# A Multicast Transport Driver for Globus XIO

Karl Jeacle and Jon Crowcroft
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue, Cambridge CB3 0FD, UK
{firstname.lastname}@cl.cam.ac.uk

## Abstract

*In recent years, much work has been done on attempting to scale multicast data transmission to hundreds or thousands of receivers. There are, however, many situations where an application might involve transmission to just ten or twenty sites. Today's Grid environments, for example, see High Energy Physicists carry out multi-gigabyte bulk data transfers to a handful of destinations.*

*In this project, we are investigating how TCP-XM, a modified version of TCP that supports multicast, can be integrated with Globus to to provide Grid users with a reliable multicast transport protocol.*

*Our approach has been to use Globus XIO – an eXtensible Input/Output library for Globus, that provides a POSIX-like API to swappable I/O implementations. We have wrapped TCP-XM in XIO to extend Globus to support multicast transmission.*

*This paper describes the implementation and operation of our Globus XIO multicast driver, reviews the TCP-XM protocol design, and provides some experimental results.*

## 1. Introduction

The Globus Toolkit is an open source software toolkit primarily developed by the Globus Alliance. It has become the de-facto standard for middleware used to build Grid services.

The toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability. It is packaged as a set of components that can be used either independently or together to develop applications.

At present, almost all bulk data transfer is carried out using the GridFTP protocol [16, 1]. This is based on the conventional FTP protocol, but includes some extra features to optimize bulk data transfer e.g. parallel data streams. Software that makes use of the protocol must support the Grid Security Infrastructure (GSI) [11] so that user authentication can take place using Grid certificates.

## 2. Globus XIO

Globus XIO is an eXtensible Input/Output library for the Globus Toolkit. It provides a POSIX-like API to swappable I/O implementations – essentially "I/O plugins" for Globus [3].

There are two main goals for Globus XIO:

1. Provide a single user API to all Grid I/O protocols. There are many different APIs for many different protocols. XIO should abstract this complexity for Grid developers.
2. Minimize the development time for creating new protocols. Writing with the XIO framework in mind allows the protocol designer to maximize the time spent on protocol code.

This approach is similar to the Streams [17] concept originally introduced in System V Unix. A stream is a full-duplex connection between a user process and a device. It consists of one or more connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. A key advantage of the Streams module approach is the ability to develop new code within the protocol stack without requiring changes to the kernel source.

Figure 1 shows the Globus XIO Architecture. The User API provides a familiar and convenient POSIX-like open/close/read/write interface to programmers. The Framework facilitates the XIO system itself, while the Driver Stack comprises one or more transform drivers over a single transport driver.

Transform drivers manipulate data buffers passed to them via the user API and the XIO framework. Transport drivers are always at the bottom of the stack and are solely responsible for sending data over the wire.

Example transform driver functionality includes tasks such as compression, logging and security. Transport drivers typically implement protocols such as TCP or UDP.

The configuration and order of drivers in the stack can be selected at runtime.

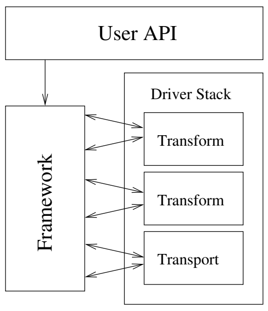

**Figure 1. Globus XIO Architecture**

Globus XIO provides an ideal mechanism for introducing new protocols to Grid users deploying Globus applications. We have built a modified version of TCP that supports multicast, and wrapped it using XIO to create a multicast transport driver for Globus.

## 3. TCP-XM

TCP-XM is a modified TCP engine that supports multicast, and runs in userspace over UDP. It forms the basis for our XIO transport driver.

This section provides some background rationale. A more detailed description of the protocol can be found in [13].

Today, applications use TCP for reliable unicast transfers. It is a mature and well-understood protocol. By modifying TCP to deliver data to more than one receiver at a time, and use multicast when available, an application can transparently send data reliably to multiple recipients. Using existing TCP mechanisms, the modified protocol ensures that data is delivered reliably. Multicast transmission is attempted for performance reasons, but fallback to unicast preserves backwards compatibility and reliability guarantees, and capitalizes on more than a decade of experience that TCP implementations enjoy.

Network protocols are typically implemented in the kernels of hosts and network devices. Any proposals that require modifications to these protocols imply changes to kernel code. This immediately restricts deployment opportunities. By limiting changes to code that runs in userspace on end stations, new protocols can be developed and tested on live networks.

Our approach is to implement a modified multicast TCP over UDP. User-space applications can freely send and receive UDP packets, so a small shim layer can be introduced to encapsulate and decapsulate the TCP-like engine's packets into UDP.

While there are performance implications by running in userspace, the instant deployment potential of a userspace implementation, coupled with the scalability of multicast, mean that any such limitations are more than acceptable.

Therefore, it is possible to build a new end-to-end protocol, and implement as a userspace library. Grid applications can avail of this via a Globus XIO driver.

The key advantage of this approach is that any Globus application can make use of this new protocol by simply pushing its XIO driver onto the stack, while non-Globus Grid applications can link against an independent userspace library.

No changes are required in the network (other than enabling IP multicast).

Because the protocol is not tightly coupled to the application, should it become adopted for widespread use, a native implementation can be built in the kernel to boost performance.

Our implementation of TCP-XM has been built as an extension to the lwIP TCP/IP stack [7].

## 4. Previous work

Reliable multicast has proved to be a difficult problem to solve, and over the last decade, much research has been carried out into how best to approach this problem [10, 14, 15, 5].

Moving TCP out of the kernel into userspace is not a new idea. A number of projects have done this in the past, either as a by-product of a larger project, or as an end in itself [9, 6, 8, 2, 7].

A number of transform and transport drivers have been built as part of the Globus XIO system. These include GSI and HTTP transform drivers, and TCP, UDP, File and SABUL transport drivers. An XIO implementation of GridFTP is in development[4].

There are no known existing Globus XIO multicast transport drivers.

## 5. XIO driver operation

Two important XIO data structures must be considered when implementing and making use of a transport driver:

1. Attribute – in order to set driver-specific parameters, a custom attribute structure can be used. For TCP-XM, the list of $n$ destination addresses is supplied in this way.
2. Handle – this is returned to the user once the XIO framework has all the information needed to open a

new connection. It is then used to reference the connection on all future I/O calls. An lwIP `netconn` structure will constitute the handle for multicast.

With these data structures in place, the transport driver is built by mapping XIO's POSIX open/close/read/write calls to the appropriate TCP-XM API calls.

Figure 2 shows how a developer can take advantage of XIO. First, the XIO stack is initialized. Next, drivers are pushed onto the stack (in this example, a TCP-XM transport driver). Any necessary driver attributes are created and with a stack in place, a handle for all subsequent I/O operations is created. The example finishes with six bytes of data being written to the network.

```
// init stack
globus_xio_stack_init(&stk, NULL);

// load drivers onto stack
globus_xio_driver_load("tcpxm", &tdrv);
globus_xio_stack_push_driver(stk,tdrv);

// init attributes
globus_xio_attr_init(&attr);
globus_xio_attr_cntl(attr, tdrv,
  GLOBUS_XIO_TCPXM_SET_REMOTE_HOSTS,
  hosts, numhosts);

// create handle
globus_xio_handle_create(&handle, stk);

// send data
globus_xio_open(handle, NULL, attr);
globus_xio_write(handle, "hello\n", 6,
1, &nbytes, NULL);
globus_xio_close(handle, NULL);
```

**Figure 2. Sample XIO User Code**

## 6. Internals

This section contains detailed information on the internals of the XIO driver implementation. Some knowledge of Globus and XIO development is required.

### 6.1. Data structures

The `globus_l_server_t` structure is used for XIO servers when TCP-XM is receiving data. `listen_conn` points to the initial `netconn` structure used on server listen, while `conn` points to the post-accept structure.

The `globus_l_attr_t` structure contains TCP-XM specific protocol information. If the driver is acting as a server, the `server` variable points to the relevant `globus_l_server_t` structure. It acts as a bridge for the driver between the creation of a server object and the assignment of a handle. This is because the `globus_l_xio_tcpxm_open()` call will use the value of the attribute server pointer to determine if the driver is operating as a client or a server. If the pointer is non-NULL, the user must have called `globus_xio_server_create`. Otherwise, the driver is a client, so a call to `netconn_connect()` will yield a new handle.

Two client specific variables in `globus_l_attr_t` are `hosts` and `numhosts`. These are passed into the driver when acting as a client. `hosts` is an array of destination hostnames, while `numhosts` is the number of hosts. The `GLOBUS_XIO_TCPXM_SET_REMOTE_HOSTS` command is used to set these attribute values.

`globus_l_attr_t` also contains `srcport` and `dstport` variables. These are used by both clients and servers to set the underlying UDP ports used by TCP-XM. The `GLOBUS_XIO_TCPXM_SET_UDP_PORTS` command is used to set these attribute values.

Finally, the `globus_l_handle_t` structure is very simple, containing a single pointer to the `netconn` structure that is ultimately used as the user handle for all I/O calls.

### 6.2. Function calls

[Note: the `glxt` prefix is used in this section as to abbreviate `globus_l_xio_tcpxm`.]

The XIO framework uses `glxt_activate()` to active the driver and `glxt_deactivate()` to later deactivate it. These are followed respectively by `glxt_init()` to tell XIO what functions are present in the driver, and later by `glxt_destroy()` to deallocate the driver.

Handles are created with `glxt_handle_init()` and destroyed with `glxt_handle_destroy()`.

`glxt_attr_init()` initialises attributes; copies are made with `glxt_attr_copy()` and then, when no longer of use to the driver, destroyed with `glxt_attr_destroy()`. User-specified commands such as `GLXT_SET_REMOTE_HOSTS` used to set the destination hosts and number of host, and `GLXT_SET_UDP_PORTS` to set the UDP ports used are made via `glxt_attr_cntl()`

If the driver is a server, `glxt_server_init()` initialises the `globus_l_server_t` structure, starts the lwIP/TCP-XM threads, and then binds and listens for new connections. Calling `glxt_server_destroy()` cleans up.

If a client, `glxt_connect()` is used internally to open a connection and create a new handle. `glxt_server_accept()` is the corresponding internal function on the server side; it blocks waiting for an incoming connection.

When a handle has been created, `glxt_open()` will open a new connection if a client, or block waiting if a server. Reads are then made via `glxt_read()`; writes are via `glxt_write()`, and `glxt_close()` is used to clean up.

## 7. One-to-many caveats

Two significant caveats with the current XIO approach have become apparent during the implementation of the multicast driver.

1. The XIO architecture assumes one-to-one connections. The XIO User API therefore requires modifications to better support one-to-many protocols. While minimal changes are required at the API, there may be more significant changes required within the XIO framework.

2. GSI is one-to-one. Most Globus application make use of GSI to authenticate with peers on connection setup. However, as it stands, GSI cannot be expected to authenticate $n$ peers. Some form of "GSI-M" that supports one-to-many authentication is required.

The first of the above caveats is a relatively minor difficulty. Workarounds are possible due to the flexible nature of the XIO attribute data structure.

The second caveat, however, is more serious. From a practical perspective, the multicast transport driver provides Globus applications with multicast data transfer capability to multiple destinations. But as it is not possible to push a one-to-one transform driver on top of a one-to-many transport driver, multicast support currently comes at the expense of security.

It is worth noting that security for many-to-many is often a problem because of late joiners and early leavers. But unlike many multicast protocols, TCP-XM has per-receiver state in the sender. And for bulk transfer from one to $n$ hosts, it is assumed that session and transport lifetimes are aligned. Because of this, while building a "GSI-M" transform driver may require changes to XIO, it is a far less onerous task than addressing many-to-many security [12].

## 8. Experimental results

We have carried out tests on both local and wide area networks. For local testing, a selection of departmental workstations were used. For wide area testing, shell accounts on machines at eScience Centres around the UK were obtained. These machines and locations were primarily chosen as they are representative of the target audience for our work i.e. physicists requiring bulk data transfer to a relatively small number of regional sites.

The UK eScience Centres are connected via the JANET academic network. Figure 3 illustrates the geographical connections.

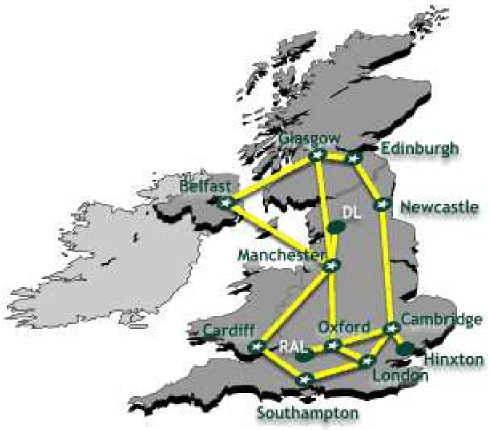

**Figure 3. The UK eScience Network**

Table 1 lists the hosts used. As the table shows, most of these sites have functional multicast connectivity. This is, in large part, due to the frequent use of the multicast-enabled AccessGrid video conferencing system.

| Site | Hostname | Mcast |
|---|---|---|
| Belfast | gridmon.cc.qub.ac.uk | No |
| Cambridge | mimiru.escience.cam.ac.uk | Yes |
| Cardiff | agents-comsc.grid.cf.ac.uk | Yes |
| Daresbury | ag-control-2.dl.ac.uk | Yes |
| Glasgow | cordelia.nesc.gla.ac.uk | No |
| Imperial | mariner.lesc.doc.ic.ac.uk | Yes |
| Manchester | vermont.mvc.mcc.ac.uk | Yes |
| Newcastle | accessgrid02.ncl.ac.uk | Yes |
| Oxford | esci1.oucs.ox.ac.uk | Yes |
| Rutherford | gridmon.rl.ac.uk | No |
| Southampton | beacon1.net.soton.ac.uk | Yes |
| UCL | sonic.cs.ucl.ac.uk | Yes |

**Table 1. UK eScience Testbed Hosts**

The specifications, network connectivity and operating systems used by the hosts varies widely from site to site. Some hosts are high speed machines connected close to the

WAN backbone. Others are smaller and older departmental machines with poorer connectivity.

Table 2 shows the average round-trip times and transfer rates seen between our test system and other sites around the network. The round-trip times vary in range from approximately 5 to 21 milliseconds. The transfer rates attainable on single TCP connections varied from as little as 1.5 Mb/s to over 50 Mb/s.

While this mixture may not be conducive to optimal headline results, it allows a truly representative set of protocol performance results for a live wide area network.

| Site | RTT (ms) | B/W (Mb/s) |
|---|---|---|
| Belfast | 18.6 | 16.0 |
| Cardiff | 13.5 | 22.4 |
| Daresbury | 21.3 | 28.1 |
| Glasgow | 16.2 | 33.9 |
| Imperial | 17.1 | 51.0 |
| Manchester | 9.9 | 34.5 |
| Newcastle | 11.8 | 1.5 |
| Oxford | 7.0 | 4.0 |
| Southampton | 8.8 | 39.3 |
| UCL | 4.9 | 42.1 |

**Table 2. WAN RTTs & Bandwidth**

Note that for simplicity, these experiments were carried out using a standalone test program linked against the TCP-XM userspace library, and not with a Globus application using the XIO driver. We would not expect to see any performance difference.

All tests compare TCP-XM (in userspace over UDP) with native kernel-based TCP. Our userspace implementation of TCP-XM means that it is at an immediate performance disadvantage to native TCP. Nevertheless, the results provide a useful indicator of the protocol's worth.

Figure 4 shows a comparison of the TCP and TCP-XM data transfer rate to $n$ hosts on a local departmental LAN. As would be expected, TCP's throughput declines as the host count increases. TCP-XM peaks at a much lower rate, but then consistently maintains this rate despite the introduction of more destination hosts.

Figure 5 shows the number of bytes being sent on the wire for the same transfer. Because TCP-XM is multicasting, it naturally scales. TCP is sending more and more data as the host count increases, so performance inevitably suffers. Note that the TCP-XM data is split in two: unicast bytes and multicast bytes. The unicast bytes are barely visible at the bottom of the graph. These account for connection setup, close, and retransmissions. The majority of the TCP-XM data transfer is composed of multicast bytes.

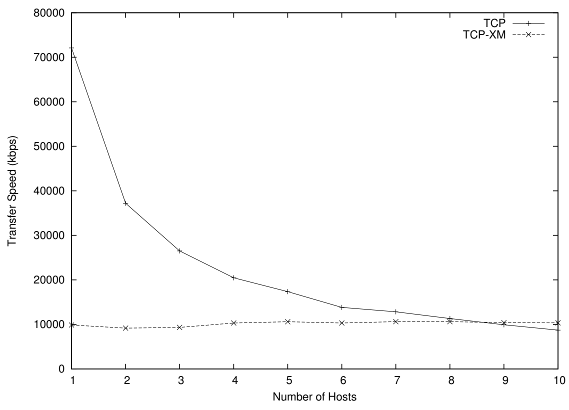

**Figure 4. LAN Speed**

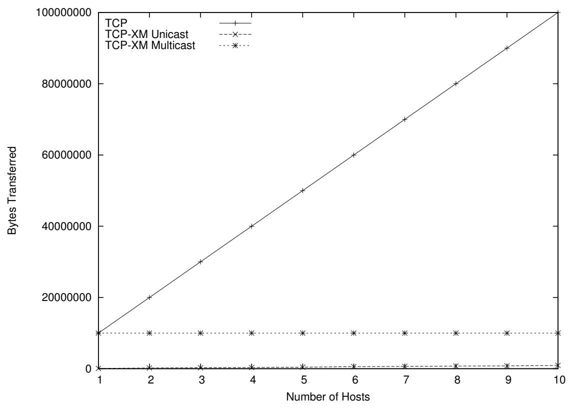

**Figure 5. LAN Efficiency**

Figure 6 shows how TCP and TCP-XM compare when a transfer to $n$ hosts takes place using a wide area network. As in the local area, TCP outperforms TCP-XM in raw speed, but less so than might be expected. The inherent bottlenecks present across the WAN, and the varied performance specification of receivers, prevent TCP from achieving the same strong results that are possible on a LAN. TCP-XM finds its optimum transfer rate quickly and again manages to maintain this rate across the WAN while making use of both unicast and multicast simultaneously.

Figure 7 once again shows TCP's inefficiencies as the host count increases. More interestingly, we can see more clearly how TCP-XM is combining unicast and multicast. Unlike the LAN test above, not all destinations are multicast capable, so TCP-XM cannot quickly switch to multicast after connection setup. The number of bytes unicast by TCP-XM is therefore much more significant. There is an obvious step up in unicast bytes sent each time TCP-XM encounters a destination host without multicast.

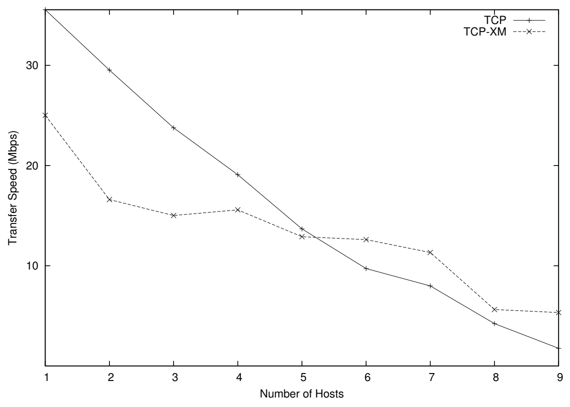

**Figure 6. WAN Speed**

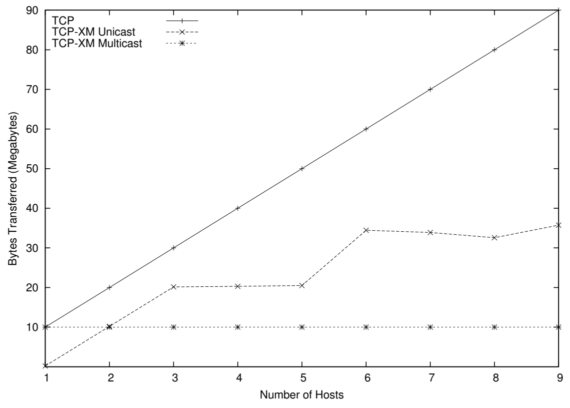

**Figure 7. WAN Efficiency**

With some optimisation in our implementation, we would expect considerable improvements in performance to be possible. And, of course, while kernel-based TCP will always have an advantage over a userspace implementation of TCP-XM encapsulated in UDP, the same would not be true of a kernel-based TCP-XM implementation.

In addition, the key benefit from TCP-XM and its multicast capability is not its raw data transfer rate, but its ability to reliably transfer large amounts of data in a far more efficient manner. In an appropriate application domain, this feature will outweigh native TCP's data transfer rate so much that even a limited userspace implementation can be more desirable than kernel-based TCP.

## 9. Conclusion

We have described the work to date on the TCP-XM protocol, and its implementation as a Globus XIO transport driver. By implementing this protocol in userspace above UDP, we are in a position to test the operation of the protocol in live networks, while delivering a reliable multicast transport mechanism to the Grid community.

## References

[1] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP Protocol Specification. GGF GridFTP Working Group Document, Sept. 2002.

[2] T. Braun, C. Diot, A. Hoglander, and V. Roca. An Experimental User Level Implementation of TCP. Technical report, INRIA RR-2650, Sept. 1995.

[3] J. Bresnahan. Globus XIO. www-unix.globus.org/developer/xio/, Dec. 2003.

[4] J. Bresnahan and B. Allcock. Globus XIO and GridFTP for Developers. In *Proceedings of GlobusWorld 2004*, San Francisco, Jan. 2004.

[5] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *SIGCOMM*, pages 56–67, 1998.

[6] T. Dunigan and F. Fowler. A TCP-over-UDP Test Harness. Technical report, Oak Ridge National Laboratory, ORNL/TM-2002/76, May 2002.

[7] A. Dunkels. Minimal TCP/IP implementation with proxy support. Technical report, Swedish Institute of Computer Science, SICS-T-2001/20-SE, Feb. 2001.

[8] A. Edwards and S. Muir. Experiences Implementing a High-Performance TCP in User-Space. Technical report, HP Laboratories Bristol, HPL-95-110, Sept. 1995.

[9] D. Ely, S. Savage, and D. Wetherall. Alpine: A User-Level Infrastructure for Network Protocol Development. In *Proceedings of USENIX USITS*, 2001.

[10] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, Dec. 1997.

[11] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *5th ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[12] T. Hardjono and G. Tsudik. IP Multicast Security: Issues and Directions. *Annales de Telecom*, 2000.

[13] K. Jeacle and J. Crowcroft. Reliable high-speed Grid data delivery using IP multicast. In *Proceedings of All Hands Meeting 2003*, Nottingham, UK, Sept. 2003.

[14] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[15] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven Layered Multicast. In *ACM SIGCOMM*, volume 26,4, pages 117–130, New York, Aug. 1996. ACM Press.

[16] T. G. Project. GridFTP: Universal Data Transfer for the Grid. Globus Project White Paper, Sept. 2000.

[17] D. M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.