XIOPerf : A Tool For Evaluating Network Protocols

John Bresnahan, Rajkumar Kettimuthu and Ian Foster
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439
Email: {bresnaha,kettimut,foster}@mcs.anl.gov

***Abstract*— The nature of Grid and distributed computing implies network communication between heterogeneous systems over a wide and ever-changing variety of network environments. Often times large amounts of data is stored in remote locations and must be transmitted in bulk. It is desirable to have the bulk data transfers be as fast as possible, however due to the dynamic networks involved it is often hard to predict what protocol will provide the fastest service for a given situation. In this paper we present XIOPerf, a network protocol testing and evaluation tool. XIOPerf is a command line program written on top of GlobusXIO with a simple and well defined interface to many different protocol implementations. XIOPerf was created to give users a way to quickly and easily experiment with an open ended set of protocols over real networks to determine which will best suit their needs. We present a brief study of the overhead introduced by XIOPerf and the performance of it when using a variety of protocols.**

I. Introduction

The nature of Grid [1]–[4] and Distributed computing implies an inherent need for communication. Resources that need to interact are distributed across many networks. An important type of communication in these environments is that of bulk data transfers. Often this means sending a file from one resource to another, but it can also mean streaming large data sets from a scientific instrument or that result from some computation. The important aspect of bulk transfers is that they involve very large data sets. Because of the size to be transferred, it is important that the protocols used are as efficient as possible. The quest to find the most efficient transfer protocols is a large and ongoing area of research.

Many protocols [5]–[10] have been developed and continue to evolve. Researchers strive to solve the problem of efficient bulk data transfer in better and faster ways. Their solutions are usually targeted at solving a specific part of the problem. Some protocols are designed for dedicated networks and are aimed at the greedy acquisition of bandwidth while others are designed to nicely coexist with the traffic of multiple users in a shared network. Sometimes the problem is looked at by throttle the send rate, and other times how fast the user consumes data is the dominating factor. However, no matter what the thought process the protocol designer goes through, the end user typically has one simple question. What protocol is best for my needs?

Answering that question on paper can be a difficult task. Many factors must be identified and considered. Every protocol has its strengths and could be a potential candidate. There is no single fastest protocol for every situation. The best choice most often depends on the environment in which the users application exists. The user must consider at a minimum the following parameters:

Network type: Is it a dedicated link, or does some quality of service guarantee that a portion of the bandwidth is dedicated to the user?

Network activity: Is the network typically over-utilized or under-utilized? How congested is it? How much packet loss is expected? Does the protocol need to be fair to other users?

Endpoints: Are the endpoint machines fast enough to keep up with the network, or are they the bottle neck in the pipeline?

Application: How does the application consume the data? Does it write to disk? If so what is the disk speed? Are many memory copies made? How will the applications consumption of data affect data sending rates or data packet loss.

Unfortunately determining these categories is not an easy thing to do. Many of the factors are not known, and some, like network activity, are always changing. Additionally, there are factors too subtle to even categorize that can have dramatic results on the achieved performance. Even if a valid conclusion can be drawn on paper it may not actually be the best solution in practice due to things like errors in (or liberties taken with) the actual implementations of the protocol stack.

The easiest, and most practical way for a user to determine what protocol is best is to actually try them. If the user were able to experiment with all of the bulk data transfer protocols available they could empirically determine which works the best for their environment. This, of course, brings up another problem, how can a user tests out every protocol proposed in research? Clearly this is not realistic. However by defining standard interfaces for test applications to use and creating a framework for assisting protocol implementation it may be feasible to try a large number of them.

If the effort required to transform a proof of concept implementation or a proprietary reference implementation into a standard interface were minimized a situation could be created where protocol authors were willing to do so. In cases where they did not, application developers interested in experimenting with the protocols may be motivated to spend the small amount of effort required to morph the implementation into the standard framework. Once the implementation is accessible via the standard interface a common testing tool can perform fair and accurate evaluation of protocols over real world networks. This is exactly what XIOPerf strives to accomplish.

This paper introduces XIOPerf and proposes that it be this ubiquitous network performance testing tool. XIOPerf is a command line tool that presents the user with a familiar interface and set of options for performing bulk data transfers over a network. The tool measures the performance characteristics of a transfer and reports them to the user. XIOPerf is written on top of GlobusXIO [**?**] so it has all of the dynamically loadable transport driver functionality allowing it to address the concerns we have outlined.

The remainder of this paper is organized in the following way. We first present related work. We then introduce the reader to the XIOPerf program, its architecture, and how to use its basic functionality. We next describe the wrapblock functionality which makes driver creation much easier. Wrapblock has been added to GlobusXIO as part of this work. Finally we evaluate the testing tool by measuring the amount of overhead the abstraction layer adds and the performance achieved using XIOPerf with real protocols over real networks.

II. Related Work

A. IPerf

Presently IPerf is the defacto standard for measuring and optimizing bulk transfer performance. IPerf is a command line tool written in C++ by NLANR [11]. IPerf allows a user to send messages via either TCP [12] or UDP [**?**]. A user can choose to either send a file and have the receiver write the file to disk, or remove the disk from the equation and only send data to and from memory in its process space. In addition to measuring bandwidth, IPerf also measures jitter. This is an important aspect that XIOPerf does not yet address. IPerf has proved to be an exceptionally useful tool. A typical IPerf session involves a user running IPerf as a server on one side of the network and as a client which connects to that server on the other. The client then sends either a set amount of bytes, or a sends for a set amount of time and completes by reporting the throughput and other stats to the user. The main difference between IPerf and XIOPerf is that while IPerf is limited to TCP and UDP, XIOPerf is written on a framework that allows the user to plug in arbitrary protocol implementations.

```
-i  #   sets the reporting interval.  Every # seconds a throughput
        report will be written to stdout.
-bs #   set the length of the read/write buffer and thereby controls
        the amount of data to post at one time.
-w #    A TCP specific option.  This sets the TCP window size and
        maintains interface compatibility with IPerf
-n #    Sets the number of bytes to transfer to #.
-F <path>   filename for input if sending, and output if receiving.
        By default transfers are memory to memory.
-S  Be a sender
-R  Be a receiver
-P #    The number of parallel transfer to conduct at 1 time.
-s  run in server mode
-c <contact>run in client mode and connect to the given contact string.
-D <name>   The name of the next driver to add to the stack.
```

Fig. 1. Some command line options to XIOPerf.

B. TTCP

Test TCP (TTCP) is a predecessor of IPerf. They offer much of the same functionality. Both measure the performance of TCP over a network. TTCP was originally created for the BSD operating system, since that time some ports of it have been made to be more user friendly and to run on more operating systems, like Microsoft Windows.

C. Others

There is a bunch of other network measurement and network testing tools [**?**]. While these tools are often used in conjunction with iperf, they are not aimed at end-to-end bandwidth testing. One exception to that is pathrate. But none of these tools provide a feature to test the performance of different protocols on a network.

III. XIOPerf

XIOPerf presents a similar interface to that of IPerf. It is a command line tool with many of the same options and behaviors. Just as with IPerf, XIOPerf runs as a server on one side of the network and a client on the other. The client connects to the server and a bulk data transfer occurs according to the parameters given. The user is given many runtime options including the amount of data to transfer, which side sends and which receives, buffer sizes to use, whether or not to perform disk IO, etc. Some of the options can be found in figure 1.

The option that is most important, and that makes XIOPerf unique among applications of its kind is -D. This allows the user to specify what protocol will drive the bulk transfer. The protocol must be implemented as a GlobusXIO driver (how this is done will be discussed later). For example if the user wishes to measure the achieved bandwidth of the network using TCP they would run the XIOPerf on the server:

```
 % globus-xioperf -s -D tcp
---------------------------------------------------------------
server listening on: localhost:50002
---------------------------------------------------------------
```

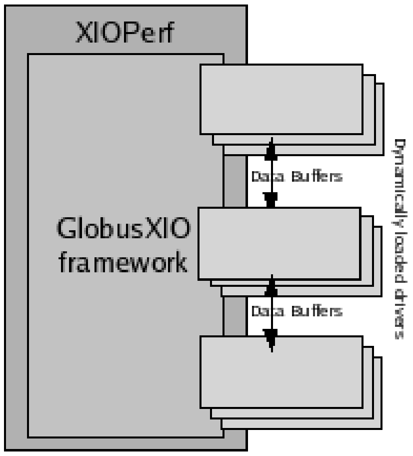

Fig. 2. XIOPerf Architecture

and on the client:

```
 % globus-xioperf -D tcp -c localhost:50002
----------------------------------------------------------------
Connection established
----------------------------------------------------------------
Time exceeded.  Terminating.
        Time:          00:10.0009
        Bytes sent:    5474.50 M
        Write BW:      4379.19 m/s
        Time:          00:10.0000
```

To change the protocol used by XIOPerf to UDT [7], the user need only change the string they run with from 'tcp' to 'udt'. By specifying the -D option multiple times the user can build a protocol stack over which the bulk transfer will occur. For example, if the user wanted to measure how a TCP transfer performed with GSI security they would run the above command with an addition '-D gsi' argument appended to the command line.

IV. GlobusXIO

XIOPerf achieves the multi-protocol abstraction because it is built on top of GlobusXIO [?]. GlobusXIO is the Extensible Input Output component of the Globus Toolkit(tm) [**?**]. It is a framework that presents a single standard open/close/read/write type interface to many different protocol implementations. The protocol implementations are called drivers. Creation of drivers is discussed later in this paper. Once created, a drivers can be dynamically loaded and stacked by any GlobusXIO application. XIOPerf takes full advantage of this feature.

XIOPerf is a fairly simple application and gets most of its power from the GlobusXIO library. The diagram in figure 2 illustrates how XIOPerf uses GlobusXIO. XIOPerf is linked against GlobusXIO library. It uses the GlobusXIO API for all of its IO needs. GlobusXIO then takes care of finding and loading the specified protocol drivers, establishing the connections using that driver stack, and passing the data buffers down the chain of drivers.

Since the drivers are loaded dynamically and adhere to a standard interface, they do not have to be linked or compiled into the application. New drivers can be added to the `LD_LIBRARY_PATH` at any time after the binary XIOPerf installation has taken place. This is an important aspect which allows for growth as new protocols are developed by researchers.

V. Globus XIO Driver Creation

The success of XIOPerf is hinged on the existence of GlobusXIO drivers for many bulk transfer protocols. We propose to achieve this in a couple of ways. The first is creating drivers within the GlobusXIO community. There are a limited set of resources within the community so we cannot expect to create all drivers in this way. The second way we propose to scale up on driver production is to make it very easy to write wrapper code that can glue a prototype or reference implementation into a driver interface. This will allow protocols that only exist in some unstable proof of concept implementation and those that do have robust reference implementations to be used.

In pursuit of this goal part we introduce the wrapblock feature to GlobusXIO. This is a simple extension to the original GlobusXIO driver interface that allows for much easier creation of drivers. The stock GlobusXIO driver interface in written on a asynchronous model. While this is the most scalable and efficient model it is also the most difficult to code against. The wrapblock functionality uses thread pooling and event callback techniques to transform the asynchronous interface to a blocking interface, thus presenting the implementer with a much easier task. Some drivers such as TCP, UDP, HTTP, File, Mode E [**?**], UDT, Telnet, Queuing, Ordering, GSI and Multicast Transport [**?**] have been developed for Globus XIO.

As an example we look at the `udt_ref` driver. This driver uses the wrapblock feature to glue the standard UDT [13] reference implementation into a GlobusXIO driver. We were able to accomplish creating this driver and using it in the XIOPerf program in less than one day of work. To illustrate the ease in creation we look at the code required to implement write functionality in figure 3.

As is shown, the implementation requires a simple pass through call to the UDT library. The actual number of bytes written is passed back to GlobusXIO by reference in the nbytes parameter. The data structure `xio_l_udt_ref_handle_t` is created in the open interface call and is passed to all other interface calls as a generic void * memory pointer. This allows the developer to maintain connection state across operations. Similar code is written to handle reading data. In the open and

```
static
globus_result_t
globus_l_xio_udt_ref_write(
    void *                          driver_specific_handle,
    const globus_xio_iovec_t *      iovec,
    int                             iovec_count,
    globus_size_t *                 nbytes)
{
    globus_result_t                 result;
    xio_l_udt_ref_handle_t *        handle;
    GlobusXIOName(globus_l_xio_udt_ref_write);

    handle = (xio_l_udt_ref_handle_t *) driver_specific_handle;

    *nbytes = (globus_size_t) UDT::send(
        handle->sock, (char*)iovec[0].iov_base, iovec[0].iov_len, 0);
    if(*nbytes < 0)
    {
        result = GlobusXIOUdtError("UDT::send failed");
        goto error;
    }

    return GLOBUS_SUCCESS;
error:
    return result;
}
```

Fig. 3. A sample wrapblock write interface implementation for UDT.

close interface function the developer initializes and cleans up resources as would be expected. The code inside the driver looks very much like a simple program using the third party API. There is little GlobusXIO specific code beyond the interface function signatures. Additionally, there are driver specific hooks that allow the user to directly interact with the driver in order to provide it with optimization parameters. This is handled via cntl functions that look much like the standard UNIX ioctl(). Further discussion on this can be found at http://www.globus.org/toolkit/docs/4.0/common/xio/

VI. Experiments

The first set of experiments shows the overhead introduced by the GlobusXIO framework. Since we are adding an abstraction layer between the application and the code that does the actual work of shipping bits there will necessarily be some overhead. In our first set of results we recorded the time before we registered an event in the application space, and then again when the event made its way to the drivers interface function. This is the exact interval from the time the user requests an operation to the time it can begin to be delivered by the protocol implementation. This measurement is referred to as “down the stack”. We also measure the time ’up the stack’. This is the interval from immediately before the driver signals it has completed its work to the time the application is notified of the completion. These two measurements show the exact overhead of the GlobusXIO abstraction. We took the average of many hundreds of read and write operations and averaged them together both separately and together. We ran the experiment on a UC TeraGrid [14] node with Dual 1.5Ghz Itanium processors and the Linux 2.4.21 kernel. Table 1 shows the results of the average overhead per operation in milliseconds.

The results show that much more overhead is introduced up the stack than down the stack. The code path for each

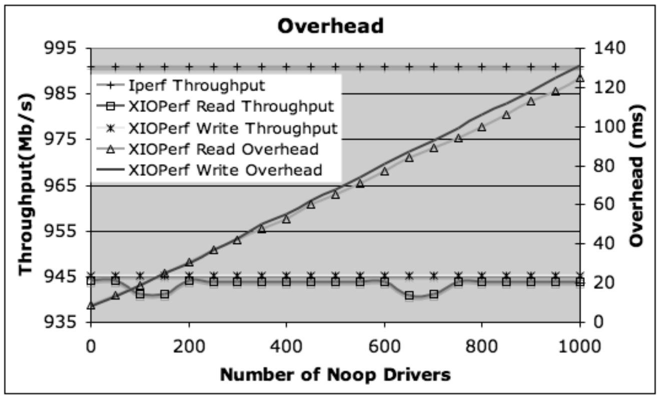

Fig. 4. Measurement of overhead with noop drivers

is entirely different and therefore it is appropriate that we see different times. However, the difference between the two is somewhat dramatic. This is due to some convenience functionality in GlobusXIO. Internally GlobusXIO does some event synchronization on the way up the stack to ensure that the user of the library receives events in a sane manner. For example, there is a barrier between all data operation events and close events. This gives users a guarantee that when the close event is delivered they will receive no other events and thus can safely cleanup their resources. Without this users would have to reference count their events or track them in some other way that would unnecessarily complicate the application.

$Operation$	Up	$Down$	$Both$
Read	0.014	0.001	0.007
Write	0.015	0.001	0.008
Both	0.015	0.001	0.008

TABLE I
Overhead Times

To show how the overhead scaled in the presence of many drivers we created the noop driver. This driver only forwards requests down the stack and completion notifications up the stack. It is intended to sit in the middle and do nothing but add the overhead required for each additional driver. Many of these were added to the stack to show how additional drivers affect the performance. We measured the average overhead times up and down the stack, as we did above, but with an increasing number of noop drivers. On the bottom of the stack was the TCP driver which did a bulk data transfer across the local gigabit network of the UC TeraGrid. To show how this overhead affects performance we also measured the achieved throughput of XIOPerf and IPerf. Figure 4 shows this.

Overhead increased linearly with the addition of more noop drivers. This was expected. On average each driver adds

.125 microseconds of overhead on our test system. Both reads and writes add roughly the same amount of overhead. The achieved bandwidth was unaffected by the introduced overhead. The achieved throughput is steadily maintained at around 950 Megabits per second. Since the ping time between the nodes in the transfer is approximately 0.372 milliseconds, which is significantly higher than all of the latency added by GlobusXIO between serial IO operations, the delay of buffer delivery that is added does not affect the throughput.

IPerf achieves a steady 990 Mb/s, which is better than XIOPerf. The performance differences are likely due to the asynchronous implementation of the TCP driver. GlobusXIO and IPerf are designed on different IO models. IPerf is written with blocking socket code and threads. GlobusXIO is designed for highly parallel and scalable systems so it is on an asynchronous model. Because of this the performance of GlobusXIO with applications displaying high levels of concurrent IO should be very steadily distributed across streams and ultimately achieve the most scalable performance.

VII. Driver experiments

In an effort to show the need for XIOPerf we familiarize the reader with various distinctly optimized and commonly cited bulk transfer protocols. We look at TCP, UDT, and GridFTP (Mode E). To show the effectiveness of XIOPerf when evaluating protocols we have compared the performance inside of XIOPerf against the reference implementation for each protocol. Bulk transfers of increasing sizes were run over the UC TeraGrid LAN and on the wide area network between UC TeraGrid nodes and SDSC TeraGrid nodes. No disk IO was done in this study.

A. TCP

TCP [12], [15], [16] is a well known and ubiquitous protocol. We will therefore only touch on a few aspects of it here. TCP is targeted at the Internet at large. It has done an impressive job scaling as the Internet has gone through a boom in terms of users as well as transfer rates. It provides reliable and fair access to many users of a network. For its targeted audience it is a very good protocol. However, for lambda networks [17] and LFNs [18] it is not ideal.

TCP is window based. A window size constitutes a certain number of bytes that can be in flight at a given time. In flight refers to the bytes that the receiver has not yet acknowledged as having received. Various algorithms which will not be discussed here determine how and when the receiver acknowledges bytes received. The size of this window and the latency on the network greatly affects the rate at which data can flow. The ideal size of the window is calculated by the bandwidth delay product:

$bwdp = rtt * bw$

Where rtt is the round trip time, and bw is the available bandwidth. The rtt is used because it takes into account the time for a byte of payload to move from the sender to the receiver and the time it takes for the acknowledgment to move from the receiver to the sender. BW reflects the number of bytes that can be sent in a given time slice.

A user of TCP can select the maximum window size, however TCP scales up and down the percentage of the window that it will use at any particular time. The algorithms that TCP uses to decide on the current window size are well documented elsewhere. Here we only want to point out two aspects that greatly affect its effectiveness in LFNs and lambda networks.

The first is TCP slow start. TCP starts with a very small window size and exponentially increases it as it receives acknowledgments. Slow Start may sound like a bit of a misnomer when the growth is exponential but it really does make for a slow start. While linear growth would be much worse, this still makes for many round trips before the window can be fully open. In LFNs where the optimal window is large and the time it takes to receive acknowledgments is large, it may take the entire lifetime of the transfer or more to increase to an optimal window size. This is a performance killer.

The next issue in TCP is how it handles congestion events. Since TCP is designed to be multi-stream friendly if it detects that one stream is moving too fast and thus causing congestion, it will decrease this streams window size. The problem enters with how drastically the window size is decreased and how slowly it is rebuilt. When TCP detects a congestion event it divides the window size by two. However, it will only increase the window by the size of one segment, which is typically around 1500 bytes, per acknowledgment received. When the ideal window size is many megabytes this obviously makes a dropped packet very costly in terms of performance. This algorithm is commonly referred to as Additive Increase Multiplicative Decrease (AIMD).

The GlobusXIO TCP driver is written on top of the standard BSD socket interface available on all UNIX platforms. The actual protocol is implemented inside of the kernel.

B. UDT

UDT is a UDP based reliable protocol and like TCP it is targeted at shared networks. However, UDTs main audience is under used networks with a small number of UDT streams. It is designed to be able to coexist fairly with TCP streams, but also be able to achieve high throughput faster and have less of a penalty for a congestion event. Beyond being a protocol UDT is a framework that allows users to plug in their own congestion control algorithms.

Like TCP, UDT uses a window and an AIMD strategy for congestion control. The difference is that UDT determines the factors to use by a much more sophisticated strategy. To determine these factors UDT must estimate the available bandwidth on the link. It does this by sending two probe packets for every sixteen data packets. The probe packets are sent sequentially without any regard for rate limiting. Based on the time apart that they arrive a bandwidth estimate is made. This bandwidth estimate is used to calculated the additive increase factor.

The decrease factor is much less severe than TCP. UDT only decreases the window size when a NAK is received. This occurs less often then a TCP congestion event. Additionally, when the NAK is received, instead of cutting the window in half, the window is set to 8/9ths of its previous size. This is still a multiplicative decrease, and a back off of sending rate, but it is a much less severe penalty. The increase factor is determined by looking at the greater of the following:

- $10^{\lceil(log10((B-C)*MTU))\rceil} * \beta/MTU$
- $1/MTU$

Where B is the bandwidth estimate, C is the current sending rate. β is a constant value of 0.0000015. MTU is the maximum transmission unit for the network. The UDT driver is implemented using the previously discussed wrapblock feature newly added to GlobusXIO. We have wrapped the reference implementation provided by Grossman et al. [13] into a GlobusXIO driver.

C. GridFTP

GridFTP [19] is a protocol for file transfers. GridFTP is commonly misunderstood to be a single protocol for bulk transfer. This is not exactly true. It is not itself a single protocol, but rather is a collection of protocols. Much like the standard well know FTP protocol documented in RFC959, it has two channels. There is a control channel specification that allows users to execute shell-like commands such as mkdir, rename, delete, etc, and most importantly request files for transfer. The control channel protocol is specified on top of telnet and TCP. This protocol is not optimized for efficiency and is thus not intended to be fast. It is simply a reliable means of establishing data channels pathways. The data channel is the pathway of which the bulk data transfer flows.

The protocol used for the data channel is open ended. The authors of RFC959 had much foresight in realizing that different users may prefer a different means of transferring the bulk data. To allow for this they defined the data channel protocol to be a Mode and then created a control channel command that allows the client to select what mode they wish to use. This gives the user the ability to decide what bulk protocol they will use to send their file at transfer time.

The commonly used mode in GridFTP is called ModeE. Since Mode E is the bulk data transfer protocol in GridFTP it is therefore what we will be exploring. Mode E is a parallel TCP protocol. A set of TCP connection are established and the data is transferred equally across them. As the transfer is in progress TCP streams can be added or removed. When the transfer completes the establish TCP connections can be cached for use with a later transfer.

Categorically parallel TCP protocols all have the same advantages and disadvantages. The difference between them are largely based on implementation. GridFTP is different in that it does not stop at using parallel streams for endpoint to endpoint performance gain. It has extended the concept parallel TCP streams so that many different endpoints can participate in a single coordinated transfer in a M to N fashion. This has the obvious advantage of summing the collective bandwidth available at all endpoint pairs.

The general advantage to using multiple streams is that it proportionally reduces the disadvantages associated with TCP by the number of parallel streams. For clarity we will refer to the number of parallel streams as P. The bandwidth delay product for each stream in a transfer is bw*latency/P which is 1/P smaller than the optimal window size of a single TCP stream. Therefor each individual window can be fully opened faster. And since all streams are used in parallel the slow start of TCP is theoretically reduced by 1/P. Similarly, the penalty associated with a congestion event is also reduced by 1/P. As stated above, TCP is most efficient when a properly set window is fully open. When a congestion event occurs window is closed and slowly rebuilt. This constitutes a large penalty in a bulk transfer. However if many streams are used a single congestion event only affects one stream, and therefore affects only 1/P of the overall transfer.

The target network of Mode E is similar to that of UDT. It is aimed at underused networks. If too many parallel streams are used the protocol becomes unfair to other streams and can potentially choke itself by causing too may congestion events.

The Mode E driver was written using the native GlobusXIO driver API. This is the best solution for creating scalable and efficient protocol drivers. GlobusXIO provides an assistance API for creating drivers in this way.

D. Results

The results of the performance evaluation are shown in figures 5 through 10. We measured the achieved throughput of each protocol with increasing bulk transfer lengths. The important difference between the two networks over which we tested is the latency between endpoints. On the LAN study the latency was about 0.372 microseconds and the WAN was about 58.140 milliseconds. The networks were not congested so some of the aspects of each protocol were not tested.

Along with the throughput results, each graph also shows the percentage by which the reference implementation outperformed the XIOPerf implementation. In all cases this

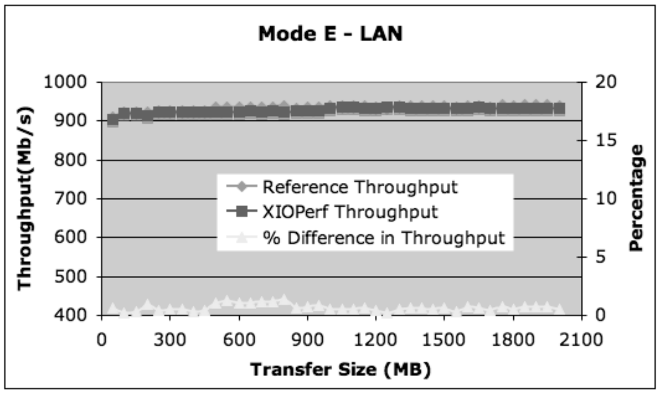

Fig. 5. Comparison of Mode E protocol's performance on a local area network as observed with XIOPerf and the actual run of the Reference implementation of the protocol.

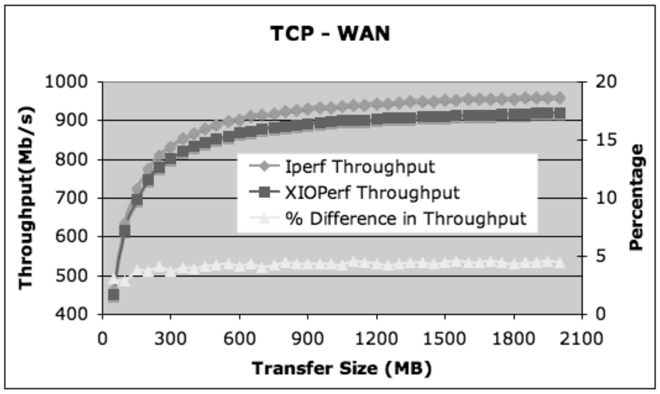

Fig. 8. Comparison of TCP protocol's performance on a wide area network as observed with XIOPerf and Iperf.

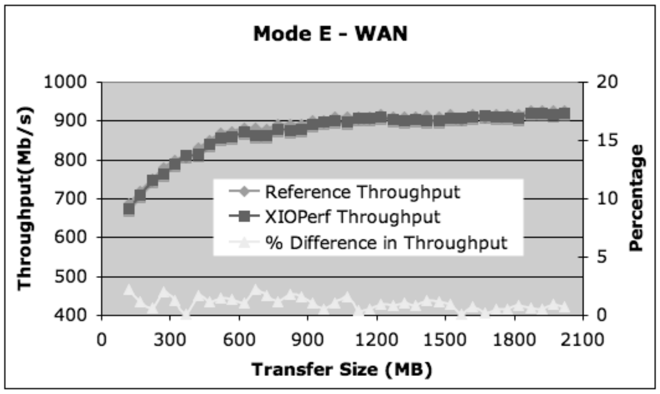

Fig. 6. Comparison of Mode E protocol's performance on a wide area network as observed with XIOPerf and the actual run of the Reference implementation of the protocol.

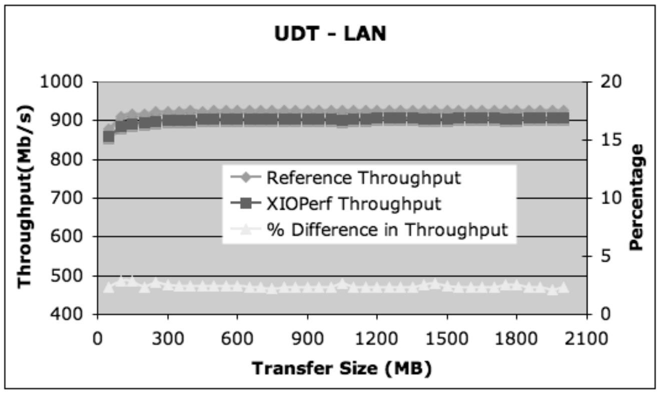

Fig. 9. Comparison of UDT protocol's performance on a local area network as observed with XIOPerf and the actual run of the reference implementation of the protocol.

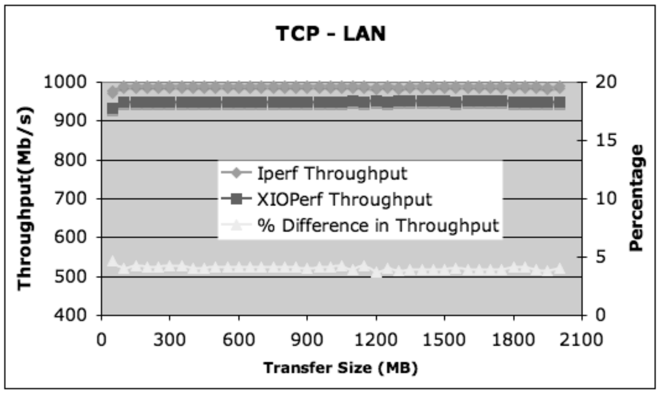

Fig. 7. Comparison of TCP protocol's performance on a local area network as observed with XIOPerf and Iperf.

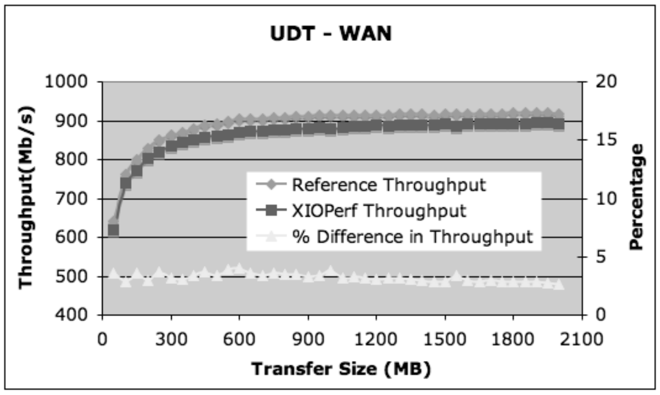

Fig. 10. Comparison of UDT protocol's performance on a wide area network as observed with XIOPerf and the actual run of the reference implementation of the protocol.

percentage is less than five percent, which means that XIOPerf was always within 95% of achieved throughput. As we stated above, each of the drivers studied here were written on different models. The variance in percent throughput difference is accounted for the differences in the different implementations. Mode E has the lowest percentage. It is written using the native GlobusXIO driver library therefore it is used in the most efficient way possible.

TCP has the highest percentage difference in throughput. We believe this to be an anomaly due to how efficiently IPerf uses the kernel's TCP stack. While we hope to increase the throughput of the GlobusXIO TCP driver, IPerf is strictly a performance measurement tool and XIOPerf is much closer to a real application. Therefore IPerf has an advantage in achieving very high throughput, but XIOPerf is likely to be closer to what an actually application will achieve, especially if the application uses GlobusXIO,

In future work we hope to decrease the performance gaps substantially, especially in the case of TCP. However, even with this performance gap XIOPerf is an useful tool. The purpose of it is to determine which protocol is best to use. Since the achieved throughput inside of XIOPerf is very close to that of the reference implementation all protocols are on level ground and can be fairly compared. As part of the comparison the way the driver was created and the results shown here can also be taken into account.

ACKNOWLEDGMENTS

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.

LICENSE

REFERENCES

[1] I. Foster, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations,” in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*. Springer-Verlag, 2001, pp. 1–4.

[2] I. Foster and C. Kesselman, “Computational Grids,” in *Selected Papers and Invited Talks from the 4th International Conference on Vector and Parallel Processing*. Springer-Verlag, 2001, pp. 3–37.

[3] ——, “Computational Grids: On-Demand Computing in Science and Engineering,” *Computers in Physics*, vol. 12, no. 2, p. 109, 1998.

[4] ——, “Computational grids,” pp. 15–51, 1999.

[5] D. Katabi, M. Handley, and C. Rohrs, “Congestion control for high bandwidth-delay product networks,” in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, 2002, pp. 89–102.

[6] W. T. Strayer, M. Lewis, and R. E. Cline, Jr., “XTP as a transport protocol for distributed parallel processing,” in *Proceedings of the USENIX Symposium on High-Speed Networking, oakland, CA, August 1994*, pp. 91–101. [Online]. Available: citeseer.nj.nec.com/strayer94xtp.html

[7] Y. Gu and R. Grossman, “UDT (UDP based Data Transfer Protocol): An Application Level Transport Protocol for Grid Computing,” in *Second International Workshop on Protocols for Fast Long-Distance Networks, Argonne, IL, 2004*.

[8] H. Sivakumar, R. L. Grossman, M. Mazzucco, Y. Pan, and Q. Zhang, “Simple available bandwidth utilization library for high-speed wide area networks,” *Journal of Supercomputing*, 2004.

[9] D. Clark, M. Lambert, and L. Zhang, “NETBLT: A Bulk Data Transfer Protocol. IETF, RFC 998,” March 1987.

[10] E. He, J. Leigh, O. Yu, and T. A. DeFanti, “Reliable blast udp: Predictable high performance bulk data transfer,” in *CLUSTER ’02: Proceedings of the IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2002, p. 317.

[11] “Iperf web page,” `http://dast.nlanr.net/Projects/Iperf/`.

[12] J. Postel, “RFC 793: Transmission Control Protocol,” 1981.

[13] R. L. Grossman, Y. Gu, X. Hong, A. Antony, J. Blom, F. Dijkstra, and C. de Laat, “Teraflows over gigabit wans with udt,” *Future Gener. Comput. Syst.*, vol. 21, no. 4, pp. 501–513, 2005.

[14] “Teragrid web page,” `http://www.teragrid.org`.

[15] V. Jacobson, R. Braden, and D. Borman, “RFC 1323: TCP Extensions for High Performance,” 1992.

[16] M. Allman, V. Paxson, and W. Stevens, “RFC 2581: TCP Congestion Control,” 1999.

[17] X. R. Wu, “Evaluation of rate-based transport protocols for lambda-grids.” [Online]. Available: citeseer.ist.psu.edu/698560.html

[18] K. Kumazoe, Y. Hori, M. Tsuru, and Y. Oie, “Transport protocols for fast long-distance networks: Comparison of their performances in jgn,” *saint-w*, vol. 00, p. 645, 2004.

[19] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, “The globus striped gridftp framework and server,” in *SC ’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 54.

APPENDIX

GLOBUS_XIO_EXAMPLE.C

```
#include "globus_xio.h"

int
main(
    int                             argc,
    char *                          argv[])
{
    globus_result_t                 res;
    char *                          driver_name;
    globus_xio_driver_t             driver;
    globus_xio_stack_t              stack;
    globus_xio_handle_t             handle;
    globus_size_t                   nbytes;
    char *                          contact_string = NULL;
    char                            buf[256];

    contact_string = argv[1];
    driver_name = argv[2];

    globus_module_activate(GLOBUS_XIO_MODULE);
    res = globus_xio_driver_load(
            driver_name,
            &driver);
    assert(res == GLOBUS_SUCCESS);

    res = globus_xio_stack_init(&stack, NULL);
    assert(res == GLOBUS_SUCCESS);
    res = globus_xio_stack_push_driver(stack, driver);
    assert(res == GLOBUS_SUCCESS);

    res = globus_xio_handle_create(&handle, stack);
    assert(res == GLOBUS_SUCCESS);

    res = globus_xio_open(handle, contact_string, NULL);
    assert(res == GLOBUS_SUCCESS);

    do
    {
        res = globus_xio_read(
            handle, buf, sizeof(buf) - 1, 1, &nbytes, NULL);
        if(nbytes > 0)
        {
            buf[nbytes] = ’\0’;
            fprintf(stderr, "%s", buf);
        }
    } while(res == GLOBUS_SUCCESS);

    globus_xio_close(handle, NULL);

    globus_module_deactivate(GLOBUS_XIO_MODULE);

    return 0;
}
```

Fig. 11. A example GlobusXIO user program