A Case for UDP Offload Engines in LambdaGrids

V. Vishwanath[‡] P. Balaji[§] W. Feng[¶] J. Leigh[‡] D. K. Panda[§]

[‡]Electronic Visualization Lab., Univ. of Illinois, Chicago {venkat, spiff}@evl.uic.edu

[§]Network Based Computing Lab., Ohio State University {balaji, panda}@cse.ohio-state.edu

[¶]Dept. of Computer Science, Virginia Tech feng@cs.vt.edu

Abstract

Though TCP/IP is considered the *de facto* standard for Internet related wide area computing, its failure for *LambdaGrids* is well documented. On the other hand, rate-controlled UDP/IP-based protocols are strongly emerging as a feasible solution for meeting the performance goals in such environments. While such protocols have been able to avoid most the drawbacks of TCP/IP, they are still plagued by the drawbacks of UDP/IP, such as a host-based implementation, limiting their performance on high-speed networks. On the other hand, researchers have attempted to fix this drawback in TCP/IP using hardware offloaded TCP/IP implementations such as the TCP Offload Engines (TOEs). Given these two orthogonal developments, it is not completely clear about which is a better solution, i.e., a rate-controlled UDP/IP based protocol that is implemented in the host or a hardware offloaded TCP/IP solution such as the TOE. In this paper, we combine the benefits of both these solutions to design and develop an emulated UDP Offload Engine (UOE) based on the Chelsio T110 TOE; a solution which would transparently improve the performance for existing UDP/IP-based applications. Evaluations of our emulated UOE stack show that our design can achieve up to a 35% improvement in the performance while maintaining a significantly lower CPU usage.

Keywords: LambdaGrid, UDP, TOE, Sockets

1 Introduction

LambdaGrids are a new paradigm in distributed computing, where dedicated high-bandwidth optical networks allow globally distributed compute, storage and visualization systems to work together as a *planetary-scale* supercomputer. A supercomputer, which can enable scientists to analyze, correlate and visualize extremely large remote datasets, on demand and with a high performance; an effective platform for global scientific research. Realizing the *LambdaGrid* is comprised of two related, but subtly different aspects. The first aspect, is an environment to enable the *LambdaGrid*, i.e., several globally distributed nodes with different capabilities bundled together with high-bandwidth optical networks. This aspect is, to a large extent, a reality today [12, 13, 14]. The second aspect, is the capability to utilize the *LambdaGrid*, i.e., a networking protocol stack which can allow researchers to harness the potential of the *LambdaGrid*; an aspect which has been the focus of a significant amount of research in the past few years.

Though TCP/IP is considered the *de facto* standard for Internet related wide area computing, its failure for *LambdaGrids* is well documented [8]. Accordingly, researchers have looked for alternate solutions which can mask the limitations of TCP/IP and provide high performance data communication capabilities for such environments. Rate-controlled UDP/IP-based protocols [11, 20, 9, 19, 6, 21, 17] are strongly emerging as feasible solutions to meet these goals.

While these solutions have been able to avoid most of the limitations of TCP/IP, they are still plagued by the drawbacks of UDP/IP. For example, the host-based implementation of UDP/IP significantly hinders the performance these protocols can achieve for high-speed networks supporting throughputs of 10Gbps and higher. On the other hand, researchers have attempted to fix this drawback in TCP/IP using hardware offloaded TCP/IP implementations such as the TCP Offload Engines (TOEs). At where we stand today, it is not completely clear about which is a better solution, i.e., a rate-controlled UDP/IP-based protocol that is implemented in the host or a hardware offloaded TCP/IP solution such as the TOE. The ideal solution, however, is to take the best of both worlds – a hardware offloaded UDP/IP implementation, i.e., a UDP Offload Engine (UOE).

Given that there exists no UOE solution today, we decided to take on the challenge to utilize the capabilities of a TOE and enhance its features to form an emulated UOE suiting the requirement of the *LambdaGrid*. We take a two phase approach: (i) we develop a pseudo UDP/IP sockets layer which allows existing UDP/IP applications and application-level protocols to transparently run over the TOE and (ii) we enhance the features of the TOE to introduce the essence of UDP/IP communication into it. These two phases together form the emulated UOE. In this paper, we design and develop the emulated UOE based on the Chelsio T110 TOE. We also perform evaluations of our emulated UOE and compare it with the host-based UDP/IP stack. Our results show that our design can achieve up to a 35% improvement in the performance while maintaining a significantly lower CPU usage.

2 Background

In this section, we present a brief background about Protocol Offload Engines (POEs) and High Performance Sockets implementations for such POEs.

2.1 Protocol Offload Engines (POEs)

Traditionally, the processing of protocols such as TCP/IP and UDP/IP is accomplished via software running on the host CPU. As network speeds scale beyond a gigabit per second (Gbps), the CPU becomes overburdened with the large amount of protocol processing required. Resource-intensive memory copies, checksum computation, interrupts, and reassembly of out-of-order packets impose a heavy load on the host CPU. In high-

speed networks, the CPU has to dedicate more cycles to handle the network traffic than to the application(s) it is running. Protocol-Offload Engines (POEs) are emerging as a solution to limit the processing required by CPUs for networking.

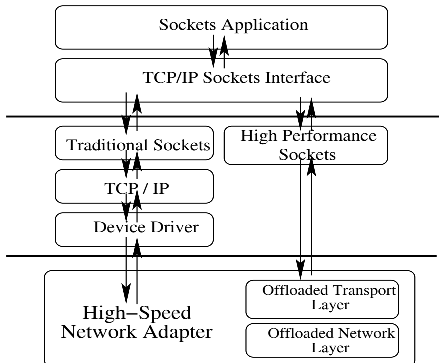


Figure 1: High Performance Sockets

The basic idea of a POE is to offload the processing of protocols from the host CPU to the network adapter. A POE can be implemented with a network processor and firmware, specialized ASICs, or both. High-performance networks such as InfiniBand (IBA) [1] and Myrinet [5] provide their own protocol stacks that are offloaded onto the network-adapter hardware. Many 10-Gigabit Ethernet (10GigE) vendors, on the other hand, have chosen to offload the ubiquitous TCP/IP protocol stack in order to maintain compatibility with existing infrastructure, particularly over the wide-area network (WAN) [7]. Consequently, this offloading is more popularly known as a TCP Offload Engine (TOE). The Chelsio T110 is one such TOE.

2.2 High Performance Sockets (HPS)

High-performance sockets are pseudo-sockets implementations that are built around two goals: (a) to provide a smooth transition to deploy existing sockets-based applications on clusters connected with networks using offloaded protocol stacks and (b) to sustain most of the network performance by utilizing the offloaded stack for protocol processing. These sockets layers override the existing kernel-based sockets and force the data to be transferred directly to the offloaded stack (Figure 1). The Sockets Direct Protocol (SDP) is an industry-standard specification for such high-performance sockets implementations.

3 Design and Implementation Details

As described in Section 1, we utilize the capabilities of TCP Offload Engines (TOEs) and the benefits of UDP/IP in Lambda-Grid environments to develop a UDP Offload Engine (UOE) emulation. In this section, we detail our approach in achieving this. In particular, we describe a high-performance UDP/IP sockets implementation on top of TOEs to allow transparent compatibility for existing UDP/IP applications in Section 3.1. In Section 3.2, we describe the modifications to the features of the Chelsio T110 TOEs to incorporate the benefits of UDP/IP.

3.1 High Performance UDP/IP Sockets

In order to implement a high-performance UDP/IP sockets implementation on top of TOEs, many intrinsic issues and non-trivial challenges need to be addressed. In this section, we discuss some of these challenges.

Connection Management Layer: UDP/IP is a connectionless protocol, i.e., applications using UDP/IP sockets do not explicitly need to open connections before data communication. On the other hand, TCP/IP is a connection-based protocol, i.e., explicit connection establishment is required for data communication. Building UDP/IP sockets on top of TCP/IP sockets requires building a thin layer to hide connection management issues from UDP/IP based applications. In our implementation, we built one such layer, termed as the Connection Management Layer (CML) to handle these details (Figure 2).

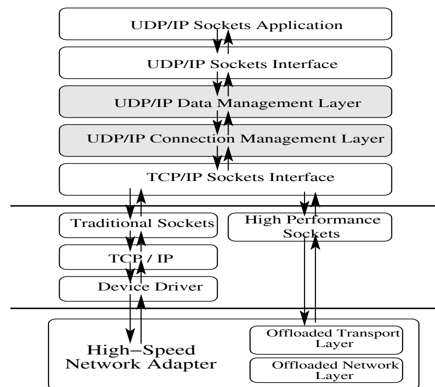


Figure 2: High Performance UDP/IP Sockets Architecture

When the upper layer tries to send or receive data to or from another node, the request is handed over to the CML. On the receiver side (when a *recv()* request is issued), the CML performs a *select()* operation waiting for either data to arrive on any of the previously established TCP/IP connections or for a connection request to arrive. If data arrives on any of the previously established TCP/IP connections, the data is copied into the appropriate buffer and the control returned to the upper layer. If on the other hand a connection request arrives, the connection is accepted, data received, copied into the appropriate buffer and then control returned to the upper layer.

On the sender side, the CML checks if a TCP/IP connection is previously established between the two nodes. If it is, the data is directly pushed out through a TCP/IP connection. Otherwise, the CML attempts to open a TCP/IP connection on a pre-decided port (provided as a configuration option to the CML). Here, there are two scenarios possible. In the first scenario, the receiver has already initiated a *listen()* call on the pre-decided port to *listen* for incoming connections. In this case, the connection establishment would be successful and the CML can directly send the data to the receiver. In the second scenario, the receiver has been delayed and has not initiated a *listen()* call on the pre-decided port yet. In this case, the connection establishment would not be successful and would return an error. The CML just returns this error to the upper layer to be handled in an appropriate manner.

Data Management Layer: The Data Management Layer

(DML) sits on top of the CML and manages error cases returned by the CML. For example, as mentioned earlier, if the receiver side is delayed due to some reason and does not call a *listen()* call, the CML's attempt to establish a connection fails; in this case, the CML just returns the error to the DML. The DML can handle this in a number of approaches.

In the first approach, the DML copies the data to be communicated into a temporary buffer, schedules the data to be communicated and returns the control to the application. Though this approach is simple, it has several disadvantages. First, the copy of the data into a temporary buffer adds a significant amount of overhead, especially for high-speed networks. Second, once the data is copied, though the communication is scheduled, the actual communication can only be initiated when the application tries to send or receive the next time. So, if the application gets delayed in a computation loop and does not send or receive any more data for a long time, the previously buffered data is not sent out and remains in the DML's temporary buffer.

The second approach is an extension of the first one. In this approach, the DML blocks while attempting to communicate the data, i.e., if the CML returns an error, it waits for a short amount of time and attempts to establish a connection again. It continues this till either a connection is established or a certain amount of time is elapsed after which it falls back to the first approach. In the current implementation, the fallback path is not entirely stable; hence, for all our experimental results we set an infinite timeout to fallback.

3.2 TOE Enhancements for the LambdaGrid

As described in Section 1, TCP/IP has several disadvantages as compared to UDP/IP in high-latency, high-bandwidth networks such as those in the LambdaGrid. Thus, building UDP/IP sockets on top of a TOE still suffers from the disadvantages of TCP/IP in such environments. In this section, we describe the features of TOEs that we tweaked to make them essentially behave like UDP/IP.

Disabling Congestion Control: The T110 adapters allow disabling congestion control for the flows being sent out. However, blindly disabling congestion using this approach has several disadvantages. Primarily, this setting is global for all connections in the system, i.e., we cannot force some connections to bypass congestion control while the others to retain them. Thus, if an application uses both TCP/IP sockets as well as UDP/IP sockets, congestion control is disabled for TCP/IP as well. To avoid this we would like all TCP/IP communication to go through the host-based TCP/IP stack and all UDP/IP communication to go over the offloaded TCP/IP stack (without congestion control).

Also, the T110 adapter does not allow selective connection offloading, i.e., during connection establishment the application cannot decide whether this connection is offloaded or not. However, the adapter allows the user to configure the number of connections that need to be offloaded; if this value is set to *five*, the first *five* connections are handed over to the TOE stack and the remaining are handled by the host-based TCP/IP stack. We will refer to this value as *N* for convenience.

In our implementation, we initially set the value of *N* to *zero* and disable congestion control on the TOE. When an application opens a TCP/IP connection, we directly allow it to do so; since *N* is set to *zero*, this connection is handled by the host-based TCP/IP stack, i.e., congestion control is enabled for this connection. If an application wants to do UDP/IP communication, as mentioned in Section 3.1, the CML tries to open a TCP/IP connection to the remote node; before doing this, however, the CML increments *N* to *one*. Thus, when the TCP/IP connection is established, it is handed over to the TOE, i.e., congestion control is disabled. In summary, if the application attempts to open a TCP/IP connection, it is always handled by the host-based TCP/IP stack and hence congestion control is on; if the application attempts to perform a UDP/IP communication, it is always handled by the TOE and hence congestion control is disabled.

Handling Out-of-Order Data: One of the advantages of UDP/IP is that out-of-order data is directly delivered to the application. In TCP/IP, if a packet is dropped, the following out-of-order data is not delivered to the user, but is held in the socket buffer. Thus, for high-latency network (e.g., those in the LambdaGrid), one packet loss could result in the out-of-order data being buffered for a very long time before its delivered to the application.

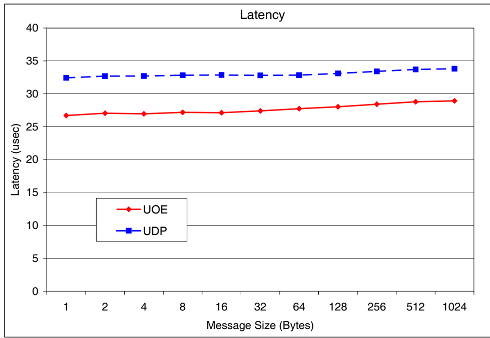
In our implementation, we added a kernel-level patch to the high-performance sockets layer above the TOE to directly deliver out-of-order data to the application. A copy of the data, however, is still maintained at the sockets layer, so that the TOE can use that to match sequence numbers and maintain reliability in the data transfer. Once the intermediate data arrives, when the sockets layer tries to deliver the data to the application, this data is discarded.

4 Experimental Results

In this section, we present micro-benchmark level evaluation comparing our implementation (UOE) with the traditional host-based UDP/IP protocol stack (UDP). Specifically, we present ping-pong latency, uni-directional bandwidth and CPU utilization results.

The test-bed used for the evaluation consists of two nodes equipped with dual Opteron 2.4 GHz processors, 1 MB of L2-cache and 4 GB of 200 MHz DDR SDRAM. The Linux distribution used was Suse 9.3 with the 2.6.6-SMP kernel.org kernel patched with the Chelsio TOE modules. The motherboard used was Tyan K8W Thunder and the chipset was AMD-8131. Each node also had a Chelsio T110 TOE-based 10-Gigabit Ethernet network adapter plugged into a 133 MHz/64-bit PCI-X slot; the nodes were connected back-to-back. The driver version used for the network adapters was 2.1.1.

Ping-Pong Latency: Figure 3(a) shows the point-to-point latency achieved by the two stacks, UOE and UDP. In this test, the sender node first sends a message to the receiver; the receiver receives this message and sends back another message to the sender. Such exchange of messages is carried out for several iterations, the total time calculated and averaged over the number of iterations. This gives the time for a complete message exchange. The ping-pong latency demonstrated in the figure is



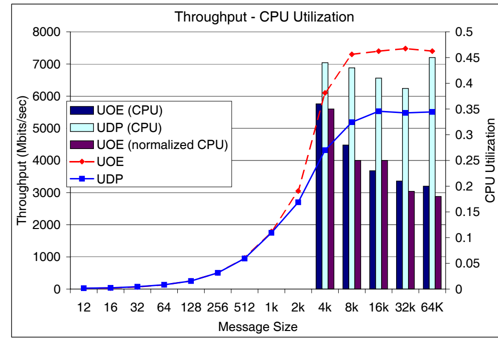


Figure 3: Micro-Benchmarks: (a) Ping-Pong Latency and (b) Unidirectional Throughput

half of this amount, i.e., one-way communication latency. As shown in the figure, UOE achieves a latency of about 26μs as compared to the 32μs achieved by host-based UDP/IP for a 1-byte message (improvement of 17%). This improvement in the latency is attributed to the better packet-processing capability of the TOE as compared to the host-based UDP/IP stack.

Uni-directional Throughput: Figure 3(b) shows the uni-directional throughput and CPU usage achieved by the two stacks. In this test, the sender node keeps streaming data at a constant rate to the receiver. Once the data transfer is completed, the data rate is calculated as the number of bytes sent out per unit time. We used the *iperf* benchmark version 2.0.2 for this experiment. As shown in the figure, UOE achieves a peak throughput of about 7.4Gbps as compared to the 5.5Gbps achieved by host-based UDP/IP (improvement of 35%).

For the CPU usage, we show three sets of numbers (Figure 3(b)). UDP (CPU) and UOE (CPU) are the amount of CPU utilized for achieving the peak throughput by the two stacks. However, this comparison is not entirely fair. Since UOE achieves a better throughput as compared to the host-based UDP/IP stack, the network adapter is able to send out data faster from the socket buffer. Accordingly, the sockets layer can copy data from the application faster (since its getting emptied faster, it has to wait less). The faster data copy rate for UOE, hence, shows a higher CPU usage in the measurements. To allow a fairer comparison, we slow down the UOE to the same speed as the host-based UDP/IP stack, measure the CPU usage and present the results as *UOE (normalized CPU)*. In this case, since the network is sending out data at the same rate, the comparison of the amount of CPU used is fairer. As shown in the figure, both UOE and normalized UOE outperform host-based UDP/IP. This is attributed to two reasons. First, the TOE performs a offloaded checksum computation as compared to the host-based UDP/IP which uses host cycles to do this. Second, the host-based UDP/IP uses an MTU size of 1500 bytes. Thus, if it has to send a 45Kbyte message, it uses 30 segments and receives 10 interrupts (assuming an interrupt every 3 segments). The TOE on the other hand, allows for segmentation offload where the NIC exposes a much larger MTU size to the host than the actual wire MTU. For example, in our experiments, the wire MTU was 1500 bytes while the TOE exposes a 16Kbyte MTU size. So, in the previous example, to send a 45Kbyte message, the host uses 3 segments and gets only 1 interrupt, reducing the CPU usage.

5 Related Work

Due to the poor performance of TCP/IP and the practical difficulties in deploying high speed TCP/IP kernel variants in LambdaGrid environments, several researchers including ourselves, have developed a number of UDP-based application-level protocols [11, 20, 9, 19, 6, 21, 17] for such environments. However, all of these implementations rely on developing a application-level protocols to handle details such as reliability, rate-control, etc.; this can cause their performance to be severely affected by factors such as context switches, CPU utilization, memory copies, operating system scheduling [10].

There has also been significant amount of research in high performance sockets implementations on the hardware offloaded protocol stacks of various networks [18, 15, 3, 4, 2, 7, 16]. However, most of these solutions have been for TCP/IP sockets and all of them have been developed and evaluated for the System Area Network (SAN) environment.

In this paper, we present a novel design to combine the capabilities of TOEs and the benefits of UDP/IP to provide an emulation of a UDP/IP Offload Engine (UOE) leading to a significantly higher performance and scalability in *LambdaGrids*.

6 Conclusions and Future Work

Though TCP/IP is considered the *de facto* standard for Internet related wide area computing, its failure for *LambdaGrids* is well documented. Several researchers came up with different solutions to tackle this problem. Two of these solutions seem to be particularly noticeable: (i) rate-controlled UDP/IP-based protocols and (ii) TCP/IP offload engines (TOEs). Rate-controlled UDP/IP-based protocols have been widely used by a number of researchers for the past several years; however, like UDP/IP, they rely on a host-based implementation, restricting the performance they can achieve on high-speed networks. On the other hand, TOEs allow a hardware-based implementation of the protocol stack, but suffer from the drawbacks of TCP/IP in the *LambdaGrid* environments. In this paper, we combined the benefits of both these solutions to form an emulated UDP Offload Engine (UOE) based on the Chelsio T110 TOE; a solution which would transparently improve the performance for existing UDP/IP-based applications.

As a part of the future work, we plan to design a version of *LambdaStream* [20] to utilize the UOE and benefit from the rate control on the network adapter.

7 Acknowledgments

We would like to thank Alan Verlo, Akira Hirano, Lance Long, Patrick Hallihan and Jeremy Archuleta for all their support.We would also like to thank Wael Noureddine, Felix Marti and Dimitrios Michailidis of Chelsio Communications for their help with the 10G Network adapters and valuable suggestions.

The Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago specializes in the design and development of high-resolution visualization and virtual-reality display systems, collaboration software for use on multi-gigabit networks, and advanced networking infrastructure. These projects are made possible by major funding from the National Science Foundation (NSF), awards CNS-0115809, CNS-0224306, CNS-0420477, SCI-9980480, SCI-0229642, SCI-9730202, SCI-0123399, ANI-0129527 and EAR-0218918, as well as the NSF Information Technology Research (ITR) cooperative agreement (SCI-0225642) to the University of California San Diego (UCSD) for "The OptIPuter" and the NSF Partnerships for Advanced Computational Infrastructure (PACI) cooperative agreement (SCI-9619019) to the National Computational Science Alliance. EVL also receives funding from the State of Illinois, General Motors Research, the Office of Naval Research on behalf of the Technology Research, Education, and Commercialization Center (TRECC), and Pacific Interface Inc. on behalf of NTT Optical Network Systems Laboratory in Japan.

This work was also supported in part by the DOE ASC Program through Los Alamos National Laboratory contract W-7405-ENG-36, and technical and equipment support from Chelsio Communications.

References

[1] Infiniband Trade Association. http://www.infinibandta.org.

[2] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial? In *the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 10-12 2004.

[3] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *the Proceedings of the IEEE International Conference on Cluster Computing*, pages 179–186, Chicago, Illinois, September 23-26 2002.

[4] P. Balaji, J. Wu, T. Kurc, U. Catalyurek, D. K. Panda, and J. Saltz. Impact of High Performance Sockets on Data Intensive Applications. In *the Proceedings of the IEEE International Conference on High Performance Distributed Computing (HPDC 2003)*, June 2003.

[5] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[6] Philip M. Dickens. FOBS: A Lightweight Communication Protocol for Grid Computing. In *Europar*, 2003.

[7] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. In *Proceedings of the IEEE International Symposium on High-Performance Interconnects (HotI)*, Palo Alto, CA, Aug 17-19 2005.

[8] W. Feng and P. Tinnakornsrisuphap. The Failure of TCP in High-Performance Computational Grids. In *SC 2000: High-Performance Networking and Computing Conference*, 2000.

[9] Y. Gu and R. Grossman. Experiences in the Design and Implementation of a High Performance Transport Protocol. In *SC*, 2004.

[10] Y. Gu and R. Grossman. Optimizing UDP-based Protocol Implementations. In *PFLDNet*, 2005.

[11] E. He, J. Leigh, O. Yu, and T. Defanti. Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. In *Cluster Computing*, 2002.

[12] http://www.nlr.net. National lambda rail.

[13] http://www.startap.net/translight/. Translight.

[14] http://www.surfnet.nl/. Surfnet.

[15] J. S. Kim, K. Kim, and S. I. Jung. SOVIA: A User-level Sockets Layer over Virtual Interface Architecture. In *Proceedings of Cluster Computing*, 2001.

[16] Myricom Inc. Sockets-GM Overview and Performance.

[17] N. Rao, Q. Wu, S. Carter, and W. Wing. Experimental Results On Data Transfers Over Dedicated Channels. In *BroadNets*, 2004.

[18] H. V. Shah, C. Pu, and R. S. Madukkarumukumana. High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In *Proceedings of CANPC workshop*, 1999.

[19] X . Wu and A. Chien. GTP: Group Transport Protocol for Lambda-Grids. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 2004.

[20] C. Xiong, J. Leigh, E. He, V. Vishwanath, T. Murata, L. Renambot, and T. Defanti. LambdaStream - a Data Transport Protocol for Streaming Network-Intensive Applications over Photnic Networks. In *PFLDNet*, 2005.

[21] X. Zheng, A. Mudambi, and M. Veeraraghavan. FRTP: Fixed Rate Transport Protocol. In *BroadNets*, 2004.