nothing more to do so it notifies the framework that it is complete and the framework then notifies the user.

5. Driver Interface

There is a well defined interface to a driver. Drivers are modular components with specific tasks. The purpose of drivers in the Globus XIO library is extensibility. As more and more protocols are developed, more and more drivers can be written to implement these protocols. As new drivers are written they can be added to Globus XIO as either statically linked libraries or dynamically loaded libraries. In the case of dynamic loading it is not even necessary to recompile existing source code. Each driver has a unique name according to the Globus XIO driver naming convention. A program simply needs to be aware of this name (this can be passed in via the command line) and the Globus XIO framework will be responsible for loading the driver.

Note

The above example is simplified for the purposes of understanding. There are optimizations built into Globus XIO which alter the course of events outlined above. However, conceptually the above is accurate.

DRAFT

Chapter 6. XIO APIs

1. Programming Model Overview

Globus XIO is based on an event based asynchronous programming model. This model is described in great detail at: Asynchronous Event Handling. In short, with Globus XIO, connections are opened and closed. While open, read and write requests are posted with a callback function pointer given by the user. When the event completes, the given callback is called.

2. Component API

You can find documentation of the XIO library at:

http://www.globus.org/api/c/globus_xio/html/index.html

For information on the internationalization API, see Chapter 1, APIs.

DRAFT

Chapter 7. Environment Variable Interface

1. Environmental variables for XIO

The vast majority of the environment variables that effect the Globus XIO framework are defined by the driver in use. The following are links to descriptions of the more common driver environment variables:

- http://www.globus.org/api/c-globus-4.2.0/globus_xio/html/group__tcp__driver__envs.html
- http://www.globus.org/api/c-globus-4.2.0/globus_xio/html/group__file__driver__envs.html
- http://www.globus.org/api/c-globus-4.2.0/globus_xio/html/group__gsi__driver__envs.html
- http://www.globus.org/api/c-globus-4.2.0/globus_xio/html/group__udp__driver__envs.html

DRAFT

Chapter 8. Debugging

All standard C debugging techniques apply to debugging XIO applications. Also, Globus XIO provides users with some additional debugging information. If the environment variable GLOBUS_XIO_DEBUG is set debugging information will be written to a file or stdout. The information generated is particularly useful to identify a suspect bug in Globus XIO. GLOBUS_XIO_DEBUG is set in the following way:

```
GLOBUS_XIO_DEBUG="<level>  [,[[#]<file name>][,<flag>[,<timestamp_levels>]]"
```

The value of <level> can take on the logical OR of any of the following:

- GLOBUS_XIO_DEBUG_ERROR = 1
- GLOBUS_XIO_DEBUG_WARNING = 2
- GLOBUS_XIO_DEBUG_TRACE = 4
- GLOBUS_XIO_DEBUG_INTERNAL_TRACE = 8
- GLOBUS_XIO_DEBUG_INFO = 16
- GLOBUS_XIO_DEBUG_STATE = 32
- GLOBUS_XIO_DEBUG_INFO_VERBOSE = 64

<file name> is a debug output file, if empty stderr will be used by default. If a '#' precedes the filename, the file will be overwritten on each run. Otherwise, the output will be appended to the existing file.

<flags>

- 0 = default
- 1 = show thread ids
- 2 = append the pid to debug filename

DRAFT

Chapter 9. Troubleshooting

For a list of common errors in GT, see Error Codes.

1. Set GLOBUS_LOCATION correctly

- The environment variable `GLOBUS_LOCATION` must be set to a valid Globus 4.2.0 installation.
- Various other environment variables must be set in order to easily use the GlobusXIO application. The proper environment can be established by running: `source $GLOBUS_LOCATION/etc/globus-user-env.sh` or `source $GLOBUS_LOCATION/etc/globus-user-env.csh` depending on the shell you are using.

DRAFT

2. Errors

Table 9.1. XIO Errors

Error Code	Definition	Possible Solutions
`Operation was canceled`	An I/O operation has been canceled by a close or a cancel	In most cases this will be intentionally performed by the application developer. In unexpected cases the applciation developer should verify that there is not a race condition relating to closing a handle.
`Operation timed out`	Occurs when the application developer associates a timeout with a handle's I/O operations. If no I/O is performed before the timeout expires this error will be triggered.	The remote side of connection might be hung and busy. The network could have higher latencies than expected. The filesystem might be over worked.
`An end of file occurred`	This occurs when and EOF is detected on the file descriptor	When doing file I/O this like means you read to the end of the file and thus you are finished and should now close it. On network connections however it means the socket was closed on the remote end. This can happen it the remote side suddenly dies (segfault is common here) or if the remote side chooses to close the connection.
`Contact string invalid`	A poorly formed contact string was passed in to open	Verify the format of the contact string with the documentation of the drivers in use.
`Memory allocation failed on XXXX`	malloc failed. The system is likely quite overloaded	Free up memory in your application
`System error in XXXX`	A low level system error occurred. The errno and errstring should indicate more information.	
`Invalid stack`	The requested stack does not meet XIO standards	Most likely a transport driver is not on the bottom of the stack, or 2 transport drivers are in the stack.
`Operation already registered`	With certain common drivers like TCP and FILE, only one specific operations can be registered at a time (1 read, 1 write). If another operation of the same type is posted to the handle before receiving the previous operations callback, this error can occur.	Restructure the application code so that it waits for the callback before registering the next IO operation.
`Unexpected state`	The internal logic of XIO came across a logical path that should not be possible. Often times this is due to application memory corruption or trying to perform an IO operation on a closed or otherwise invalid handle.	Use valgrind or some sort of memory management tool to verify there is no memory corruption. Try to recreate the problem in a small program. Submit the program and the memory trace at bugzilla.globus.org
`Driver in handle has been unloaded`	A driver associated with the offending operation has already been unloaded by the application code.	Verify that you are not unloading drivers until they are no longer in use.

Error Code	Definition	Possible Solutions
`Module not activated`	globus_module_activate(GLOBUS_XIO_MODULE); has not been called.	Call this before making any other XIO API calls.

DRAFT

Chapter 10. Related Documentation

- Performance [1]
- Previous HTML doc [2]

DRAFT

[1] http://www-unix.mcs.anl.gov/~kettimut/xio/XIO_Performance.pdf
[2] http://www-unix.mcs.anl.gov/~bresnaha/xio

Appendix A. GT 4.2.0 XIO : Driver Add-ons

1. Introduction

Here we describe a GPT bundle of some useful GlobusXIO drivers. The drivers are not packaged with the standard globus release because they require external dependencies, or are otherwise best distributed in a separate release. Here you can get each driver individually, or a bundle of all the drivers.

2. UDT Reference Implementation Driver

This driver uses the reference implementation of UDT provided at LINK and the Globus XIO wrapblock feature to create a Globus XIO UDT driver.

3. Bandwidth Limiting Driver

This is a bandwidth limiting driver. It allows the user to specify a bandwidth cap for a given connection.

Features

•

String Options

- rate=
- read_rate=
- write_rate=
- period=
- read_period=
- write_period=

4. Pipe Open Driver

This driver will run an executable program with pipes connecting STDIN and STDOUT of the program to this Globus XIO driver. In this way buffers are routed from Globus XIO to the executable. This driver is most notably used with ssh for running remote programs over a secure link.

String Options

- blocking=
- pass_env=
- argv=

DRAFT

5. Bidirectional Driver

The BIDI driver, bi-directional driver allows a user to turn a unidirectional pipe, into a bidirectional stream. It does this simply by opening up 1 uni-directional pipe in each direction. By default it assumes the use of the ModeE driver, but it can be configured to use any driver.

Important attribute controls are listed here. For a complete list see: BIDI API[1]

- GLOBUS_XIO_BIDI_SET_READ_STACK
- GLOBUS_XIO_BIDI_SET_WRITE_STACK
- GLOBUS_XIO_BIDI_SET_READ_ATTR
- GLOBUS_XIO_BIDI_SET_WRITE_ATTR

6. Netlogger Driver

This driver observes IO events on a driver stack and uses netlogger to log these events.

String Options

- filename=
- mask=
- type=
- id=

7. Skeleton Driver

This is a very simple driver to serve as an example and base for writing additional drivers. It does nothing but pass operations along the stack

[1] http://www.globus.org/api/c-globus-4.2.0/globus_xio_bidi_driver/html/index.html