# Optimizing UDP-based Protocol Implementations

Yunhong Gu and Robert L. Grossman

*Abstract*—**Because of the poor performance of standard TCP over long distance high-speed networks and the practical difficulty in deploying kernel TCP variants, UDP-based transport protocols are often used for bulk data transfer. However, writing a transport protocol from scratch is not an easy job. One particular difficulty is achieving efficiency in the application level implementation.**

**This paper analyzes the performance characteristics of UDP and describes some optimization techniques that may be useful in most UDP-based protocol implementations, including memory processing, acknowledgment, loss processing, timing and self-clocking, and UDP IO. Based on these optimization principles, we implemented an open protocol framework (Composable UDT) that uses callbacks to process user defined event handlers. This framework can save significant time for network researchers and developers.**

## I. Introduction

Transmission Control Protocol (TCP) has been used successfully for decades. However, recently it has been shown that TCP has some performance shortcomings over wide area high-speed networks. The AIMD-based congestion control algorithm used by TCP is poor in discovering available bandwidth and recovering from packet loss in high bandwidth-delay product networks [1].

Network researchers have been working on new transport protocols and congestion control algorithms to support next generation high-speed networks. A lot of work, including TCP variants (FAST [2], BiC [3], Scalable [4], and HighSpeed [5]) and XCP [6], has demonstrated better performance in simulation and several limited network experiments. However, practical usage in real applications of these protocols is still very limited because of the implementation and installation difficulties. People who need to transfer bulk data (e.g., in grid computing) usually turn to application level solutions, among which UDP-based protocols are very popular, e.g., SABUL [7], UDT [8], Tsunami [9], RBUDP [10], FOBS [11], and GTP [12].

UDP-based protocols provide much better portability and are easy to install. However, although implementation of user level protocols needs less time to test and debug than the in-kernel implementations, it is difficult to make them as efficient. Because user level implementations cannot modify the kernel code, there may be additional context switches and memory copies. At high transfer speeds, these operations are very sensitive to CPU utilization and protocol performance.

In this paper, we introduce our work in optimizing the efficiency of UDP-based protocol implementations and show that with these ideas it is possible to implement efficient and practical composable frameworks for building UDP-based protocols. For example, our framework of Composable UDT, which is based on the UDT (UDP-based Data Transfer library) implementation [8], can easily support different congestion control algorithms, such as those of high-speed TCP variants [2, 3, 4, 5] or RBUDP's blasting of packets [5].

The advantages of the Composable UDT framework include: 1) new UDP-based protocols can be rapidly prototyped and tested; 2) different congestion control approaches can be easily compared experimentally; and 3) application specific protocols using UDP can be developed relatively easily. For example, specific protocols for distributed data intensive computing or streaming media applications can be developed using this framework.

The main disadvantage is the additional overhead of the framework itself, which compounds the additional overhead of an application layer protocol compared to a kernel layer protocol. To achieve high performance, we have been guided by two principles:

*There should be no bursts in CPU utilization, especially at the data receiving side*. CPU bursts can cause incoming data to not be processed in time and thus be dropped by the end system. This can cause serious damage to loss based congestion control algorithms like TCP's AIMD, in which a packet drop usually takes a long time to recover in long distance networks.

*The overall CPU utilization should be as small as possible*. Poor implementation can prevent applications from using the maximum available bandwidth (because CPU is used up first). Moreover, other data processing operations, which also need significant CPU time, often co-exist with the data transfers.

The paper is organized as follows. We will first analyze UDP's performance on different systems in Section 2. In Section 3 we will describe how to optimize the implementation for the two principles above. Section 4 introduces a composable framework that provides optimized protocol components that can be configured directly for new protocols. The paper is concluded in Section 5.

## II. UDP Performance Characteristics

A good experimental understanding of UDP's performance is critical for implementing the two principles above. In this section we perform several experiments to check UDP's CPU utilization, end system delay, and the impact of packet and buffer sizes on the transfer speed. This

Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet 2005), Lyon, France, February 3 – 4, 2005.

The authors are with the Laboratory for Advanced Computing, University of Illinois, Chicago, IL 60607 USA. (Email: gu@lac.uic.edu, grossman@uic.edu)

Robert L. Grossman is also with Open Data Partners.

The work was supported in part by National Science Foundation under grants number 0129609 and 9977868.

**Table 1. Experiment Environments**

| Name | CPU | Memory | NIC | OS |
|---|---|---|---|---|
| *onno* | Dual Itanium2 1.5GHz | 8 GB | 10 GbE | Linux 2.6.0 |
| *sara77* | Dual Xeon 2.4GHz | 2 GB | 1 GbE | Linux 2.4.18 |
| *ncdm171* | Dual PowerPC G4 1GHz | 2 GB | 1 GbE | Mac OS X |
| *win91* | Dual Xeon 2.4GHz | 2 GB | 1 GbE | Windows XP Professional |
| *ncdm87* | Dual Opteron 2.4GHz | 4 GB | 1 GbE | Linux 2.6.8 |

**Table 2. CPU Utilization and End System Delay of UDP and TCP**

| Name | UDP | | | TCP | | |
|---|---|---|---|---|---|---|
| | CPU Util. (MHz/Mbps) | | Delay (ms) | CPU Util. (MHz/Mbps) | | Delay (ms) |
| | Sending | Receiving | | Sending | Receiving | |
| *onno* | 0.22 | 0.35 | 0.062 | 0.23 | 0.50 | 0.068 |
| *sara77* | 0.40 | 0.45 | 0.070 | 0.51 | 0.51 | 0.086 |
| *ncdm171* | 1.22 | 1.45 | 0.202 | 2.22 | 2.73 | 0.245 |
| *win91* | 1.03 | 1.09 | 0.203 | 1.14 | 1.28 | 0.302 |
| *ncdm87* | 0.26 | 0.40 | 0.065 | 0.25 | 0.56 | 0.087 |

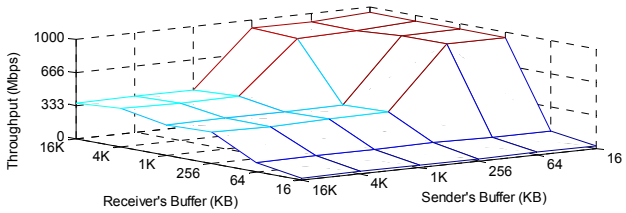


**Figure 1. UDP performance vs. socket buffer size.**

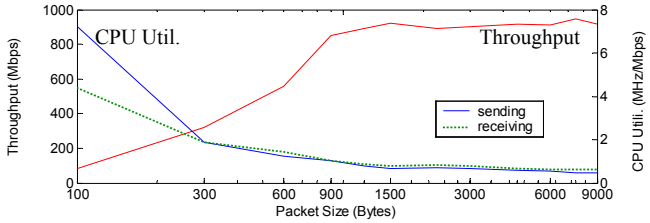


**Figure 2. UDP performance vs. packet size.**

forms the basis of some of our optimization schemes in the next section.

The experiments were performed on five different systems (Table 1). They were all connected to at least one other machine with the same configuration through a network with bandwidth no less than the NIC speed. The MTU on the testbeds is 1500 bytes.

We fixed the UDP packet size at 1500 bytes (measured at IP level) and the UDP socket buffer at 1 MB, and recorded the CPU utilization and end system delay. As a comparison, the same measurements for TCP are also listed. The results are listed in Table 2. The CPU utilization is measured by MHz/Mbps, which is the product of the CPU frequency and the CPU usage percentage over the data transfer rate. The experiment was performed on local networks. (Here the CPU frequency is the aggregate frequency of all processors used by the testing application. The number of processors can be 0.5 in some systems where hyper-threading technology is used.)

To eliminate the RTT error impact on the result, we use local network connections with very small RTT values; otherwise large RTT values can impair the accuracy of the relatively small end system delays. The values for delay listed in Table 2 are only the delay caused by the end system, excluding the network delay.

The experimental results are the average values of 100 runs.

Table 2 shows that UDP has smaller CPU utilization and end system delay than TCP, mainly because it has less processing overhead. This means that with effective optimization, UDP-based protocol implementations can have performance similar to in-kernel TCP implementations.

We conducted additional experiments by changing the two parameters of 1) UDP socket buffer and 2) packet size to find out their impact on performance. The result between *sara77* and another local machine can be found in Figures 1 and 2. In the experiment we maximized the UDP transfer speed by limiting the loss rate below 0.1%.

Figure 1 shows that the socket buffer size at the receiver side is the deciding factor relate to throughput. It should be large enough to reach optimal throughput but after an optimal value larger buffers do not help. Meanwhile, the sender side buffer should be significantly smaller than the receiver side buffer.

The small sender side buffer is required because: 1) a larger buffer increases the network RTT; and 2) a larger buffer increases the possibility of overwhelming the network. The former situation causes more serious problems when packet loss occurs, whereas the latter situation causes more packet losses. Meanwhile, the sending buffer size should be large enough so that it will not limit the packet sending speed. This minimum value is related to RTT.

Figure 2 shows that the packet size should be equal to the MTU size. Although it can be seen that larger packet size leads to smaller CPU utilization (because there is less processing overhead), the possibility of segmentation collapse makes this dangerous [13].

A performance improvement provided by packet scattering/gathering is also significant. This technique can be used to avoid one memory copy by avoiding using a temporary buffer for packing and unpacking packet headers and user data.

## III. OPTIMIZATIONS

### A. Memory and Buffer Management

In most transport protocols, memory processing and network IO take the most CPU time [8]. For a UDP-based application level protocol, it is important to bypass the memory copy into the kernel space by using the overlapped IO semantics [8].

It is important to effectively utilize the CPU time and to reduce waste in the CPU utilization. Idle time should be avoided, which means packet sending has to be continuous. The gap between two data blocks has to be eliminated, otherwise it can interrupt packet sending and cause throughput drop. We have used double buffering in a previous version of SABUL for block based data transfer. In streaming transfer, the protocol needs to support multiple buffers in memory management, within the total size limit.

Finally, since the protocol is used for bulk data transfer, a large sized buffer is also necessary. However, the buffer, either at the sending side or at the receiving side, needs to be created or resized dynamically, otherwise the system memory will be used up as the number of flows increases.

### B. Timing, Rate Control, and Self-clocking

Rate-based control is often used in bulk data transfer in order to eliminate packet sending bursts. At high sending speed, an accurate timer is needed. For example, at 1Gbps, the packet sending period is 12.5 microseconds for 1500-byte packets. Only a timer that is at least a magnitude more precise than a microsecond can guarantee a smooth rate for such high speed. In UDT we use CPU clock cycles.

However, rate control results in an implementation problem. Currently, there is no general operating system that provides such a high precision timer. This ironically turns into a situation where rate-control implementations may still depend on self-clocking and cause similar packet sending burst problems as window-based approaches do.

Self-clocking implementations can potentially be improved. One important principle is to trigger the data sending by any events or signals the protocol can use, including timeouts, API calls, packet arrivals, etc.

We are still working on this issue to provide a packet sending mechanism that has limited packet burst with only limited use of busy loop waiting.

### C. Acknowledgment

Acknowledgment is a very expensive operation to both the end host and the networks. To send and process an acknowledgment packet takes significant time, compared to sending a data packet. A regular acknowledgment packet needs to update the protocol buffer and protocol parameters (e.g., window size). In addition, although an acknowledgment packet can be very small in size, it requires almost the same time as an MTU-sized packet to be sent and received. In contrast to kernel space implementation, processing of acknowledgements at user space causes additional context switches and synchronization (when using multiple threads) costs, which are not necessary in kernel space.

In UDT, we have recommended timer-based selective acknowledgment, which only acknowledges at every constant interval. To reduce the end system delay, an acknowledgment is also sent when a user buffer is fulfilled.

For protocols that require acknowledging every data packet, a lightweight acknowledgment can be sent. Such a lightweight acknowledgment only updates the related sequence number, and leaves the other processing for a regular acknowledgment, which is still sent at every constant time.

### D. UDP IO

Since at least a sequence number will be added to each data packet as the packet header, the memory copy bypassing also needs to use the socket gathering/scattering technique, which is well known today.

At the receiver side, the sequence number of the incoming packet has to be guessed in order to put the data directly into the protocol buffer, instead of a temporary storage. This prediction is not difficult since most of the packets are in order.

### E. Disk IO

File transfer contributes to a large portion of the network usage in grid computing. It is important to provide a file transfer interface from a network transport library because it has more opportunity to optimize the process. In fact, many UDP-based protocols are only designed for file transfer, e.g., Tsunami and FOBS.

One optimization technique is to use the *mmap* call on the files and access them as pseudo memory blocks. The performance improvement of this technique depends on system implementations.

Considering that there are already 10Gb/s network widely available, disk IO is often the bottleneck for bulk file transfer. A specific optimized disk IO module for large file IO may be very helpful. Certain disk optimization techniques are evaluated in [19].

### F. Threading

Inside the application level protocol, there could be several threads to support different functionalities at the same time, e.g., API, listening socket, etc. For packet sending and receiving, a single thread can be used (e.g., with the *select* call). However, such implementations cannot make use of multiple processors for packet processing. On the other hand, using multiple threads can bring some synchronization overhead.

For example, UDT uses two threads for data sending and receiving, respectively, where the sending threads will not be started until the first *send* call (lazy start). The synchronization in a sending-only UDT entity is about 5% (Figure 3).

### G. Loss Processing

Loss processing is one of the most critical factors that can cause the problem of CPU utilization burst, especially in high-speed long distance networks where there may be thousands of outstanding packets and their states have to be maintained in a very efficient way. Many researchers have

identified the deficiency in the loss processing of the Linux TCP implementation.

We have described a loss list management scheme in [8]. The technique makes use of the fact that large amounts of packet losses are often continuous. We record loss events instead of lost packets; the former is much less than the latter. The idea is also useful for explicit loss notification, similar to the SACK option in TCP [14].

### *H. Code Optimization*

The hot spots of CPU utilization in transport protocol implementations are the data sending and receiving loops. The code inside the loops can be executed about $10^5$ times per second at 1Gb/s transfer speed with 1500 bytes packet size, so they must be fine tuned. In addition, if multi-threading is used, the synchronization between the threads also needs to be tuned to fine granularity

### *I. CPU Usage Profiling*

In the last part of this section, we use our UDT implementation [8] as an example to demonstrate how each protocol module affects the CPU utilization. UDT has two threads for packet sending and receiving, respectively.

We ran two simple UDT applications: one sends data from *win91*, the other receives data on *win91*, both using overlapped IO. We profiled the CPU clock ticks on the UDT level. The CPU time consumed at the upper layer (application) and the lower layer (UDP) was not sampled because the UDT library cannot optimize those layers.

Figure 3 depicts the CPU utilization of each UDT module. Note that the modules of buffer management, loss processing, congestion/flow control, and UDP IO are called from the sending and receiving threads, so their CPU times are overlapped with the sending threads for sending and with the receiving thread for receiving, respectively. The sender and receiver threads cost most of the CPU time. The UDP IO time mentioned below is only the time consumed in the UDT layer (i.e., function calls of OS socket and necessary preparation of the calls) and excludes the data IO time inside the OS kernel.

Buffer management, loss information processing, congestion/flow control, and UDP IO take the majority of the CPU time. Inside the congestion control module, there are functionalities including timing and performance monitoring, which are the major CPU consumers (Figure 3) in this module.

For data sending, both the sender and the receiver threads have to be started, so there is synchronization overhead between the two threads (about 5%). For data receiving, only one thread is necessary, and the synchronization time is zero.

Figure 4 depicts the CPU utilization of each UDT functionality. In Figure 4 we note that the CPU utilization on the overhead of the congestion control is very small (less than 1%). Therefore, the use of callback functions (which will be introduced in the next section) will not significantly impact efficiency in UDT.

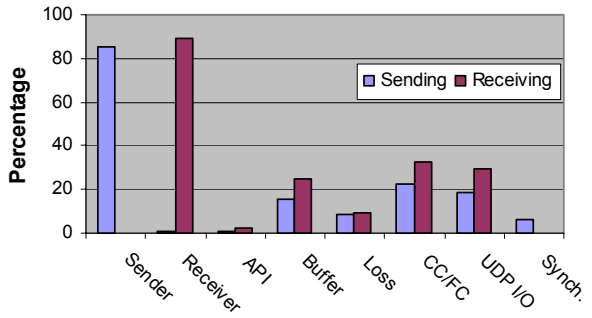


**Figure 3. UDT CPU utilization by modules. This figure shows the CPU utilization of seven UDT modules (Sender, Receiver, API, Buffer management, Loss processing, Congestion and flow control, and UDP channel) and the synchronization between the sender and receiver threads.**

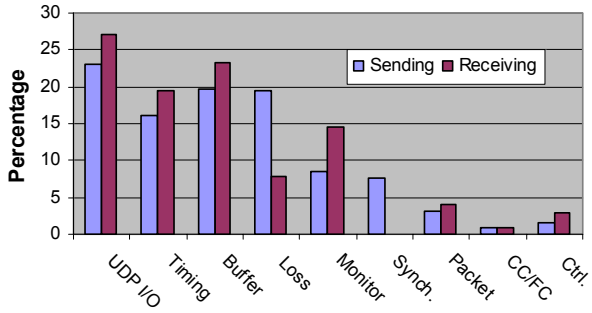


**Figure 4. UDT CPU utilization by functionalities. The functionalities listed above include UDP IO, Timing, buffer and memory operation, loss list access, performance and statistics monitoring, synchronization between threads, data packing/unpacking, congestion/flow control, and control message processing.**

## IV. A COMPOSABLE FRAMEWORK FOR PROTOCOLS

In this section, we describe the composable UDT framework, with which new UDP-based protocols can be easily developed. Currently, the framework enables alternate congestion control algorithms to be easily developed. We plan next to support protocols that do not necessarily require lossless transport.

This support for configurable congestion control is based upon our UDT implementation [15]. We provide four categories of extensions: 1) access to internal parameters (e.g., RTT, RTO, etc), 2) callbacks of control event handlers, 3) user defined packets, and 4) packet sending control.

The internal UDT protocol parameters, such as RTT and loss rate, can be read from a special UDT program interface (*perfmon*). Such information is useful in many equation control algorithms such as TFRC [16]. We will also provide write access to some of the parameters. For example, there are multiple proposals for the TCP RTO calculation. This performance monitoring facility can also be used for the diagnosis of the new control mechanisms.

We provide a C++ base class that collected a series of control event handlers. A new control algorithm should inherit from this class and override necessary event handlers

to update at least one of the two control parameters: the packet sending period (for rate control) and the congestion window size (for window control). These event handlers are callback functions and will be processed inside UDT.

Such event handlers include *init*, *onACK*, *onLoss*, *onTimeout*, *onPktSent*, *onPktReceived*, and *processCustonMsg*, which are called at the beginning of a connection (initialization), when ACK is received, when loss is reported, when timeout occurs, when a data packet is sent, when a data packet is received, and when a customized control message is received, respectively.

To support receiver-based control algorithms, such as certain video streaming control, UDT provides user defined control messages to acknowledge the sender with the receiver requested sending rate.

Figure 5 demonstrates the convenience of using composable UDT with an example of the implementation of a simplified version of TCP (without slow start). The *CTCP* class inherits the base *CCC* class, which contains all the control parameters and event handlers. It disables the rate control (*m_dPktSndPeriod = 0.0*) and sets an initial *cwnd* size of two segments (*m_dCWndSize = 2.0*). The receiver should acknowledge every second data packet (*setACKInterval(2)*). The remaining steps are to increase the congestion window size by 1/*cwnd* per ACK and decrease it by half per loss event.

```
class CTCP: public CCC
{
public:
   virtual void init()
   {
      m_dPktSndPeriod = 0.0;
      m_dCWndSize = 2.0;
      setACKInterval(2);
   }

   virtual void onACK(const int&)
   {
      m_dCWndSize += 1.0/m_dCWndSize;
   }

   virtual void onLoss(const int*, const int&)
   {
      m_dCWndSize *= 0.5;
   }
};
```

**Figure 5. Simplified TCP algorithm using Composable UDT.**

To demonstrate that this approach provides adequate performance we compared standard TCP implemented in the kernel (K-TCP) to an application layer TCP implemented using our composable framework (U-TCP). Table 3 shows the ratio of *MHz/Mbps* for various machines described in Table 1. (For convenient comparison, Table 3 also shows the K-TCP performance listed in Table 2.)

Figure 6 depicts the throughput changes of these two TCP implementations against duration time. In this situation U-TCP behaves quite similarly to K-TCP. The moving average (the average throughput from the beginning until the current instance) is also shown on the figure and it can be seen that the two implementations have similar performance. The average throughput tends to be equal as the duration increases.

**Table 3. CPU utilization of K-TCP and U-TCP (MHz/Mbps)**

| Machines | Sender | | Receiver | |
|---|---|---|---|---|
| | K-TCP | U-TCP | K-TCP | U-TCP |
| *onno* | 0.23 | 0.60 | 0.50 | 0.44 |
| *sara77* | 0.51 | 0.65 | 0.51 | 0.78 |
| *ncdm171* | 2.22 | 3.46 | 2.73 | 3.26 |
| *win91* | 1.14 | 2.07 | 1.28 | 1.20 |
| *ncdm87* | 0.25 | 0.52 | 0.56 | 0.60 |

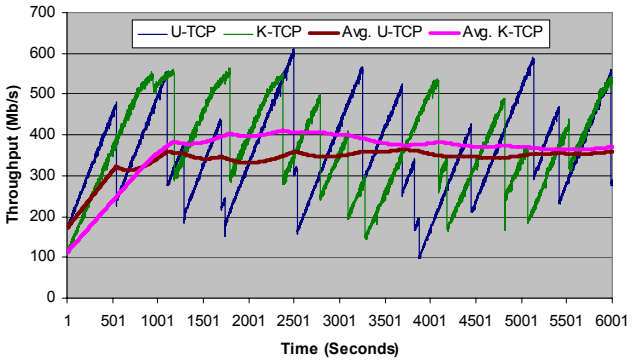


**Figure 6: Throughout of K-TCP and U-TCP**

We plan to perform more experiments to examine the performance of Composable UDT framework. On the one hand, we will compare K-TCP and U-TCP in more realistic network environments with various topologies, bandwidth, loss rates, etc. On the other hand, we will implement more control algorithms using Composable UDT to compare the performance with their respective independent implementations.

## V. Conclusion and Future Work

UDP-based protocols are often used in data intensive applications for bulk data transfer. However, it is difficult to implement a new protocol in a fast and efficient way. We introduce a composable framework so that new transport protocols can be rapidly developed and easily compared to one another. Using this framework, a new congestion control algorithm can be implemented by writing a few lines of C++ code. A framework like this requires carefully optimizing the implementation, which is also described in this paper.

We emphasize here that the composable UDT framework is by no means a "one-size-fits-all" approach for common transport protocols. It is only intended to be used for implementations of UDP-based protocols and experimental studies of new congestion control algorithms. It is not meant to replace certain kernel space protocols including TCP, SCTP [17], and DCCP [18].

We plan to provide more built-in congestion control classes (e.g., FAST TCP) as part of the composable UDT releases, thus the users will not need to implement them repeatedly. User contributed codes are also welcome.

## REFERENCES


[1] W. Feng and P. Tinnakornsrisuphap. The Failure of TCP in High-Performance Computational Grids. SC '00, Dallas, TX, Nov. 4 - 10, 2000.

[2] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. IEEE Infocom '04, Hongkong, China, Mar. 2004.

[3] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, Experimental Standard, Dec. 2003.

[4] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. ACM Computer Communication Review, Apr. 2003.

[5] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control for Fast Long-Distance Networks. IEEE Infocom '04, Hongkong, China, Mar. 2004.

[6] D. Katabi, M. Hardley, and C. Rohrs. Internet Congestion Control for Future High Bandwidth-Delay Product Environments, ACM SIGCOMM '02, Pittsburgh, PA, Aug. 19 - 23, 2002.

[7] Yunhong Gu and Robert Grossman, SABUL: A Transport Protocol for Grid Computing, Journal of Grid Computing, 2003, Volume 1, Issue 4, pp. 377-386.

[8] Yunhong Gu, Xinwei Hong, and Robert L. Grossman, Experiences in the Design and Implementation of a High Performance Transport Protocol, SC '04, Pittsburgh, PA.

[9] Mark, R. Meiss, Tsunami: A High-Speed Rate-Controlled Protocol for File Transfer, http://steinbeck.ucs.indiana.edu/~mmeiss/papers/tsunami.pdf, retrieved on Sep. 28, 2004.

[10] He, E., Leigh, J., Yu, O., DeFanti, T. A., Reliable Blast UDP: Predictable High Performance Bulk Data Transfer, Proc. IEEE Cluster Computing, Sept, Chicago, Illinois, 2002.

[11] Phillip M. Dickens, FOBS: A Lightweight Communication Protocol for Grid Computing, Europar 2003.

[12] Ryan X. Wu, and Andrew Chien, Evaluation of Rate Based Transport Protocols for Lambda-Grids, to appear in Proceedings of the 13th IEEE HPDC, Honolulu, Hawaii, June 2004.

[13] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. IEEE/ACM Trans. on Networking, 7(4): 458-472, 1999.

[14] Floyd, S., Mahdavi, J., Mathis, M., and Podolsky, M., An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883, Proposed Standard, July 2000.

[15] UDT software release, http://sourceforge.net/projects/udt.

[16] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. Equation-Based Congestion Control for Unicast Applications. In ACM SIGCOMM 2000, Stockholm, Aug. 2000.

[17] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream control transmission protocol. RFC 2960, Oct. 2000.

[18] Eddie Kohler, Mark Handley, Sally Floyd, Jitendra Padhye, Datagram Congestion Control Protocol (DCCP), http://www.icir.org/kohler/dcp/. Jan. 2005.

[19] W. Hsu and A. J. Smith, The performance impact of I/O optimizations and disk improvements, IBM Journal of Research and Development, Volume 48, Issue 2 (March 2004), Pages: 255 – 289.