# The Globus eXtensible Input/Output System (XIO): A protocol independent IO system for the Grid

**William Allcock, John Bresnahan, Rajkumar Kettimuthu, Joseph Link**
Argonne National Laboratory, Globus Alliance

## Abstract

***In distributed heterogeneous Grid environments the protocols used to exchange bits are crucial. As researchers work hard to discover the best new protocol for the Grid, application developers struggle with ways to use these new protocols. A stable, consistent, and intuitive framework is needed to aid in the implementation and use of these protocols. While the application must not be burdened with the protocol details some of it may need to be exposed to take advantage of potential optimizations. In this paper we examine how the Globus XIO API provides this framework. We will explore the performance implications of using this abstraction layer and the benefits gained in application as well as protocol development.***

## Introduction

It is amazing -- or frustrating, depending on your point of view -- how many I/O problems appear to be solvable with simple Open/Close/Read/Write (OCRW) functionality. The devil is in the details, however, and the solutions to these problems are just different enough to result in a variety of API models with varying sets of semantics. From a very broad perspective they all look alike, since they all have a source and a sink between which a stream of data flows. However, a more detailed examination shows differences that are extremely significant to the application developer. For example, if the end goal of an application is to transfer files between two locations, HTTP [7] and GridFTP [1,2,3,4] may be logical choices. The choice between them may be based on performance and environment characteristics, but the architecture and design of the application is the same. However when looked at from the perspective of a developer actually implementing that design, the asynchronous push model FTP API, and the synchronous multi-threaded API for HTTP messaging is entirely different and drastically affects the design of the application.

The Grid [5,6] is a heterogeneous, dynamic and evolving environment. Development of communication protocols to optimize use of and access to the underlying fabric of the Grid is a key area of research. This research turns out protocols that have similar goals but with varying properties ideal for different environments. As researchers develop new transfer and control protocols, application developers struggle to update their applications in order to gain the benefits of increased network utilization. In order to make their applications portable to many heterogeneous grid environments, the developers may try to integrate as many protocols as possible. This typically means many different APIs all with different semantics, many different programming models, and even different sets of bugs to work around. This issue certainly leads to a longer development cycle and inevitably leads to messy code.

On the other end of this problem is the creation of new protocols. As researchers and developers create new protocols the focus of their effort should be on the details of their protocol, yet too often a majority of time is spent on side issues. The development of these new protocols can be significantly delayed by issues that have more to do with user interface than they do with the protocol itself. Designing a good intuitive API is a task in itself, and one that is unfortunately often neglected. Further, the majority of bugs and development time in a robust full featured protocol library are results of race conditions due to multi-threading and event notification. These issues are paramount to the implementation of a protocol but not central to the protocol itself. Additionally, most new protocols are layered on top of standard transport protocols, like TCP [10]. This leaves the developer coding directly to the socket library, which is a difficult art to master.

## Overview of Globus XIO

Globus XIO is a simple OCRW abstraction layer to transport protocols that was designed with these problems in mind. To application developers, Globus XIO is an API that enables different I/O problems to be presented uniformly as a simple OCRW interface with a single set of semantics. To network protocol developers, it is a support framework for developing communication protocols. To mass storage vendors, it is an interface that enables an existing application written with XIO to access their hardware.

Globus XIO does not try to solve every I/O problem by mapping every type of I/O to a single API; that is clearly not possible. Nor does Globus XIO attempt to hide all details of the underlying protocol; that is clearly undesirable. While it is nice to think that all I/O can be lumped into a single abstraction, protocols are different for a reason. They solve different problems and do different jobs. Often a user must be able to control these differences to properly take full advantage of the I/O protocol involved.

The first problem that Globus XIO addresses is the abstraction of byte stream IO into a simple OCRW API. It is unnatural (and unwise) to attempt to map all I/O types to an OCRW API. However, those that can be viewed as streams of bytes, which is a significant number of them, map very well. All the differences cannot be hidden, but the impact on the application code can be minimized and the differences isolated. So whether data is read from a file or a network or an electron microscope, whether it is using TCP, UDP [11], or some custom protocol, to the application using Globus XIO the interface is simply OCRW.

Globus XIO also helps to minimize the development time and effort involved in creating and prototyping new I/O protocols and new device interfaces. Globus XIO provides a framework that handles many of the non-protocol specific ancillary requirements of a protocol developer such as event handling, error handling, etc. The protocol specific functionality is abstracted into a driver, where the protocol developer can focus on the interesting details of the protocol they are prototyping or producing for production.

The driver must implement a well-defined set of interface functions, with a well-defined set of semantics associated with them. Essentially, these functions consist of Open, Close, Read, and Write. For instance, a driver developer defines an open function, and implements the code specific to their protocol for opening. The same steps are taken for the read, write and close functions. When implementing these functions the developer does not need to be concerned with user error checking, this has all been done by the framework and all parameters are guaranteed to be good. This and all other user interface issues are removed from the concern of the driver developer.

Often, APIs are created according to a blocking model. This has the advantage of being much easier to design and implement, however it will quite possibly have performance issues. It is likely that while an application is waiting for bytes to arrive via some transport that it can perform data processing. If using a blocking API the only way an application can do these parallel tasks is to have separate threads for IO and processing. This can easily lead to complicated synchronization issues that a scientific application may not be willing to deal with. Further, it does not scale well as more and more simultaneous IO occurs. To over come this issue the protocol developer must create their own polling loop, which can be very complicated low level code and is nothing more than a distraction to the protocol developer. The Globus XIO framework alleviates this problem from the driver developer by providing asynchronous functionality. A rich set of system polling code is distributed with XIO, and an internal API provides the driver with polling and barriers and other asynchronous functionality.

One important feature of XIO is that drivers can also use other drivers. This means that once a TCP driver is created (and one is distributed with Globus XIO) no other driver needs to code directly to sockets again, they can instead use the TCP driver. So when creating the HTTP driver, the details of framing HTTP messages can be focused on and the work of transporting the messages can simply be passed onto the TCP driver.

## A Common Usage Scenario

One of the common Grid computing usage scenarios is referred to as a "DataGrid" problem [1]. This involves accessing a remote data source, transporting it to a compute node, performing computation on it, and then sending the results to a remote destination. This scenario involves multiple IO operations, each of which may involve a different programming model, API, etc. We will examine this scenario in some detail to see how XIO might be applied.

In our scenario, a remote data source is accessed for input data to the computation. Common access methodologies for this data might include GridFTP, HTTP, or a proprietary mass storage system such as HPSS [8,9]. However, the source of data could also be something more unusual such as a scientific instrument of some kind or even the output of another computational program. In all these cases, this IO can be easily represented as a byte stream, yet the programming model for each could (and likely would) be quite different. If the application wishes to be able to access all these sources, the application must contain code for all these proprietary APIs and deal with all of the different programming models. However, if we interpose XIO as an abstraction layer, the application sees a uniform programming model, uniform API, uniform error semantics, etc. Obviously, the differences are still present, but they have been isolated to the driver, making the application logic much cleaner and localizing the location of source specific issues.

Our scenario also calls for sending the results of the computation to a remote destination. It might be the same or similar access methods as already described, but what if it were a visualization display of some kind? This case is much less clear and it depends on the characteristics of the display. If the display were some sort of movie viewer and expected a simple video stream, then XIO might well be appropriate. However, if complex windowing operations were required, XIO would definitely NOT be appropriate.

## Basic Architecture

There are two important concepts in the architecture of Globus XIO, the driver and the stack. We discussed the driver above; it is the abstracted component where all the protocol specific implementation exists. There are two types of drivers, transform drivers and transport drivers. Transport drivers are those that actually move data into or out of the process space. Examples of this are TCP and UDP. Transform drivers are those that manipulate, examine, frame, or change the data, or in other words, drivers that take any action other than moving the data across the process boundary. Examples would be compression, logging, and HTTP.

A particular protocol implementation can involve many drivers; this is where the concept of a stack of drivers comes into play. In a stack there must only be one transport driver, and it must be at the bottom of the stack. This stands to reason since the transport driver is what actually moves the bits on a wire. There can be any number of transform drivers on a driver stack. As data is written it moves from the user to the first driver on the stack. The data is moved down the stack, through the effects of each transform driver (if any are used) to the transport driver. The transport driver then ships the data over the wire. Data follows a similar path for reads, only in the opposite direction, from wire to transport driver, and up the stack to the user.

Driver stacks can be mixed and matched. An HTTP driver has been created as a transform driver, and typically it sits in a stack directly above TCP. For example, if a driver is created implementing UDT (a reliability layer over UDP) a stack can be formed of HTTP on top of UDT without a single code change to any of the drivers involved.

## The Globus XIO User API

Our discussion up to this point has been fairly abstract. At this point we will give a brief overview of the status of Globus XIO and some code snippets demonstrating the use of the user API.

At the time of this writing, Globus XIO is available in the alpha version of version 3.2 of the Globus Toolkit. By publication, GT3.2 should have reached final release. Documentation for the entire API is available at [12]. Globus XIO is a C library. Since the core API is simply OCRW, the user API is fairly straightforward. We provide synchronous calls (globus_xio_read()), asynchronous calls (globus_xio_register_read()) and have vector variants of each (globus_xio_[register]_readv()). The best way to become familiar with Globus XIO is by looking at some code snippets.

***Step 1: Activate Globus***

XIO was developed by the Globus Alliance and so inherits some Globus Toolkit semantics. Accordingly, as with all Globus Toolkit programs, you must first activate the Globus module. Until activation is complete, no XIO function calls can be successfully executed. The module is activated with the following line:

```
globus_module_activate(GLOBUS_XIO_MODULE);
```

***Step 2: Load Driver***
The next step is to load all the drivers needed to complete the I/O operations in which you are interested. The function globus_xio_load_driver() is used to load a driver. To successfully call this function, you must know the name of all the drivers you wish to load. For this example we want to load only the file io driver. The prepackaged file io drivers name is "file." This driver is loaded with the following code:

```
globus_result_t        res;
globus_xio_driver_t    driver;
res = globus_xio_load_driver(&driver, "file");
```

If upon completion of the above function call, the variable res is equal to GLOBUS_SUCCESS, then the driver was successfully loaded and can be referenced with the variable "driver."

***Step 3: Create Stack***
Once globus_xio is activated and a driver loaded, you need to build a driver stack. In our example the stack consists of only one driver, the file driver. The stack is established with the following code (building from the above code snips):

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

***Step 4: Opening the Handle***
Once the stack is created, you can open a handle to the file, in one of two ways. The first way is a passive open. An example is a TCP listener. The open is performed passively and then it waits for some other entity to act on it. The other alternative is an active open. An example is a TCP connect. The user, who initiates the open, performs the open actively. Our example has an active open. To create a handle for an active open, you first initialize the handle object and then open it with the contact information. The following code illustrates this:

```
globus_xio_handle_t             handle;

globus_xio_handle_create(&handle, stack);
res = globus_xio_open(handle, "/tmp/junk.txt", NULL);
```

***Step 5: The Payoff***
Now that you have an open handle to a file, you can read or write data to it with either: globus_xio_read() or globus_xio_write(). Once you are finished performing I/O operations on the handle, you should call globus_xio_close(handle).

All this may seem like a lot of effort for simply reading a file. The advantages become clear, however, when you wish to use other drivers. In the above example, it would be trivial to change the I/O operations from file I/O to TCP, HTTP, or FTP. All you would need to do is change the driver name string passed to globus_xio_load_driver() and change the contact string passed to globus_xio_[register]_open(). Both can be done easily at runtime, as the program globus_xio_example.c [12] demonstrates.

## The UDT Driver

In network protocol research a common goal is optimal bandwidth utilization, while still being network friendly. The drawback of TCP inherent in its AIMD based congestion control mechanism [13] is well known [14]. Furthermore, if the bandwidth-delay product is very high, it takes a very long time for TCP to recover from a packet loss. Researchers have come up with numerous alternatives. These solutions include improvements to TCP [15,16,17], new transport protocols such as XCP [18], XTP [19] and reliable layers on top of UDP [20,21,22].

Globus XIO is the perfect framework for researching reliability layers on top of UDP. A UDP driver already exists, so all of the socket code is taken care of and the framework assists greatly in quick prototyping of protocols. Further, since the user API does not change based on the driver used, a single set of performance measuring tools can be used to compare many drivers.

One particular protocol in this category is UDT (developed at the Laboratory for Advanced Computing at University of Illinois Chicago) [23, 24]. A UDT driver was implemented for the Globus XIO framework. Globus XIO allows the driver developer to load and control the desired protocol underneath the driver. This feature is very useful for developing drivers that are in an evolving state like SABUL/UDT. SABUL was the first iteration of UDT and used TCP for exchanging control messages and UDP for transferring the data. Later, an improved version of SABUL, namely UDT, which use UDP for transferring both control and data, was proposed. These changes were made easily in Globus XIO by simply altering the driver loaded. This would have been a much more substantial effort if the development were done using the socket library.

The provision of both synchronous and asynchronous API and the framework handling the technicalities of that made the development of UDT easier. There was little worry of synchronizing processing by waiting and signaling. Instead the framework abstracted user requests for data into operations. When a request to read data from UDT was made it was given an operation. UDT then checks its read buffer and if it had already read the requested amount of data, it finishes the operation immediately. Otherwise it puts the operation in the read operation queue. Similarly, when a request to write is made, the buffer and the operation would be put in the queue. It then will split the data into multiple packets, add header and send it to the destination. Whenever an operation was complete, the driver simply notified the framework and the framework took care of all functionality necessary to safely notify the user.

Additionally, the Globus XIO framework allows the drivers to initiate new asynchronous operations that are performed in parallel to user initiated operations. This is an important feature for UDT, and any UDP based reliable transport protocols as they need to allow for the protocol layer to continuously read the data. Without this feature of Globus XIO the protocol developer would have to use synchronous operations and manage threads internally.

Also, when the user initiates a close, the XIO framework makes sure all the outstanding operations complete/cancel and its callback called before the close callback is called. This makes the driver developer's job easier by relieving him from worrying about canceling or waiting for the outstanding operations while performing the close operation.

## Performance Results

In computer science there is the old adage that there is no problem you cannot solve by adding a layer of abstraction, and there is no optimization you cannot make by removing one. Since Globus XIO is an abstraction layer to protocols it is important to look at its performance characteristics. We have conducted several performance tests bench marking Globus XIO against Globus IO (an API with similar features, yet it has been coded directly against the socket library), and the standard socket library. We did both latency and bandwidth tests on a local host, between two different hosts on a LAN, between two hosts connected over a wide area link.

### *Latency Tests*

The latency tests were carried out in a ping-pong fashion. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Each ping-pong test was carried out 1000 times, the total time to complete those iterations was then used to determine an average one-way latency number.

Figures 1 through 3 show the results of the latency tests. For tests conducted on the loopback interface, shown in Figure 1, the latency of a single byte increased from 20 us for raw socket IO to 37 us for IO via Globus XIO. However, the percentage increase in latency decreases as the message size increases. For a message of size of 100KB, the percentage increase is 8 % and for a message size of 10 MB, it is just 2%. Tests conducted on the LAN provide much better results, as shown in figure 2. The percentage increase in latency for Globus XIO is 8% for transferring a 1000 byte message and it decreases as the message size increases and it is only 0.003% for a message of size of 10MB. Finally, figure 3 shows the latency numbers over a WAN. It is here that Globus XIO comes into its own. The overhead introduced is less than 1.5% for any message size.

In general, for most Globus applications, latency is not a critical issue and the use of Globus XIO will be of negligible impact. In certain latency critical applications, such as MPI application the utility of XIO would need to be determined with more exhaustive tests.

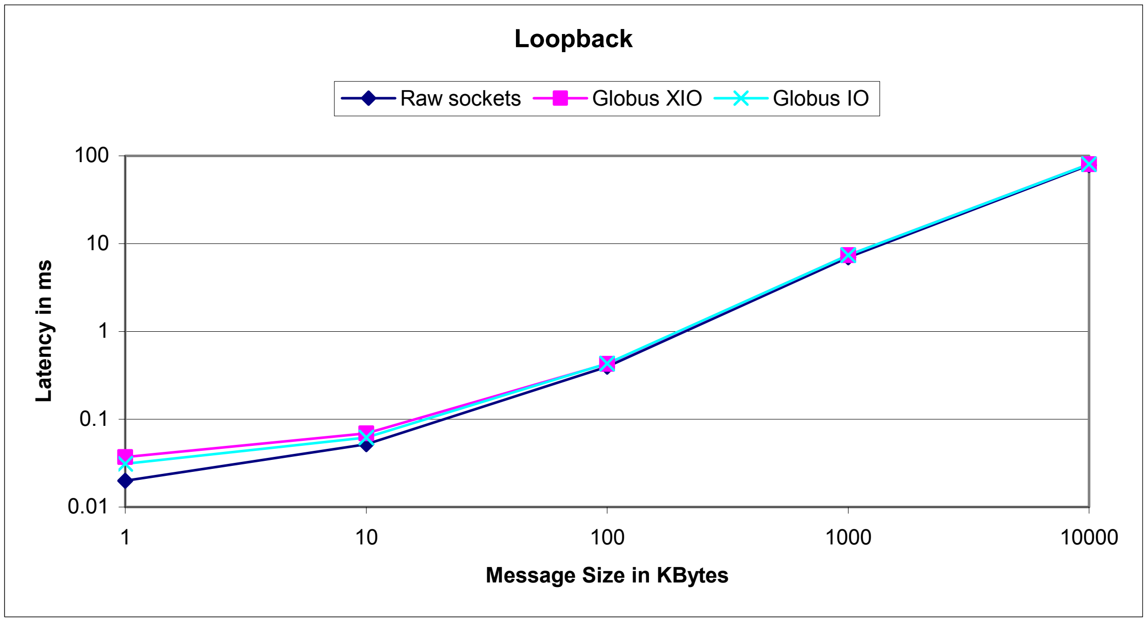

Figure 1: Comparison of one-way latency for transferring messages of various sizes over the loopback interface on a local host.

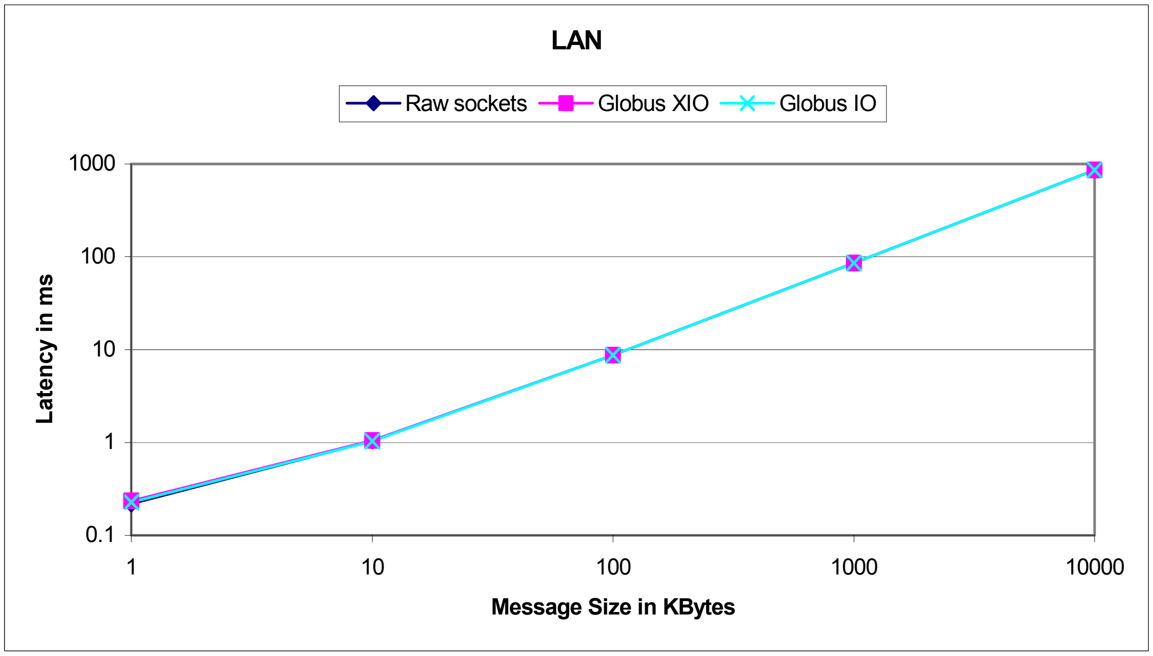

Figure 2: Comparison of one-way latency for transferring messages of various sizes between two hosts on a LAN

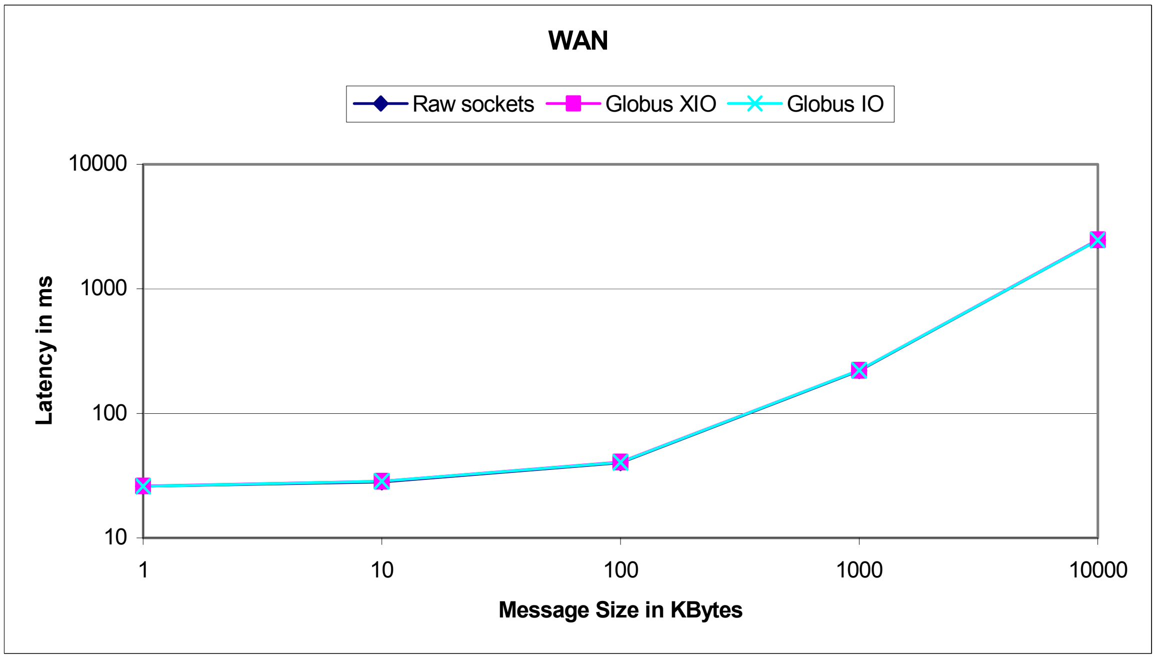

Figure 3: Comparison of one-way latency for transferring messages of various sizes between two hosts connected over a wide area link

***Bandwidth Tests***

The bandwidth tests were carried out by having the sender sending out a large number (1000) of back-to-back messages to the receiver and then waiting for a reply from the receiver. The receiver sends the reply only after receiving all (1000) messages. Then the bandwidth was calculated based on the elapsed time (from the time sender sends the first message until the time it receives the reply back from the receiver) and the number of bytes sent by the sender.

Figures 4 and 5 indicate that the throughput achieved with Globus XIO is nearly the same as that of raw sockets. The percentage decrease in the throughput is less than 3% in all cases, and for large messages it is less than 1.5%. While some applications may not be able to tolerate this loss in bandwidth, in general, the losses are negligible in a real world application.

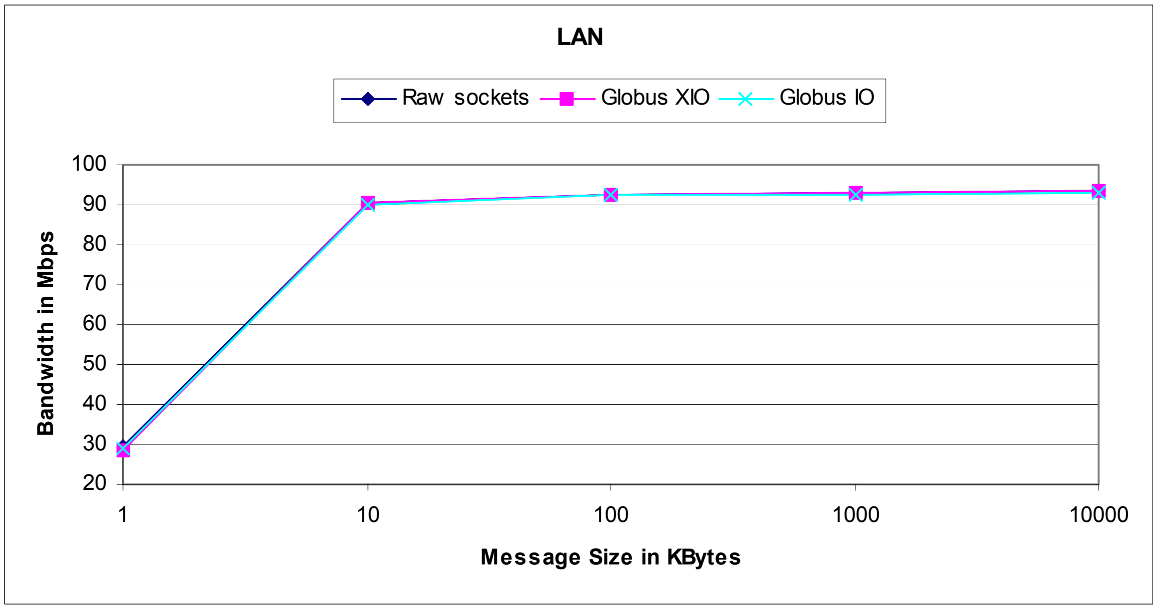

Figure 4: Comparison of throughput achieved over a gigabit link on a local area network.

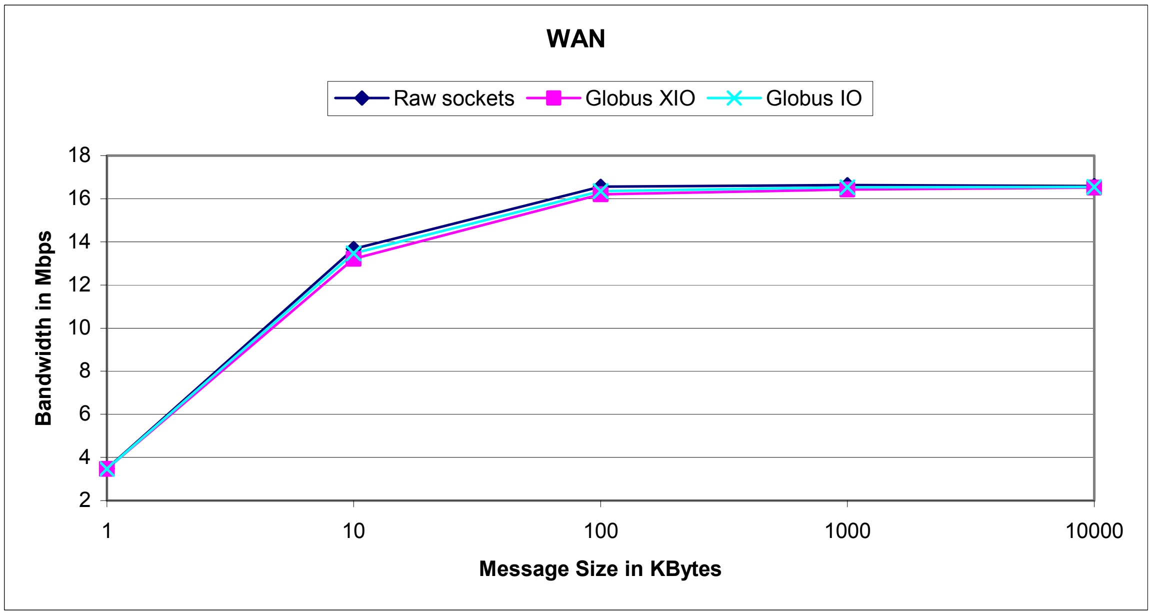

Figure 5: Comparison of throughput achieved over a wide area network where the bottleneck is an OC-12 link.

All the tests shown in the figures above were conducted with a single XIO driver loaded. However, XIO was designed to have a "stack" of multiple drivers and in some cases this is required to obtain equivalent functionality. Figure 6 shows a comparison of Globus IO using GSI authentication against Globus XIO using a stack consisting of a TCP driver and a GSI driver. The latency overhead is still minimal at less than 5% for small messages and less than 2% for larger messages

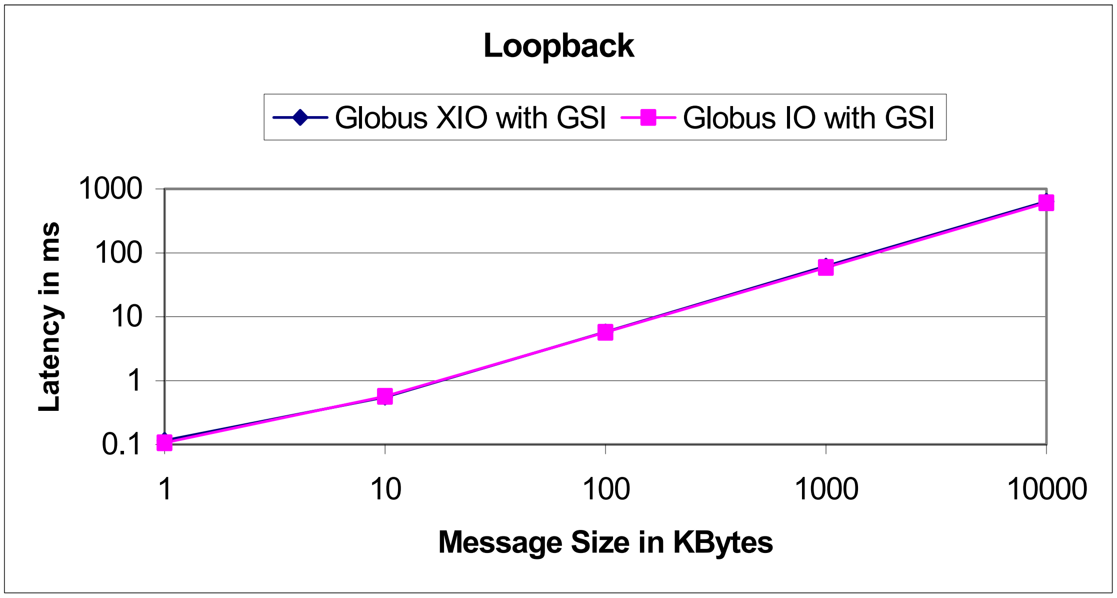

Figure 6: Comparison of latency for Globus XIO over GSI with Globus IO over GSI over the loopback interface on a local host.

## Summary

Globus XIO provides a framework that implements a simple Open/Close/Read/Write interface that is appropriate for most ***byte stream oriented*** IO applications. The framework user API presents a uniform API, semantics, and error handling to the application. Variations in programming models due to the different underlying protocols and APIs are isolated in drivers and drivers may be "stacked" to get composite functionality. Our early performance tests show that the overhead associated with this abstraction and the resulting benefits is minimal, particularly in most real world situations, generally less than 3%.

The primary benefits to be gained from using Globus XIO include:

***A uniform IO API that developers can learn and be comfortable with:***
A single, rational set of APIs, semantics, error responses, as well as problems and work arounds. Differences due to different network protocols or data access methods are isolated in the drivers making the application more stable and minimizing the programmer involvement with these differences.

***Ability to quickly adapt to different transport protocols:***
Applications written with XIO can switch from TCP to other protocols with minimal changes to the application.

***Ability to quickly adapt to different data access mechanisms:***
Files stored behind GridFTP servers, on HPSS or UniTree mass storage devices or any number of other vendor proprietary storage systems can all be accessed in uniform manner. This also provides providers of these access mechanisms a convenient vehicle for providing this access and immediately makes their access mechanisms available to any application utilizing XIO.

***Provides a convenient framework for research and development:***
Researchers in new protocols and data access mechanisms can utilize the XIO framework during their research and development cycles. The XIO framework provides many of the facilities required to successfully conduct their research, but outside their interest and expertise.

## Acknowledgment

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-ENG-38.

## References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, "Data Management and Transfer in High Performance Computational Grid Environments", Parallel Computing Journal, 2002.

[2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," Proceedings of the 18th IEEE Symposium on Mass Storage Systems, San Diego, California, April 17-20, 2001.

[3] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, S. Tuecke. "GridFTP Protocol Specification" *GGF* GridFTP Working Group Document, September 2002.

[4] W. Allcock, A. Chervenak, I. Foster, L. Pearlman, V. Welch, M. Wilde, "Globus Toolkit Support for Distributed Data-Intensive Science," Proceedings of International Conference on Computing in High Energy and Nuclear Physics, Beijing, China, September 2001.

[5] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," International Journal on Supercomputer Applications, 15(3), 2001.

[6] I. Foster, C. Kesselman, "The Grid: Blueprint for a Future Computing Infrastructure," Morgan Kaufmann Publishers, 1999
[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Manister, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC-2616 June 1999.
[8] http://www.sdsc.edu/hpss/
[9] D. Teaff, D. Watson, B. Coyne, "The Architecture of the High Performance Storage System (HPSS)," Proceedings of the Goddard Conference on Mass Storage & Technologies, College Park, Maryland, March 1995
[10] J. Postel, "Transmission Control Protocol," RFC-793, September 1981
[11] J. Postel, "User Datagram Protocol," RFC-768, August 1980
[12] http://www-unix.mcs.anl.gov/~bresnaha/
[13] M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control," RFC-2581, April 1999
[14] Dina Katabi, Mark Handley, and Charles Rohrs, "Internet Congestion Control for High Bandwidth-Delay Product Networks," ACM Sigcomm 2002, Pittsburgh, August, 2002
[15] Tom Kelly, "Scalable TCP: Improving Performance in HighSpeed Wide Area Networks," First International Workshop on Protocols for Fast Long Distance Networks, Geneva, February 2003
[16] Sally Floyd, "HighSpeed TCP for Large Congestion Windows", Internet-draft draft-floyd-tcp-highspeed-02.txt, Work in progress, February 2003.
[17] C. Jin, D. X. Wei, S. H. Low, "FAST TCP: motivation, architecture, algorithms, performance," IEEE Infocom, March 2004
[18] Dina Katabi, Mark Handley and Charlie Rohrs, "Congestion Control for High Bandwidth-Delay Product Networks", Proceedings on ACM Sigcomm 2002.
[19] W.T. Strayer, M.J. Lewis, R.E. Cline Jr., "XTP as Transport Protocol for Distributed Parallel Processing, Proceedings of the USENIX symposium on High-speed Networking, August 1994.
[20] E. He, J. Leigh, O. Yu, T.A. DeFanti, "Reliable Blast UDP: Predictable High Performance Bulk Data Transfer," Proceedings of IEEE Cluster Computing 2002.
[21] H. Sivakumar, R. L. Grossman, M. Mazzucco, Y. Pan, Q. Zhang, "Simple Available Bandwidth Utilization Library for High-Speed Wide Area Networks," Journal of Supercomputing, 2004 (to appear).
[22] http://www.indiana.edu/~anml/anmlresearch.html
[23] http://www.lac.uic.edu/~grossman/hpdt.htm
[24] Y. Gu, R. Grossman, "UDT: An Application Level Transport Protocol for Grid Computing" Second International Workshop on Protocols for Fast Long-Distance Networks, Feb 2004 (to appear).