A Dynamic Performance-Based Flow Control Method for High-Speed Data Transfer

Ben Eckart, *Student Member, IEEE,* Xubin He, *Senior Member, IEEE,* Qishi Wu, *Member, IEEE* and Changsheng Xie

Abstract—New types of specialized network applications are being created that need to be able to transmit large amounts of data across dedicated network links. TCP fails to be a suitable method of bulk data transfer in many of these applications, giving rise to new classes of protocols designed to circumvent TCP's shortcomings. It is typical in these high-performance applications, however, that the system hardware is simply incapable of saturating the bandwidths supported by the network infrastructure. When the bottleneck for data transfer occurs in the system itself and not in the network, it is critical that the protocol scale gracefully to prevent buffer overflow and packet loss. It is therefore necessary to build a high-speed protocol adaptive to the performance of each system by including a dynamic performance-based flow control. This paper develops such a protocol, Performance Adaptive UDP (henceforth PA-UDP), which aims to dynamically and autonomously maximize performance under different systems. A mathematical model and related algorithms are proposed to describe the theoretical basis behind effective buffer and CPU management. A novel delay-based rate throttling model is also demonstrated to be very accurate under diverse system latencies. Based on these models, we implemented a prototype under Linux and the experimental results demonstrate that PA-UDP outperforms other existing high-speed protocols on commodity hardware in terms of throughput, packet loss, and CPU utilization. PA-UDP is efficient not only for high-speed research networks but also for reliable high-performance bulk data transfer over dedicated local area networks where congestion and fairness are typically not a concern.

Index Terms—flow control, high-speed protocol, reliable UDP, bulk transfer

1 Introduction

A certain class of next generation science applications needs to be able to transfer increasingly large amounts of data between remote locations. Toward this goal, several new dedicated networks with bandwidths upwards of 10 Gbps have emerged to facilitate bulk data transfers. Such networks include UltraScience Net (USN) [1], CHEETAH [2], OSCARS [3], User Controlled Light Paths (UCLP) [4], Enlightened [5], Dynamic Resource Allocation via GMPLS Optical Networks (DRAGON) [6], Japanese Gigabit Network II [7], Bandwidth on Demand (BoD) on Geant2 network [8], Hybrid Optical and Packet Infrastructure (HOPI) [9], Bandwidth Brokers [10], and others.

- *B. Eckart and X. He are with the Department of Electrical and Computer Engineering, Tennessee Technological University, Cookeville, TN, 38505.*
  E-mail: {bdeckart21, hexb}@tntech.edu
- *Q. Wu is with the Department of Computer Science, University of Memphis, Memphis, TN, 38152*
  E-mail: qishiwu@memphis.edu
- *C. Xie is with the Data Storage Division of Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China 430074*
  E-mail: cs_xie@hust.edu.cn

Manuscript received xx; revised xx.

The goal of our work is to present a protocol that can maximally utilize the bandwidth of these private links through a novel performance-based system flow control. As Multi-Gigabit speeds become more pervasive in dedicated LANs and WANs and as hard drives remain relatively stagnant in read and write speeds, it becomes increasingly important to address these issues inside of the data transfer protocol. We demonstrate a mathematical basis for the control algorithms we use, and we implement and benchmark our method against other commonly used applications and protocols. A new protocol is necessary, unfortunately, due to the fact that the *de facto* standard of network communication, TCP, has been found to be unsuitable for high-speed bulk transfer. It is difficult to configure TCP to saturate the bandwidth of these links due to several assumptions made during its creation.

The first shortcoming is that TCP was made to distribute bandwidth equally among the current participants in a network and uses a congestion control mechanism based on packet loss. Throughput is halved in the presence of detected packet loss and only additively increased during subsequent loss-free transfer. This is the so-called Additive Increase Multiplicative Decrease algorithm (AIMD) [13]. If packet loss is a good indicator of network

congestion, then transfer rates will converge to an equal distribution among the users of the network. In a dedicated link, however, packet loss due to congestion can be avoided. The partitioning of bandwidth therefore can be done via some other, more intelligent bandwidth scheduling process, leading to more precise throughput and higher link utilization. Examples of advanced bandwidth scheduling systems include the centralized control plane of USN and GMPLS (Generalized Multiple Protocol Label Switching) for DRAGON [11] [38]. On a related note, there is no need for TCP's slow-start mechanism because dedicated links with automatic bandwidth partitioning remove the risk of a new connection overloading the network. For more information, see [12].

A second crucial shortcoming of TCP is its congestion window. To ensure in-order, reliable delivery, both parties maintain a buffer the size of the congestion window and the sender sends a burst of packets. The receiver then sends back positive acknowledgments (ACK's) in order to receive the next window. Using timeouts and logic, the sender decides which packets are lost in the window and resends them. This synchronization scheme ensures that the receiver receives all packets sent, in-order, and without duplicates; however, it can come at a price. On networks with high latencies, reliance on synchronous communication can severely stunt any attempt for high-bandwidth utilization because the protocol relies on latency-bound communication. For example, consider the following throughput equation relating latency to throughput. Disregarding the effects of queuing delays or packet loss, the effective throughput can be expressed as

$$\text{throughput} = \frac{cwin \times MSS}{rtt} \tag{1}$$

where $cwin$ is the window size, MSS the maximum segment size, and rtt the round-trip time. With a congestion window of 100 packets and a maximum segment size of 1460 bytes (the difference between the MTU and TCP/IP header), a network with an infinite bandwidth and 10 ms round-trip time would only be able to achieve approximately 120 Mbps effective throughput. One could attempt to mitigate the latency bottleneck by letting $cwin$ scale to the bandwidth-delay product ($BW \times rtt$) or by striping and parallelizing TCP streams (see BBCP [15]), but there are also difficulties associated with these techniques. Regardless, Equation 1 illustrates the potentially deleterious effect of synchronous communication on high-latency channels.

Solutions to these problems have come in primarily two forms: modifications to the TCP algorithm and application-level protocols which utilize UDP for asynchronous data transfer and TCP for control and data integrity issues. This paper focuses on the class of high-speed reliable UDP protocols [20], which include SABUL/UDT [16], [17], RBUDP [18], Tsunami [19], and Hurricane [39]. Despite the primary focus on these protocols, most of the techniques outlined in this paper could be applied to any protocol for which transfer bandwidths are set using inter-packet delay.

The rest of the paper is organized as follows. High-speed TCP and High-speed reliable UDP are discusses in Sections 2 and 3, respectively. The goals for high-speed bulk data transfer over reliable UDP are discussed in Section 4. Section 5 defines our mathematical model. Section 6 describes the architecture and algorithms for the PA-UDP protocol. Section 7 discusses the implementation details of our PA-UDP protocol. Experimental results and CPU utilization statistics are presented in Section 8. We examine related work in Section 9 and draw our conclusions in Section 10.

2 TCP Solutions

As mentioned in Section 1, the congestion window provided by TCP can make it impossible to saturate link bandwidth under certain conditions. In the example pertaining to Equation 1, one obvious speed boost would be to increase the congestion window beyond one packet. Assuming a no-loss link, a window size of n packets would allow for $12.5n$ Mbps throughput. On real networks, however, it turns out that the Bandwidth-Delay Product (BDP) of the network is integral to the window size. As the name suggests, the BDP is simply the product of the bandwidth of the channel multiplied by the end-to-end delay of the hosts. In a sense, this is the amount of data present "on the line" at any given moment. A 10 Gbps channel with a RTT of 10 ms would need approximately a 12.5 Megabyte buffer on either end, because at any given time, 12.5 Megabytes would be on the line that potentially would need to be resent due to errors in the line or packet loss at the receiving end. Ideally, a channel could sustain maximum throughput by setting the BDP equal to the congestion window, but it can be difficult to determine these parameters accurately. Moreover, the TCP header field uses only 16 bits to specify window size. Therefore, unless the TCP protocol is rewritten at the kernel level, the largest usable window is 65 Kilobytes. Note that there are modifications to TCP that can increase the window size for large BDP networks [14]. Efforts in this area also include dynamic windows, different acknowledgment procedures, and statistical measurements for channel parameters. Other TCP variants attempt to modify the congestion control algorithm to be more amenable to characteristics of high-speed networks. Still others look toward multiple

TCP streams, like bbFTP, GridFTP, and pTCP. Most employ a combination of these methods, including (but not limited to) High Speed TCP [43], Scalable TCP [44], and FAST TCP [46].

Many of the TCP-based algorithms are based in the transport layer and thus kernel modification is usually necessary to implement them. Some also rely on specially configured routers. As a result, the widespread deployment of any of these algorithms would be a very daunting task. It would be ideal to be able to run a protocol on top of the the two standard transport layer protocols, TCP and UDP, so that any computer could implement them. This would entail an application-level protocol which could combine the strengths of UDP and TCP and which could be applied universally to these types of networks.

3 High-Speed Reliable UDP

High-speed Reliable UDP protocols include SABUL/UDT [16], [17], RBUDP [18], Tsunami [19], and Hurricane [39], among others [20].

UDP-based protocols generally follow a similar structure: UDP is used for bulk data transfer, and TCP is used marginally for control mechanisms. Most high-speed reliable UDP protocols use delay-based rate control to remove the need for congestion windows. This control scheme allows a host to statically set the rate and undoes the throughput-limiting stairstep effects of AIMD. Furthermore, reliable delivery is ensured with either delayed, selective or negative acknowledgments of packets. Negative acknowledgments are optimal in cases where packet loss is minimal. If there is little loss, acknowledging only lost packets will incur the least amount of synchronous communication between the hosts. A simple packet numbering scheme and application-level logic can provide in-order, reliable delivery of data. Finally, reliable UDP is positioned at the application level, which allows users to explore more customized approaches to suit the type of transfer, whether it is disk-to-disk, memory-to-disk, or any combination thereof.

Due to deliberate design choices, most High-Speed Reliable UDP protocols have no congestion control or fairness mechanisms. Eschewing fairness for simplicity and speed improvements, UDP-based protocols are meant to be deployed only on private networks where congestion is not an issue, or where bandwidth is partitioned apart from the protocol.

Reliable UDP protocols have shown varying degrees of success in different environments, but they all ignore the effects of disk throughput and CPU latency for data transfer applications. In such high-performance distributed applications, it is critical that system attributes be taken into account to make sure both sending and receiving parties can support the required data rates. Many tests show artificially high packet loss because of the limitations of the end systems in acquiring the data and managing buffers. In this paper, we show that this packet loss can be largely attributed to the effects of lackluster disk and CPU performance. We then show how these limitations can be circumvented by a suitable architecture and a self-monitoring rate control.

4 Goals for High-Speed Bulk Transfer

Ideally, we would want a high-performing protocol suitable for a variety of high-speed, high-latency networks without much configuration necessary at the user level. Furthermore, we would like to see good performance on many types of hardware, including commodity hardware and disk systems. Understanding the interplay between these algorithms and the host properties is crucial.

On high-speed, high-latency, congestion-free networks, a protocol should strive to accomplish two goals: to maximize goodput by minimizing synchronous, latency-bound communication and to maximize the data rate according to the receiver's capacity. (Here we define goodput as the throughput of usable data, discounting any protocol headers or transport overhead [21].)

Latency-bound communication is one of the primary problems of TCP due to the positive acknowledgment congestion window mechanism. As previous solutions have shown, asynchronous communication is key to achieving maximum goodput. When UDP is used in tandem with TCP, UDP packets can be sent asynchronously, allowing the synchronous TCP component to do its job without limiting the overall bandwidth.

High-speed network throughputs put considerable strain on the receiving system. It is often the case that disk throughput is less than half of the network's potential and that high-speed processing of packets greatly taxes the CPU. Due to this large discrepancy, it is critical that the data rate is set by the receiver's capacity. An overly high data rate will cause a system buffer to grow at a rate relative to the difference between receiving and processing the data. If this mismatch continues, packet loss will inexorably occur due to finite buffer sizes. Therefore, any protocol attempting to prevent this must continually communicate with the sender to make sure that the sender only sends at the receiver's specific capacity.

5 A Mathematical Model

Given the relative simplicity of high-speed UDP algorithms, mathematical models can be constructed

with few uncontrollable parameters. We can exploit this determinism by tweaking system parameters for maximum performance. In this section, we produce a mathematical model relating buffer sizes to network rates and sending rates to inter-packet delay times. These equations will be used to predict the theoretical maximum bandwidth of any data transfer given a system's disk and CPU performance characteristics.

Since the host receiving the data is under considerably more system strain than the sender, we shall concentrate on a model for the receiver and then briefly consider the sender.

The receiver's capacity can be thought of as an equation relating its internal system characteristics with those of the network. Two buffers are of primary importance in preventing packet loss at the receiving end: the kernel's UDP buffer and the user buffer at the application level.

5.1 Receiving Application Buffers

For the protocols which receive packets and write to disk asynchronously, the time before the receiver has a full application buffer can be calculated with a simple formula. Let t be time in seconds , $r(\cdot)$ be a function which returns the data rate in bits per second (bps) of its argument, and m be the buffer size in bits. The time before m is full is given by

$$t = \frac{m}{r(\text{recv}) - r(\text{disk})} \tag{2}$$

At time t the receiver will not be able to accept any more packets and thus will have to drop some. We found this to be a substantial source of packet loss in most high-speed reliable UDP protocols. To circumvent this problem, one may put a restriction on the size of the file sent by relating file size to $r(recv) \times t$. Let f be the size of a file and f_{max} be its maximum size.

$$f_{max} = \frac{m}{1 - \frac{r(disk)}{r(recv)}} \tag{3}$$

Note that f_{max} can never be negative since $r(disk)$ can only be as fast as $r(recv)$. Also note that, if the two rates are equally matched, f_{max} will be infinite since the application buffer will never overflow.

Designing a protocol that limits file sizes is certainly not an acceptable solution, especially since we have already stipulated that these protocols need to be designed to sustain very large amounts of data. Therefore, if we can set the rate of the sender, we can design an equation to accommodate our buffer size and $r(disk)$. Rearranging, we see that

$$r(recv) = \frac{r(disk)}{1 - \frac{m}{f}} \tag{4}$$

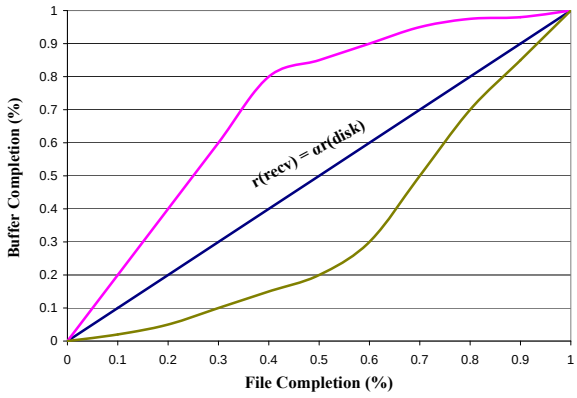


Fig. 1. File vs. buffer completion during the course of a transfer. Three paths are shown; the path that adheres to α is optimal.

or if we let

$$\alpha = \frac{1}{1 - \frac{m}{f}} \tag{5}$$

we can then arrive at

$$r(recv) = \alpha r(disk) \tag{6}$$

or

$$\alpha = \frac{r(recv)}{r(disk)} \tag{7}$$

We can intuitively see that, if the ratio between disk and network activity remains constant at α, the transfer will make full use of the buffer while minimizing the maximum value of $r(recv)$. To see why this is the case, consider Figure 1. The middle line represents a transfer which adheres to α. If the transfer is to make full use of the buffer, then any deviations from α at some point will require a slope greater than α since $r(disk)$ is assumed to be at its peak for the duration of the transfer. Thus, $r(recv)$ must be increased to compensate. Adjusting $r(recv)$ to maintain α while $r(disk)$ fluctuates will keep the transfer optimal in the sense that $r(recv)$ has the lowest possible maximum value while total throughput for the data transfer is maximized. The CPU has a maximum processing rate and by keeping the receiving rate from spiking, we remove the risk of overloading the CPU. Burstiness has been recognized as a limiting factor in previous literature [22]. Additionally, the entirety of the buffer is used during the course of transfer, avoiding the situation of a suboptimal transfer rate due to unused buffer.

Making sure the buffer is only full at the end of the data transfer has other important consequences as well. Many protocols fill up the application buffer as fast as possible, without regard to the state of the transfer. When the buffer fills completely, the receiver must issue a command to halt any

further packets from being sent. Such a requirement is problematic due to the latency involved with this type of synchronous communication. With a 100 millisecond round-trip time (*rtt*) on a 10 Gbps link, the receiver would potentially have to drop in excess of 80,000 packets of size 1500 bytes before successfully halting the sender. Furthermore, we do not want to use a higher peak bandwidth than is absolutely necessary for the duration of the transfer, especially if we are held to an imposed bandwidth cap by some external application or client. Holding to this α ratio will achieve optimal throughput in terms of disk and CPU performance.

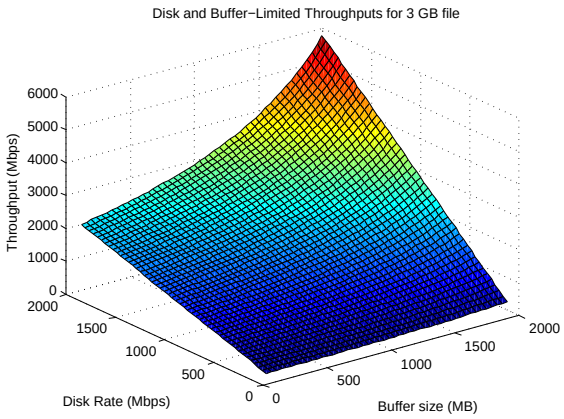


Fig. 2. Throughputs for various parameters

The theoretical effects of various system parameters on a 3 GB transfer are shown in Figure 2. Note how simply increasing the buffer size does not appreciably affect the throughput but increasing both $r(disk)$ and m provides the maximum performance gain. This graph also gives some indication of the computational and disk power required for transfers exceeding 1 Gbps for bulk transfer.

5.2 Receiving Kernel Buffers

Another source of packet loss occurs when the kernel's receiving buffer fills up. Since UDP was not designed for anything approximating reliable bulk transfer, the default buffer size for UDP on most operating systems is very small; on Linux 2.6.9, for example, it is set to a default of 131 kB. At 131 kB, a 1 Gbps transfer will quickly deplete a buffer of size m:

$$t = \frac{m}{r(recv)} = \frac{131 \text{ kB}}{1000 \text{ Mbps}} \approx 1.0 \text{ ms}$$

Note that full depletion would only occur in the complete absence of any receiving calls from the application. Nevertheless, any CPU scheduling latency must be made to be shorter than this time, and the average latency-rate must conform to the processing rate of the CPU such that the queue does not slowly build and overflow over time. A rigorous mathematical treatment of the kernel buffer would involve modeling the system as a queuing network, but this is beyond the scope of the paper.

Let $t_{\%}$ represent the percentage of time during execution that the application is actively receiving packets, and $r(CPU)$ be the rate at which the CPU can process packets.

$$t_{\%} \geq \frac{r(recv)}{r(CPU)} \tag{8}$$

For example, if $r(CPU) = 2 \times r(recv)$, then the application will only need to be actively receiving packets from the buffer 50% of the time.

Rate modeling is an important factor in all of these calculations. Indeed, equations 4, 5, and 6 would be useless if one could not set a rate to a high degree of precision. TCP has been known to produce complicated models for throughputs, but fortunately our discussion is greatly simplified by a delay-based rate that can be employed in congestion-free environments. Let L be the datagram size (set to the MTU) and t_d be the time interval between transmitted packets. Thus, we have

$$r(recv) = \frac{L}{t_d} \tag{9}$$

In practice, it is difficult to use this equation to any degree of accuracy due to context switching and timing precision limitations. We found that, by using system timers to measure the amount of time spent sending and sleeping for the difference between the desired time span and the sending time, we could set the time delay to our desired time with a predictably decreasing error rate. We found the error rate as a percentage difference between the desired sending rate and the actual sending rate as

$$e(recv) = \frac{\beta}{t_d} \tag{10}$$

where t_d is the desired inter-packet delay and β is a value which can be determined programmatically during the transfer. We used a floating β, dynamic to the statistics of the transfer. Using the *pthreads* library under Linux 2.6.9, we found that β generally was about 2e-6 for each transfer. Taking this error into account, we can update our original rate formula to obtain

$$r^*(recv) = \frac{L}{t_d} - \frac{\beta L}{t_d^2} \tag{11}$$

Figure 3 shows the percentage error rate between equation 9 and the true sending rate. As shown by *Projected*, we notice that the error due to scheduling

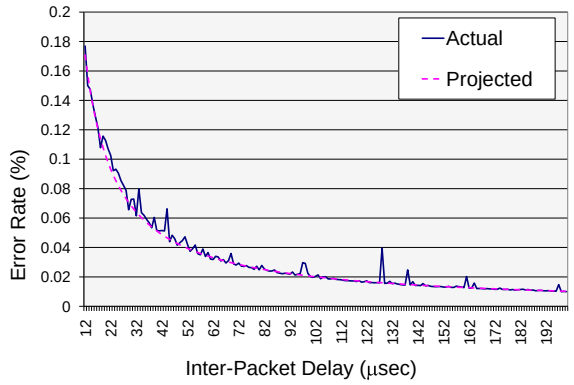


Fig. 3. Actual and predicted error rates vs inter-packet delay

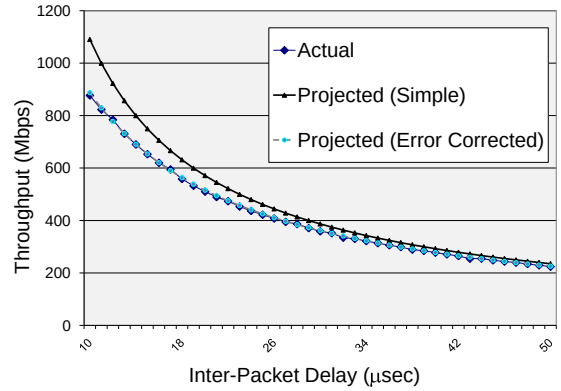


Fig. 4. Send rate vs inter-packet delay. Note that the actual and error-corrected predicted rates are nearly indistinguishable.

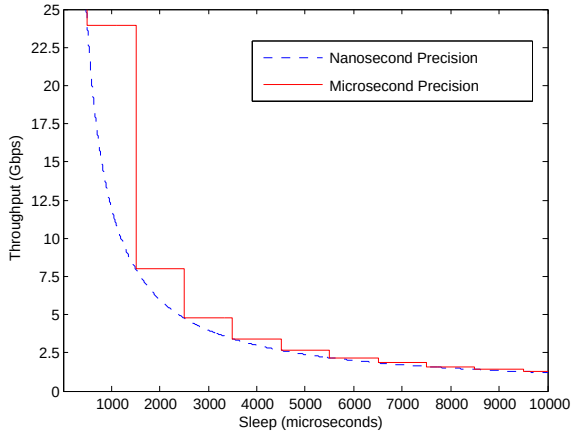


Fig. 5. Effects of timing granularity

can be predicted with a good degree of certainty by equation 10. In Figure 4, the different rate calculations for various inter-packet delays can be seen. Equation 11, with the error rate factored in, is sufficiently accurate for our purposes.

It should be noted that under extremely high bandwidths certain aspects of a system that one might take for granted begin to break down. For instance, many kernels support only up to microsecond precision in system-level timing functions. This is good enough for bandwidths lower than 1 Gbps, but unacceptable for higher capacity links. As shown in Figure 5, the resolution of the timing mechanism has a profound impact on the granularity of the delay-based rates. Even a 1 Gbps channel with microsecond precision has some trouble matching the desired sending rate. This problem has been noted previously in [22] and has been usually solved by timing using clock cycles. SABUL/UDT uses this technique for increased precision.

Another source of breakdown can occur at the hardware level. To sustain a 10 Gbps file transfer for a 10 GB file, according to Equation 6, a receiver must have a sequential disk write rate of

$$r(disk) = 10 \times 10^9 \text{ bps} \times (1 - \frac{m}{80 \times 10^9}) \qquad (12)$$

where m is in bits and $r(disk)$ in bits per second.

We can see the extreme strain this would cause to a system. In the experiments described in [23], 5 Ultra SCSI disks in RAID 0 could not achieve 800 Mbps for a 10 GB file. Assuming a sequential write speed of 1 Gbps, Equation 12 shows that we would require a 9 GB buffer. Similarly, a 10 Gbps transfer rate would put considerable strain on the CPU. The exact relation to CPU utilization would depend on the complexity of the algorithms behind the protocol.

5.3 The Data Sender

Depending on the application, the sender may be locked into the same kinds of performance-limiting factors as the receiver. For disk-to-disk transfers, if the disk read rate is slower than the bandwidth of the channel, the host must rely on pre-allocated buffers before the transfer. This is virtually the same relationship as seen in Equation 6. Unfortunately, if the bottleneck occurs at this point, nothing can be done but to improve the host's disk performance. Unlike the receiver, however, CPU latency and kernel buffers are less crucial to performance and disk read speeds are almost universally faster than disk write speeds. Therefore, if buffers of comparable size are used (meaning α will be the same), the burden will always be on the receiver to keep up with the sender and not vice versa. Note that this only applies for disk-to-disk transfers. If the data is being generated in real-time, transfer speed limitations will depend on the computational aspects of the data being generated. If the generation rate is higher than the channel bandwidth then the generation rate must be throttled down or buffers must be used. Otherwise, if the generation rate is

```
epoch: time for thread to sleep
maxMem: size of application buffer
maxSetRate: maximum rate imposed by receiver
pktSize: size of a packet
hdrSize: size of a packet header
fSize: size of the file
while transfer do
    prevDisk ← number of packets written to disk
    prevNet ← number of packets received over network
    sleep(epoch)
    curDisk ← number of packets written to disk
    curNet ← number of packets received over network
    rateNet ← (curNet − prevNet) × pktSize/epoch
    memLeft ← maxMem − (curNet − curDisk) × pktSize
    bitsLeft ← fSize − curNet × pktSize - hdrSize
    rateDisk ← (curDisk − prevDisk) × pktSize/epoch
    if bitsLeft - memLeft > 0 then
        alpha ← bitsLeft/(bitsLeft − memLeft)
        newRate ← rateDisk × alpha
        newRate ← min(maxSetRate, newRate)
    else
        newRate ← maxSetRate
    end if
    sendTCP(newRate)
end while
```

Fig. 6. A dynamic rate control algorithm based on the buffer management equations of Section 5.

lower than channel bandwidth, a bottleneck occurs at the sending side and max link utilization may be impossible.

6 Architecture and Algorithms

First we discuss a generic architecture which takes advantage of the considerations related in the previous section. In the next three sections, a real-life implementation is presented, and its performance is analyzed and compared to other existing high-speed protocols.

6.1 Rate Control Algorithms

According to Equation 6, given certain system characteristics of the host receiving the file, an optimum rate can be calculated so that the receiver will not run out of memory during the transfer. Thus, a target rate can be negotiated at connection time. We propose a simple three way handshake protocol where the first SYN packet from the sender asks for a rate. The sender may be restricted to 500 Mbps, for instance. The receiver then checks its system parameters, $r(disk)$, $r(recv)$, and m, and either accepts the supplied rate, or throttles the rate down to the maximum allowed by the system. The following SYNACK packet would instruct the sender of a change, if any.

Data could then be sent over the UDP socket at the target rate, with the receiver checking for lost packets and sending retransmission requests periodically over the TCP channel upon discovery of lost packets. The requests must be spaced out in time relative to the RTT of the channel, which can also be roughly measured during the initial handshake, so that multiple requests are not made for the same packet while the packet has already been sent but not yet received. This is an example of a negative acknowledgment system, because the sender assumes the packets were received correctly unless it receives data indicating otherwise.

TCP should also be used for dynamic rate control. The disk throughput will vary over the course of a transfer, and as a consequence should be monitored throughout. Rate adjustments can then proceed according to Equation 6. To do this, disk activity, memory usage, and data rate must be monitored at specified time intervals. The dynamic rate control algorithm is presented in Figure 6. A specific implementation is given in Section 7.

6.2 Processing Packets

Several practical solutions exist to decrease CPU latency for receiving packets. Multithreading is an indispensable step to decouple other processes which have no sequential liability with one another. Minimizing I/O and system calls and appropriately using mutexes can contribute to overall efficiency. Thread priorities can often guarantee CPU attentiveness on certain kernel scheduler implementations. Also, libraries exist which guarantee high-performance, low latency threads [24], [25]. Regardless of the measures mentioned above to curb latency, great care must be made to keep the CPU attentive to the receiving portion of the program. Even the resulting latencies from a single print statement inline with the receiving algorithm may cause the build-up and eventual overflow of the UDP buffer.

Priority should be given to the receiving portion of the program given the limitations of the CPU. When the CPU cannot receive data as fast as it is sent, the kernel UDP buffer will overflow. Thus a multithreaded program structure is mandated so that disk activity can be decoupled with the receiving algorithm. Given that disk activity and disk latencies are properly decoupled, appropriate scheduling priority is given to the receiving thread, and rate control is properly implemented, optimal transfer rates will be obtained given virtually any two host configurations.

Reliable UDP works by assigning an ordered ID to each packet. In this way, the receiver knows when packets are missing and how to group and write the packets to disk. As stipulated previously, the receiver gets packets from the network and writes them to disk in parallel. Since most disks have write speeds well below that of a high-speed network, a growing buffer of data waiting to be written to disk will occur. It is therefore a priority to maximize disk performance. If datagrams are received out of order they can be dynamically

```
seqList: array of datagram ID's in the order they were received
datagramList: array of datagrams in the order they were received
datagram: holds datagram received from socket
i ← 0
totalDatagrams ← ⌈fileSize / (packetSize - headerSize)⌉
while i < totalDatagrams do
   while i = seqList[i] do
      i ← i + 1
   end while
   swap(datagramList[i], datagramList[seqList[i]])
   swap(seqList[i], seqList[seqList[i]])
end while
```

Fig. 7. The post-file processing algorithm in pseudocode.

rearranged from within the buffer, but a system waiting for a packet will have to halt disk activity at some point. In this scenario, we propose that when using PA-UDP, most of the time it is desirable from a performance standpoint to naively write packets to disk as they are received, regardless of order. The file can then be reordered afterwards from a log detailing the order of ID reception.

See Figure 7 for pseudocode of this algorithm. Note that this algorithm is only superior to in-order disk writing if there are not too many packets lost and written out of order. If the rate control of PA-UDP functions as it should, little packet loss should occur and this method should be optimal. Otherwise, it may be better to wait for incoming packets that have been lost before flushing a section of the buffer to disk.

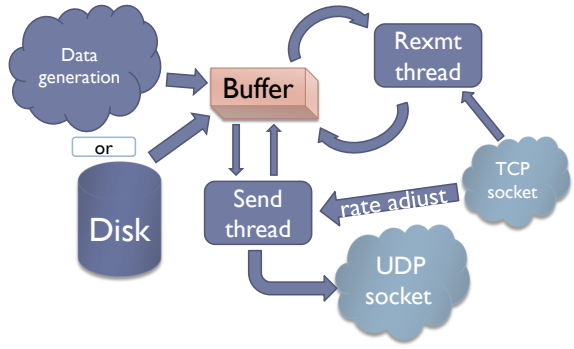


Fig. 8. PA-UDP: The Data Sender

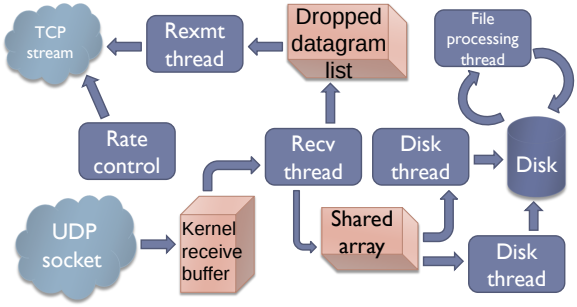


Fig. 9. PA-UDP: The Data Receiver

7 Implementation Details

To verify the effectiveness of our proposed protocol, we have implemented PA-UDP according to the architecture discussed in Section 6. Written mostly in C for use in Linux and Unix environments, PA-UDP is a multithreaded application designed to be self-configuring with minimal human input. We have also included a parametric latency simulator so we could test the effects of high-latencies over a low-latency Gigabit LAN.

7.1 Data Flow and Structures

A loose description of data flow and important data structures for both the sender and receiver is shown in Figures 8 and 9. The sender sends data through the UDP socket, which is asynchronous, while periodically probing the TCP socket for control and retransmission requests. A buffer is maintained so the sender does not have to reread from disk when a retransmitted packet is needed. Alternatively, when the data is generated, a buffer might be crucial to the integrity of the received data if data is taken from sensors or other such non-reproducible events.

At the receiver end as shown in Figure 9, there are six threads. Threads serve to provide easily-attainable parallelism, crucially hiding latencies. Furthermore, the use of threading to achieve periodicity of independent functions simplifies the system code. As the *Recv thread* receives packets, two *Disk threads* write them to disk in parallel. Asynchronously, the *Rexmt thread* sends retransmit requests, and the *Rate control* thread profiles and sends the current optimum sending rate to the sender. The *File processing thread* ensures the data is in the correct order once the transfer is over.

The *Recv thread* is very sensitive to CPU scheduling latency and thus should be given high scheduling priority to prevent packet loss from kernel buffer overflows. The UDP kernel buffer was increased to 16 Megabytes from the default of 131 kB. We found this configuration adequate for transfers of any size. Timing was done with microsecond precision by using the *gettimeofday* function. Note, however, that better timing granularity is needed for the application to support transfers in excess of 1 Gbps.

The PA-UDP protocol handles only a single client at a time, putting the others in a wait queue. Thus, the threads are not shared among multiple connections. Since our goal was maximum link utilization over a private network, we were not concerned with multiple users at a time.

7.2 Disk Activity

In the disk write threads, it is very important from a performance standpoint that writing is done syn-

chronously with the kernel. File streams normally default to being buffered, but in our case, this can have adverse effects on CPU latencies. Normally, the kernel allocates as much space as necessary in unused RAM to allow for fast returns on disk writing operations. The RAM buffer is then asynchronously written to disk, depending on which algorithm is used, write-through, or write-back. We do not care if a system call to write to disk halts thread activity, because disk activity is decoupled from data reception and halting will not affect the rate at which packets are received. Thus it is not pertinent that a buffer be kept in unused RAM. In fact, if the transfer is large enough, eventually this will cause a premature flushing of the kernel's disk buffer, which can introduce unacceptably high latencies across all threads. We found this to be the cause of many dropped packets even for file transfers having sizes less than the application buffers. Our solution was to force synchrony with repeated calls to *fsync*.

As shown in Figure 9, we employed two parallel threads to write to disk. Since part of the disk thread's job is to corral data together and do memory management, better efficiency can be achieved by having one thread do memory management while the other is blocked by the hard disk and vice versa. A single-threaded solution would introduce a delay during memory management. Parallel disk threads remove this delay because execution is effectively pipelined. We found that the addition of a second thread significantly augmented disk performance.

Since data may be written out of order due to packet loss, it is necessary to have a reordering algorithm which works to put the file in its proper order. The algorithm discussed in Section 6 is given in Figure 7.

7.3 Retransmission and Rate Control

TCP is used for both retransmission requests and rate control. PA-UDP simply waits for a set period of time and then makes grouped retransmission requests if necessary. The retransmission packet structure is identical to Hurricane [39]. An array of integers is used, denoting datagram ID's that need to be retransmitted. The sender prioritizes these requests, locking down the UDP data flow with a mutex while sending the missed packets.

It is not imperative that retransmission periods be calibrated except in cases where the sending buffer is small or there is a very large *rtt*. Care needs to be made to make sure that the *rtt* is not more than the retransmission wait period. If this is the case, requests will be sent multiple times before the sender can possibly resend them, resulting in duplicate packets. Setting the retransmission period at least 5 times higher than the *rtt* ensures that this will not happen while preserving the efficacy of the protocol.

The retransmission period does directly influence the minimum size of the sending buffer, however. For instance, if a transfer is disk-to-disk and the sender does not have a requested packet in the application buffer, a seek time cost will incur when the disk is accessed non-sequentially for the packet. In this scenario, the retransmission request would considerably slow down the transfer during this time. This can be prevented by either increasing the application buffer or sufficiently lowering the retransmission sleep period.

As outlined in Figure 6, the rate control is computationally inexpensive. Global count variables are updated per received datagram and per written datagram. A profile is stored before and after a set sleep time. After the sleep time, the pertinent data can be constructed, including $r(recv)$, $r(disk)$, m, and f. These parameters are used in conjunction with Equations 6 and 7 to update the sending rate accordingly. The request is sent over the TCP socket in the simple form "RATE: R" where R is an integer speed in Mbps. The sender receives the packet in the TCP monitoring thread and derives new sleep times from Equation 11. Specifically, the equation used by the protocol is

$$t_d = \frac{L + \sqrt{L^2 - 4\beta LR}}{2R} \tag{13}$$

where R represents the newly requested rate.

As per the algorithm in Figure 6, if the memory left is larger than the amount left to be transferred, the rate can be set at the allowed maximum.

7.4 Latency Simulator

We included a latency simulator to more closely mimic the characteristics of high-*rtt* high-speed WANs over low-latency high-speed LANs. The reasons for the simulator are twofold: the first reason is simply a matter of convenience, given that testing could be done locally, on a LAN. The second reason is that simulations provide the means for parametric testing which would otherwise be impossible in a real environment. In this way, we can test for a variety of hypothetical *rtt*'s without porting the applications to different networks. We can also use the simulator to introduce variance in latency according to any parametric distribution.

The simulator works by intercepting and timestamping every packet sent to a socket. A loop runs in the background which checks to see if the current time minus the timestamp is greater than the desired latency. If the packet has waited for the desired latency, it is sent over the socket. We should note that the buffer size needed for the simulator is

numRealloc: count variable for number of datagram reallocations currently made
numWritten: count variable for number of datagrams currently written to disk
datagramList: array of received datagrams
if numRealloc < numWritten **then**
  datagram ← reallocate(datagramList[numRealloc])
  numRealloc ← numRealloc + 1
else
  datagram ← allocate(new datagram)
end if

Fig. 10. Memory management algorithm

related to the desired latency and the sending rate. Let b be the size of the latency buffer and t_l be the average latency.

$$b \approx r(send) \times t_l \tag{14}$$

By testing high-latency effects in a parametric way, we can find out how adaptable the timing aspects are. For instance, if the retransmission thread has a static sleep time before resending retransmission requests, a high latency could result in successive yet unnecessary requests before the sender could send back the dropped packets. The profiling power of the rate control algorithm is also somewhat affected by latencies, since ideally the performance monitor would be real-time. In our tests, we found that PA-UDP could run with negligibly small side-effects with *rtt*'s over one second. This is mainly due to the relatively low variance of $r(disk)$ that we observed on our systems.

7.5 Memory Management

For high-performance applications such as these, efficient memory management is crucial. It is not necessary to delete packets which have been written to disk, since this memory can be reallocated by the application when future packets come through the network. Therefore, we used a scheme whereby each packet's memory address is marked once the data it contains is written to disk. When the network receives a new packet, if a marked packet exists, the new packet is assigned to the old allocated memory of the marked packet. In this way, we do not have to use the C function *free* until the transfer is over. The algorithm is presented in Figure 10.

8 Results and Analysis

8.1 Throughput and Packet Loss Performance

We tested PA-UDP over a Gigabit Ethernet switch on a LAN. Our setup consisted of two Dell PowerEdge 850's each equipped with a 1 Gigabit NIC, dual Pentium 4 processors, 1 GB of RAM, and a 7200 RPM IDE hard drive.

We compared PA-UDP to three UDP-based protocols: Tsunami, Hurricane, and UDT (UDT4). Five trials were conducted at each file size for both protocols using the same parameters for buffers and speeds. We used buffers 750 MB large for each protocol, and we generated test data both on-the-fly and from the disk. The average throughputs and packet loss percentages are given in Tables 1 and 2, respectively, for the case when data was generated dynamically. The results are very similar for disk-to-disk transfers.

PA-UDP performs favorably to the other protocols, excelling at each file size. Tsunami shows high throughputs, but fails to be consistent at higher file sizes due to large retransmission errors. At larger file sizes, Tsunami fails to complete the transfers, instead restarting *ad infinitum* due to internal logic decisions for retransmission. Hurricane completes all transfers, but does not perform consistently and suffers dramatically due to high packet loss. UDT shows consistent and stable throughputs, especially for large transfers, but adopts a somewhat more conservative rate control than the others.

In addition to having better throughputs as compared to Tsunami, Hurricane and UDT, PA-UDP also has virtually zero packet loss due to buffer overflow. This is a direct result of the rate control algorithm from Figure 6, which preemptively throttles bandwidth before packet loss from buffer overflows occur. Tsunami and Hurricane perform poorly in these tests largely due to unstable rate control. When the receiving rate is set above the highest rate sustainable by the hardware, packet loss eventually occurs. Since the transmission rates are already at or above the maximum capable by the hardware, any extra overhead incurred by retransmission requests and the handling of retransmitted packets causes even more packet loss, often spiraling out of control. This process can lead to final packet retransmission rates of 100% or more in some cases, depending on the file size and protocol employed. Tsunami has a simple protection scheme against retransmission spiraling that involves completely restarting the transfer after too much much packet loss has occurred. Starting the transfer over voids the pool of packets to be retransmitted with the hope that the packet loss was a one-time error. Unfortunately, this scheme causes the larger files in our tests to endlessly restart and thus never complete, as shown in Tables 1 and 2. UDT does not seem to have these problems, but shows lower throughputs than PA-UDP.

8.2 CPU Utilization

As discussed in Section 5, one of the primary benefits of our flow control method is its low CPU utilization. The flow control limits the transfer speeds to the optimal range for the current hardware profile of the host. Other protocols without this

TABLE 1
Throughput Averages

	Average Throughput / Std. Dev. (Mbps)			
File Size (MB)	PA-UDP	Tsunami	Hurricane	UDT
100	947.15 / 0.07	374.30 / 16.65	452.63 / 11.47	235.14 / 29.98
400	953.42 / 0.99	608.88 / 1.60	200.22 / 44.79	273.05 / 18.21
800	948.66 / 1.32	341.99 / 19.92	157.4 / 44.05	282.25 / 13.84
1000	938.51 / 10.94	294.96 / 11.51	145.08 / 71.49	295.90 / 11.57
2000	450.20 / 2.28	294.03 / 83.65	124.21 / 30.48	295.94 / 9.14
3000	371.01 / 10.43	timeout	87.11 / 4.27	246.78 / 6.46
5000	331.96 / 7.73	timeout	93.12 / 19.33	269.50 / 17.95

TABLE 2
Packet Loss Averages

	Average Packet Loss / Std. Dev. (%)			
File Size (MB)	PA-UDP	Tsunami	Hurricane	UDT
100	0.00 / 0.00	0.00 / 0.00	18.67 / 12.90	2.10 / 2.62
400	0.00 / 0.00	0.00 / 0.00	75.47 / 19.04	1.41 / 0.77
800	0.03 / 0.04	34.03 / 26.46	180.93 / 84.41	1.44 / 0.28
1000	0.01 / 0.02	41.48 / 21.40	122.18 / 107.95	0.46 / 0.57
2000	0.00 / 0.00	50.71 / 60.76	135.53 / 51.60	0.87 / 0.72
3000	0.00 / 0.00	time-out	230.83 / 31.95	0.00 / 0.00
5000	0.00 / 0.00	time-out	213.87 / 57.14	0.79 / 0.97

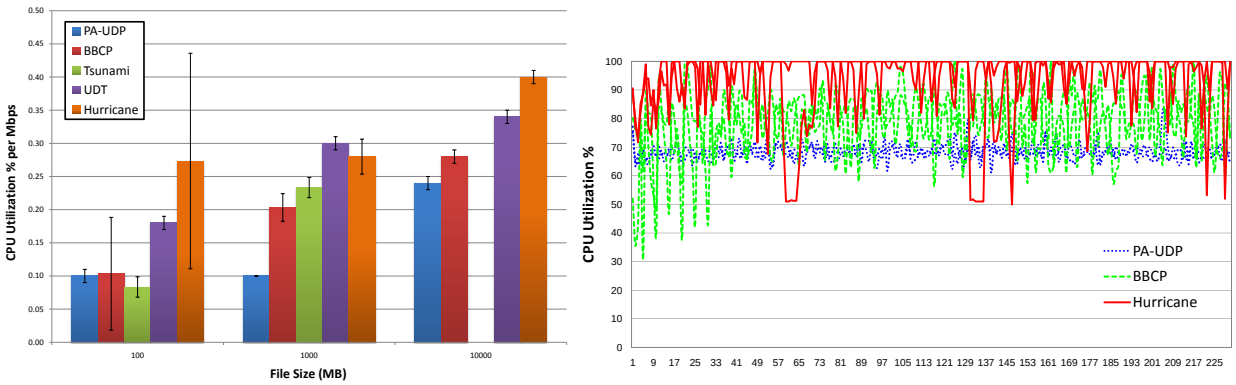


Fig. 11. (a) Percentage CPU utilization per Mbps for three file sizes: 100, 1000, and 10,000 MB. PA-UDP can drive data faster at a consistently lower computational cost. Note that we could not get UDT or Tsunami to successfully complete a 10 GB transfer, so the bars are not shown. (b) A section of a CPU trace for three transfers of a 10 GB file using PA-UDP, Hurricane, and BBCP. PA-UDP not only incurs the lowest CPU utilization, but it is the most stable.

type of flow control essentially have to "discover" the hardware-imposed maximum by running at a unsustainable rate and then reactively curbing throughput when packet loss occurs. In contrast to other high-speed protocols, PA-UDP maintains a more stable and more efficient rate.

A simple CPU utilization average during a transfer would be insufficient to compare the various protocols' computational efficiency, since higher throughputs affect CPU utilization adversely. Thus, a transfer that spends most of its time waiting for the retransmission of lost packets may look more efficient from a CPU utilization perspective though, in fact, it would perform much worse. To alleviate this problem, we introduce a measure of CPU utilization *per units of throughput*. Using this metric, a protocol which incurs high packet loss and spends time idling would be punished, and its computational efficiency would be more accurately reflected. Figure 11a shows this metric compared for several different high-speed protocols at three different file sizes over three different runs each. To obtain the throughput efficiency, the average CPU utilization is divided by the throughput of the transfer. For completeness, we included a popular high-speed TCP-based application, BBCP, as well as the other UDP-based protocols. The results shown are from the receiver, since it is the most computationally burdened. PA-UDP is considerably more efficient than the other protocols, with the discrep-

ancy being most noticeable at 1 GB. The percentage utilization is averaged across both CPU's in our testbed.

To give a more complete picture of PA-UDP's efficiency, Figure 11b shows a CPU utilization trace over a period of time during a 10 GB transfer for the data receiver. Two trials are represented for each of the three applications: PA-UDP, Hurricane, and BBCP. Not only is PA-UDP consistently less computationally expensive than other two protocols during the course of the transfer, but it is also the most stable. Hurricane, for instance, jumps between 50% and 100% CPU utilization during the course of the transfer. We note here also that BBCP, a TCP-based application, outperforms Hurricane, a UDP-based protocol implementation. Though UDP-based protocols typically have less overhead, which is the main impetus for moving from TCP to UDP, the I/O efficiency of a protocol is also very important, and BBCP appears to have better I/O efficiency compared to Hurricane. Again, the CPU utilization is averaged between both processors on the Dell PowerEdge 850.

8.3 Predicted Maxima

To demonstrate how PA-UDP achieves the predicted maximum performance, Table 3 shows the rate-controlled throughputs for various file sizes in relation to the predicted maximum throughput given disk performance over the time of the transfer. Again, a buffer of 750 Megabytes was used at the receiver.

For 400, 800, and 1000 Megabyte transfers, the discrepancy between predicted and real comes from the fact that the transfers were saturating the link's capacity. The rest of the transfers showed that the true throughputs were very close to the predicted maxima. The slight error present can be attributed to the impreciseness of the measuring methods. Nevertheless, it is constructive to see that the transfers are at the predicted maxima given the system characteristics profiled during the transfer.

9 Related Work

High-bandwidth data transport is required for large-scale distributed scientific applications. The default implementations of Transmission Control Protocol (TCP) [30] and User Datagram Protocol (UDP) do not adequately meet these requirements. While several Internet backbone links have been upgraded to OC-192 and 10GigE WAN PHY, end users have not experienced proportional throughput increases. The weekly traffic measurements reported in [41] reveal that most of bulk TCP traffic carrying more than 10MB of data on Internet2 only experiences throughput of 5Mbps or less. For control applications, TCP may result in jittery dynamics on lossy links [37].

Currently there are two approaches to transport protocol design: TCP enhancements and UDP-based transport with non-Additive Increase Multiplicative Decrease (AIMD) control. In the recent years, many changes to TCP have been introduced to improve its performance for high-speed networks [29]. Efforts by Kelly have resulted in a TCP variant called Scalable TCP [32]. High-Speed TCP Low Priority (HSTCP-LP) is a TCP-LP version with an aggressive window increase policy targeted toward high-bandwidth and long-distance networks [33]. The Fast Active-Queue-Management Scalable TCP (FAST) is based on a modification of TCP Vegas [26], [34]. The Explicit Control Protocol (XCP) has a congestion control mechanism designed for networks with a high bandwidth-delay-product (BDP) [31] [45] and requires hardware support in routers. The Stream Control Transmission Protocol (SCTP) is a new standard for robust Internet data transport proposed by the Internet Engineering Task Force [42]. Other efforts in this area are devoted to TCP buffer tuning, which retains the core algorithms of TCP but adjusts the send or receive buffer sizes to enforce supplementary rate control [27], [36], [40].

Transport protocols based on UDP have been developed by using various rate control algorithms. Such works include SABUL/UDT [16], [17], Tsunami [19], Hurricane [39], FRTP [35], and RBUDP [18] (see [20], [28] for an overview). These transport methods are implemented over UDP at the application layer for easy deployment. The main advantage of these protocols is that their efficiency in utilizing the available bandwidth is much higher than that achieved by TCP. On the other hand, these protocols may produce non-TCP-friendly flows and are better suited for dedicated network environments.

PA-UDP falls under the class of reliable UDP based protocols and like the others is implemented at the application layer. PA-UDP differentiates itself from the other high-speed reliable UDP protocols by intelligent buffer management based on dynamic system profiling considering the impact of network, CPU, and disk.

10 Conclusions

The protocol based on the ideas in this paper has shown that transfer protocols designed for high-speed networks should not only rely on good theoretical performance but should also be intimately tied to the system hardware on which they run. Thus, a high-performance protocol should adapt in different environments to ensure maximum performance, and transfer rates should be set appropri-

TABLE 3
Throughputs to predicted maxima

File Size (MB)	Throughput (Mbps)	Predicted Maximum (Mbps) given r(disk)	Average r(disk) (Mbps)
400	965.80	∞	295.29
800	959.60	4401.92	275.12
1000	958.63	1101.48	275.37
2000	447.22	450.10	281.31
3000	370.20	373.41	280.06
4000	329.81	332.51	270.16
5000	325.58	327.48	278.36

ately to proactively curb packet loss. If this relationship is properly understood, optimal transfer rates can be achieved over high-speed, high-latency networks at all times without excessive amounts of user customization and parameter guesswork.

In addition to low packet loss and high throughput, PA-UDP has shown to be computationally efficient in terms of processing power per throughput. The adaptive nature of PA-UDP shows that it can scale computationally, given different hardware constraints. PA-UDP was tested against many other high-speed reliable UDP protocols, and also against BBCP, a high-speed TCP variant. Among all protocols tested, PA-UDP consistently outperformed the other protocols in CPU utilization efficiency.

The algorithms presented in this paper are computationally inexpensive and can be added into existing protocols without much recoding as long as the protocol supports rate control via inter-packet delay. Additionally, these techniques can be used to maximize throughput for bulk transfer on Gigabit LANs where disk performance is a limiting factor. Our preliminary results are very promising, with PA-UDP matching the predicted maximum performance. The prototype code for PA-UDP is available online at http://iweb.tntech.edu/hexb/pa-udp.tgz.

REFERENCES

[1] N. S. V. Rao, W. R. Wing, S. M. Carter, and Q. Wu, "Ultrascience net: network testbed for large-scale science applications," *Communications Magazine, IEEE*, vol. 43, no. 11, pp. S12–S17, 2005.

[2] X. Zheng, M. Veeraraghavan, N. S. V. Rao, Q. Wu, and M. Zhu, "CHEETAH: Circuit-switched High-speed End-to-End Transport Architecture testbed," *IEEE Commun. Mag.*, vol. 43, no. 8, pp. 11–17, Aug. 2005.

[3] On-demand secure circuits and advance reservation system. [Online]. Available: http://www.es.net/oscars

[4] User Controlled LightPath Provisioning, http://phi.badlab.crc.ca/uclp.

[5] Enlightened Computing, www.enlightenedcomputing.org.

[6] Dynamic resource allocation via gmpls optical networks. [Online]. Available: http://dragon.maxgigapop.net

[7] JGN II: Advanced Network Testbed for Research and Development, http://www.jgn.nict.go.jp.

[8] Geant2, http://www.geant2.net.

[9] Hybrid Optical and Packet Infrastructure, http://networks.internet2.edu/hopi.

[10] Z.-L. Zhang, "Decoupling qos control from core routers: A novel bandwidth broker architecture for scalable support of guaranteed services," in *SIGCOMM*, 2000, pp. 71–83.

[11] N. S. V. Rao, Q. Wu, S. Ding, S. M. Carter, W. R. Wing, A. Banerjee, D. Ghosal, and B. Mukherjee, "Control plane for advance bandwidth scheduling in ultra high-speed networks," in *INFOCOM*. IEEE, 2006.

[12] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler, *Linux Network Architecture*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.

[13] S. Floyd, "RFC 2914: Congestion control principles," Sep. 2000, category: Best Current Practise. [Online]. Available: ftp://ftp.isi.edu/in-notes/rfc2914.txt

[14] V. Jacobson, R. Braden, and D. Borman, "RFC 2647: Tcp extensions for high performance," United States, 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1323.txt

[15] A. Hanushevsky, "Peer-to-peer computing for secure high performance data cop," Apr. 23 2007. [Online]. Available: http://www.osti.gov/servlets/purl/826702-5UdHlZ/native/

[16] R. L. Grossman, M. Mazzucco, H. Sivakumar, Y. Pan, and Q. Zhang, "Simple available bandwidth utilization library for high-speed wide area networks," *J. Supercomput.*, vol. 34, no. 3, pp. 231–242, 2005

[17] Y. Gu and R. L. Grossman, "Udt: Udp-based data transfer for high-speed wide area networks," *Comput. Networks*, vol. 51, no. 7, pp. 1777–1799, 2007.

[18] E. He, J. Leigh, O. T. Yu, and T. A. DeFanti, "Reliable blast UDP: Predictable high performance bulk data transfer," in *CLUSTER*. IEEE Computer Society, 2002, pp. 317–324. [Online]. Available: http://csdl.computer.org/

[19] M. Meiss. Tsunami: A high-speed rate-controlled protocol for file transfer. [Online]. Available: www.evl.uic.edu/eric/atp/TSUNAMI.pdf/

[20] M. Goutelle, Y. Gu, and E. He, "A survey of transport protocols other than standard tcp," 2004. [Online]. Available: citeseer.ist.psu.edu/he05survey.html

[21] D. Newman, "RFC 2647: Benchmarking terminology for firewall performance," 1999. [Online]. Available: www.ietf.org/rfc/rfc2647.txt

[22] Y. Gu and R. L. Grossman, "Optimizing udp-based protocol implementations." Proceedings of the Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2005), Lyons, France, 2005.

[23] R. L. Grossman, Y. Gu, D. Hanley, X. Hong, and B. Krishnaswamy, "Experimental studies of data transport and data access of earth-science data over networks with high bandwidth delay products," *Computer Networks*, vol. 46, no. 3, pp. 411–421, 2004. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2004.06.016

[24] A. C. Heursch and H. Rzehak, "Rapid reaction linux: Linux with low latency and high timing accuracy," in *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 4–4.

[25] "Low latency: Eliminating application jitter with solaris," White Paper, Sun Microsystems, May 2007.

[26] L.S. Brakmo and S.W. O'Malley. Tcp vegas: new techniques for congestion detection and avoidance. In *SIGCOMM*

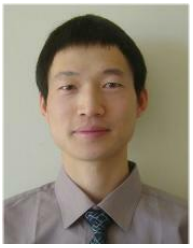
'94 Conference on Communications Architectures and Protocols, pages 24–35, London, United Kingdom, October 1994.
[27] T. Dunigan, M. Mathis, and B. Tierney. A tcp tuning daemon. In *Proceedings of Supercomputing: High-Performance Networking and Computing*, November 2002.
[28] A. Falk, T. Faber, J. Bannister, A. Chien, R. Grossman, and J. Leigh. Transport protocols for high performance. *Communications of the ACM*, 46(11):43–49, 2002.
[29] S. Floyd. Highspeed tcp for large congestion windows. Internet draft, February 2003.
[30] V. Jacobson. Congestion avoidance and control. In *Proc. of SIGCOMM*, page 314-29, 1988.
[31] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for future high-bandwidth-delay product environments. In *Proceedings of ACM SIGCOMM'02*, Pittsburgh, PA, August 19-21 2002. Also see: www.acm.org/sigcomm/sigcomm2002/papers/xcp.pdf.
[32] T. Kelly. Scalable tcp: Improving performance in highspeed wide area networks. In *Workshop on Protocols for Fast Long-Distance Networks*, Februrary 2003.
[33] A. Kuzmanovic, E. Knightly, and R. L. Cottrell. Hstcp-lp: A protocol for low-priority bulk data transfer in high-speed high-rtt networks. In *Second International Workshop on Protocols for Fast Long-Distance Networks*, February 2004.
[34] S.H. Low, L.L. Peterson, and L. Wang. Understanding vegas: a duality model. *Journal of the ACM*, 49(2):207–235, March 2002.
[35] A. P. Mudambi, X. Zheng, and M. Veeraraghavan. A transport protocol for dedicated end-to-end circuits. In *Proc. of IEEE Internation Conference on Communcations*, 2006.
[36] R. Prasad, M. Jain, and C. Dovrolis. Socket buffer auto-sizing for high-performance data transfers. *Journal of Grid Computing*, 1(4):361–376, 2004.
[37] N. S.V. Rao, J. Gao, and L. O. Chua. *Complex Dynamics in Communication Networks*, chapter On dynamics of transport protocols in wide-area internet connections. 2004.
[38] N. Rao, W. Wing, Q. Wu, N. Ghani, Q. Liu, T. Lehman, C. Guok, and E. Dart, "Measurements on hybrid dedicated bandwidth connections," *High-Speed Networks Workshop, 2007*, pp. 41–45, May 2007.
[39] N.S.V. Rao, Q. Wu, S.M. Carter, and W.R. Wing. High-speed dedicated channels and experimental results with hurricane protocol. *Annals of Telecommunications*, 61(1-2):21–45, 2006. to appear.
[40] J. Semke, J. Madhavi, and M. Mathis. Automatic tcp buffer tuning. In *Proceedings of ACM SIGCOMM*, August 1998.
[41] S. Shalunov and B. Teitelbaum. "A weekly version of the Bulk TCP Use and Performance on Internet2". Internet2 netflow: Weekly reports, 2004.
[42] R. Stewart and Q. Xie. Stream control transmission protocol. www.ietf.org/rfc/rfc2960.txt, October 2000. IETF RFC 2960.
[43] S. Floyd, "Highspeed tcp for large congestion windows," 2002. [Online]. Available: citeseer.ist.psu.edu/article/floyd02highspeed.html
[44] T. Kelly, "Scalable tcp: improving performance in high-speed wide area networks," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 83–91, 2003.
[45] Y. Zhang and M. Ahmed, "A control theoretic analysis of XCP," in *INFOCOM*. IEEE, 2005, pp. 2831–2835.
[46] C. Jin, D. X. Wei, S. H. Low, J. J. Bunn, H. D. Choe, J. C. Doyle, H. B. Newman, S. Ravot, S. Singh, F. Paganini, G. Buhrmaster, R. L. Cottrell, O. Martin, and W. chun Feng, "FAST TCP: from theory to experiments," *IEEE Network*, vol. 19, no. 1, pp. 4–11, 2005.

ACKNOWLEDGMENTS

This research was supported in part by the U.S. National Science Foundation under grants OCI-0453438 and CNS-0720617 and a Chinese 973 project under grant number 2004CB318203.

Ben Eckart received the B.S. degree in Computer Science from Tennessee Technological University, Cookeville, in 2008. He is currently a graduate student in Electrical Engineering at Tennessee Technological University in the Storage Technology Architecture Research (STAR) Lab. His research interests include distributed computing, virtualization, fault tolerant systems, and machine learning.

Xubin He received the PhD degree in electrical engineering from University of Rhode Island, USA, in 2002 and both the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1997, respectively. He is currently an associate professor in the Department of Electrical and Computer Engineering at Tennessee Technological University and supervises the Storage Technology Architecture Research (STAR) Lab. His research interests include computer architecture, storage systems, virtualization, and high availability computing. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the TTU Chapter Sigma Xi Research Award in 2005. He is a senior member of the IEEE, a member of the IEEE Computer Society and ASEE.



Qishi Wu received the B.S. degree in remote sensing and GIS from Zhejiang University, China in 1995, the M.S. degree in geomatics from Purdue University in 2000, and the Ph.D. degree in computer science from Louisiana State University in 2003. He was a research fellow in the Computer Science and Mathematics Division at Oak Ridge National Laboratory during 2003-2006. He is currently an Assistant Professor with the Department of Computer Science at University of Memphis. His research interests include computer networks, remote visualization, distributed sensor networks, high performance computing, algorithms, and artificial intelligence.

Changsheng Xie received the BS and MS degrees in Computer Science both from Huazhong University of Science and Technology (HUST), China , in 1982 and 1988, respectively. He is currently a professor in the Department of Computer Engineering at HUST. He is also the director of the Data Storage Systems Laboratory of HUST and the deputy director of the Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture, disk I/O system, networked data storage system and digital media technology. He is the vice chair of the expert committee of Storage Networking Industry Association (SNIA), China.