Experiences in Design and Implementation of a High Performance Transport Protocol

Yunhong Gu, Xinwei Hong, and Robert Grossman

Abstract—**This paper describes our experiences in the development of the UDP-based Data Transport (UDT) protocol, an application level transport protocol used in distributed data intensive applications. The new protocol is motivated by the emergence of wide area high-speed optical networks, in which TCP is often found to fail to utilize the abundant bandwidth.**

UDT demonstrates good efficiency and fairness (including RTT fairness and TCP friendliness) characteristics in high performance computing applications where a small number of bulk sources share the abundant bandwidth. It combines both rate and window control and uses bandwidth estimation to determine the control parameters automatically. This paper presents the rationale behind UDT: how UDT integrates these schemes to support high performance data transfer, why these schemes are used, and what the main issues are in the design and implementation of this high performance transport protocol.

Index Terms—**UDT, transport protocol, data intensive application, design, implementation.**

I. INTRODUCTION

THE need for high performance data transfer services is becoming more and more critical in today's distributed data intensive computing applications, such as remote data analysis and distributed data mining.

Although efficiency (i.e. high throughput) is one of the common design objectives in most network transport protocols, efficiency often decreases as the bandwidth delay product (BDP) increases. The bandwidth delay product is defined as the product of the link capacity and the round trip time (RTT) of a packet . Other considerations, like fairness and stability, sometimes make it more difficult to realize the goal of optimum efficiency. Another factor is that many of today's popular protocols were designed in the era when bandwidth was only counted in bytes per second, so performance was not thoroughly examined in high BDP environments.

Today, the emergence and spread of wide area optical networks has imposed new challenges on transport protocol design. New transport protocols should be able to be gradually deployed on the Internet and they should be friendly to existing protocols, while achieving superior performance.

Implementation also becomes critical to the performance as the network BDP increases. A regular HTTP session may only send several messages per second, and it does not matter that the message processing is delayed for a short time. However, in data intensive applications, the packet arrival speed can be as high as 10^5 packets per second (on modern 1GigE or 10GigE links). The protocol needs to process each event in a limited amount of time and inefficient implementations can lead to packet loss or timeouts.

It is unrealistic to deploy a new protocol in the transport layer or below in a short period of time. An application level solution is often more desirable and can be ported into the lower layer gradually if it proves to be effective. In fact, one of the purposes of the standard UDP protocol is to allow new transport protocols to be built on top of it. For example, the RTP protocol [1] is built on top of UDP and supports streaming multimedia.

We present in this paper our experiences in the design and implementation of a high performance transport protocol for data intensive applications, especially those that are not addressed in previous work on traditional Internet protocols. The protocol, named UDP-based Data Transport (UDT) (a successor to the SABUL protocol [2]), uses an end-to-end control approach and works over UDP. We have implemented UDT as an application level library and released it as open source software [3]. The protocol addresses both efficiency and fairness (including RTT fairness and TCP friendliness).

We believe that UDT makes the following three research contributions: 1) UDT introduces a new congestion control algorithm that increases fairness, enabling multiple UDT flows to coexist over the same path. This is important for parallel data intensive applications. 2) UDT employs an AIMD rate control algorithm that uses a bandwidth estimation technique to determine the best increase parameter for efficiency. From our experiments, this increases the effective throughput of the protocol. Another advantage of the AIMD rate control algorithm is that it is fair to commodity TCP flows. This is important since many data intensive computing applications employ, in part, TCP-based web services. 3) It uses a dynamic window control to reduce loss and oscillations, which is desirable from the application's viewpoint. With

these improvements, we believe that UDT can be effectively used for distributed high performance data intensive computing applications.

In Section 2 we present background information and a brief introduction to SABUL and UDT. The detailed design and implementation issues are discussed in Sections 3 and 4, respectively. Section 5 shows some experimental results of UDT's performance and its advantages and disadvantages compared to other protocols. Section 6 lists some lessons learned in the development of this high performance application level protocol. The paper is concluded in Section 7.

II. BACKGROUND

A. Requirements of High Performance Protocols

TCP works well on the commodity Internet, but has been found to be inefficient and unfair to concurrent flows as bandwidth and delay increase [4, 5, 6, 7]. Its congestion control algorithm needs a very long time to probe the bandwidth and recover from loss in high BDP links. Moreover, the existence of random loss on the physical link, the possible lack of a buffer on routers, and the existence of concurrent bursting flows prevent TCP from utilizing high bandwidth with a single flow. Furthermore, it exhibits a fairness problem for concurrent flows with different round trip times (RTTs) called RTT bias.

The success of TCP is mainly due to its stability and the wide presence of short lived, web-like flows on the Internet. However, the usage of network resources in high performance data intensive applications is quite different from that of traditional Internet applications. First, the data transfer often lasts a very long time at very high speeds. Second, the computation, memory replication, and disk IO at the end hosts can cause bursting packet loss or timeouts in data transfer. Third, distributed applications need cooperation among multiple data connections. Fairness between flows with different start times and network delays is desirable. Finally, in grid computing over high performance networks, the abundant optical bandwidth is usually shared by a small number of bulk sources. The concurrency is much smaller than that on the Internet.

Here we present a simple but typical example application - the streaming join. Suppose that real time data streams coming from a remote machine A and a local machine B are joined at another local machine C with a window-based join algorithm [8]. Also, assume that the two data streams are composed of records of the same size. Fig. 1 illustrates the network topology.
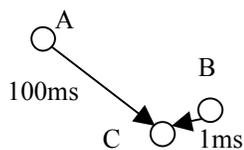

Fig. 1. A streaming join example. The two data streams from A and B are sent to C and joined there. The RTT between A and C is 100ms, whereas it is only 1ms between B and C. Both links share a 1Gb/s bottleneck at C.

We use TCP[1] to transfer the streams, in both an actual network and the NS-2 [9] simulator. The throughputs of the two streams are 3.52 and 863 Mb/s in the actual network, and 80.5 and 807 Mb/s in simulation environments, respectively. The slower stream (AC) limits the join throughput to AC*2, or 7Mb/s in the real network and 160 Mb/s in the simulated environment (out of the 1Gb/s possible maximum throughput). Although applications can sometimes tune the data source rate to alleviate this problem, this needs global knowledge of the network topology and static network environment, which is unrealistic.

The above example shows that TCP does not work well for some distributed data intensive applications. However, there are still traditional applications using TCP in these networks for remote access (e.g., Telnet and FTP) and control messaging (e.g., RPC and web service). New protocols should guarantee TCP friendliness. By ensuring TCP friendliness, those applications that use the new protocol can also run over a public network if necessary.

B. Previous Work

Several transport protocols for high-speed data transfer have been proposed, including NETBLT [10], VMTP [11], and XTP [12]. They all use rate-based congestion control. NETBLT is a block-based bulk transfer protocol designed for long delay links. It does not consider the fairness issue. VMTP is used for message transactions. XTP involves a gateway algorithm, hence it is not an end-to-end approach.

Researchers also have continually worked to improve TCP. TCP SACK [13], which is currently the most supported TCP version, uses selective acknowledgement to alleviate the TCP performance degradation from multiple continuous losses. TCP Westwood [14] is another example designed to recover quickly from packet loss by using bandwidth estimation techniques on the ACK packets. TCP Vegas [15] and FAST TCP [16] use delay instead of loss as the main indication of congestion. In particular, FAST TCP provides an equation-based control algorithm designed to react to network situations more quickly and with higher stability. HighSpeed TCP [17], Scalable TCP [18], and Bic TCP [19] are focusing on fast probing of available bandwidth.

Improvements to TCP variants are often limited by their level of compatibility with standard TCP (they only modify the TCP sender) and still have some important deficiencies, particularly in fairness and automatic parameter tuning.

XCP [4], which adds explicit feedback from routers, is a more radical change to the current transport protocol. It uses an MIMD (multiplicative increase multiplicative decrease) efficiency controller to tune the sending rate according to the current available bandwidth at the bottleneck node. Meanwhile, it still uses an AIMD (additive increase multiplicative decrease) fairness controller to distribute the bandwidth fairly among all concurrent flows.

People in the high performance computing field have been

---

[1] In this paper, when we refer to TCP or standard TCP, it means the most popular TCP version in use today: TCP SACK. In all simulations and experiments, the TCP buffer size is set to at least the BDP.

looking for application level solutions. One of the common solutions is to use parallel TCP [20] connections and tune the TCP parameters, such as window size and number of flows. However, parallel TCP is inflexible because it needs to be tuned on each particular network scenario. Moreover, parallel TCP does not address fairness issues.

For high performance data transfer, experiences in this area have shown that implementation is critical to performance. Researchers have put out some basic implementation guidelines addressing performance. Probably the most famous two are ALF (Application Level Framing [21]) and ILP (Integrated Layer Processing [22]). The basic idea behind these two guidelines is to break down the explicit layered architecture to reach more efficient information processing.

Previously, Leue and Oechslin described a parallel processing scheme for a high-speed networking protocol [23]. However, increases of CPU speed have surpassed increases in network speed, and modern CPUs can fully process the data from networks. Therefore, using multi-processors is not necessary any more.

Memory copy still costs the most in term of CPU time for high-speed data transfer (refer to Table 3). Rodrigues, et al. [24] and Chu [25] have identified this problem and addressed solutions to avoid data replication between kernel space and user space.

There is also literature that describes the overall implementation issues of specified transport protocols. For example, Edwards, et al. describe an implementation of a user level TCP in [26], and Banerjea, et al. present the Tenet protocol design and implementation in [27].

C. From SABUL to UDT

SABUL (Simple Available Bandwidth Utilization Library) first emerged in the year 2000 to support high performance data transfer for high performance data intensive computing applications..

The first prototype of SABUL is a bulk data transfer protocol that sends data block by block over UDP, and sends an acknowledgement after each block is completely received. SABUL uses an MIMD rate-based congestion control algorithm, which tunes the packet sending period[2] according to the current sending rate. The rate control interval (SYN) is constant in order to alleviate the RTT bias problem.

We removed the concept of block to allow applications to send data of any size. Accordingly, the acknowledgment is not triggered on the receipt of a data block, but is based on a constant time interval, which is equivalent to SYN.

For simplicity, SABUL uses TCP to carry control information. However, TCP's own reliability and congestion control mechanism can cause delay of control information in other protocols using TCP that have reliability and congestion control as well. The in-order delivery of control packets is unnecessary in SABUL, but the TCP reordering can delay

control information. During congestion, this delay can even be longer due to TCP's congestion control.

We therefore removed TCP from SABUL and renamed the protocol UDT, or UDP-based Data Transfer protocol.

However, the most important improvement of UDT over SABUL is the congestion control algorithm, which has a similar efficiency but is superior in regard to fairness. UDT's rate control algorithm is AIMD and it uses a bandwidth estimation technique to determine the increase parameter to use to realize efficiency. Meanwhile, UDT adds dynamic window control (SABUL has a static window) to reduce loss and oscillations.

This paper only describes the main design and implementation issues in UDT. Additional details about the protocol can be found in [28].

We make it clear here that in this paper we suppose all packets are the same size (MSS, maximum segment size) and packets or packets per second are used to describe all algorithms and formulas except where it is otherwise explicitly stated. The current UDT software release was modified to use bytes to allow packets of any size.

III. DESIGN ISSUES

In this section, we will discuss how design decisions are made in UDT to meet the requirements of high performance data transfer.

In our opinion, the major challenge is to achieve fairness without compromising efficiency. One of the key issues is how to probe, estimate, or learn the available bandwidth along the link. With this information the sender can tune the sending rate properly. For new protocols, however, TCP friendliness issues present an additional challenge.

Other design choices are also related to the implementation complexity (e.g., computation overhead) and flow shaping (e.g., window vs. rate control).

A. Acknowledging

Acknowledgement is necessary for congestion control and data reliability. In high-speed networks, generating and processing acknowledgements for every received packet may take a substantial amount of time. Meanwhile, acknowledgement itself also consumes some bandwidth.

Selective acknowledgement has demonstrated good performance in reducing control traffic and fast recovery from packet loss in several protocols, such as NETBLT, XTP, and TCP SACK.

UDT uses timer-based selective acknowledgement, which generates an acknowledgement at a fixed interval. This means that the faster the transfer speed, the smaller the ratio of bandwidth consumed by control traffic. Meanwhile, at very low bandwidth, UDT acts like protocols using cumulative acknowledgement.

The ACK interval of UDT is the same as the rate control interval (SYN).

To support this scheme, negative acknowledgement (NAK) is used to explicitly feed back packet loss. NAK is generated

---

[2] We use *packet sending period* instead of *inter-packet time* because in the real world the packet sending consumes various time intervals. Packet sending period is a more accurate measurement to send packets evenly into the network.

once a loss is detected so that the sender can react to congestion as quickly as possible. The loss information (sequence numbers of lost packets) will be resent after an increasing interval if there are timeouts indicating that the retransmission or NAK itself has been lost.

B. Window Control vs. Rate Control

Window control sends data in bursts, and may have sent a large number of packets by the time the sender learns that there is congestion along the link. In addition, the bursting traffic requires that routers have a buffer as large as the BDP but this may be unrealistic on high BDP links.

It has been proposed that packets be sent within the congestion window (say of size cwnd) at average intervals (RTT/cwnd) to alleviate this problem in TCP, which is called TCP pacing. However, using TCP's congestion control algorithm to determine the packet sending period often decreases the throughput and works especially poorly when coexisting with standard TCP, according to Aggarwal, et al. [29].

In high BDP links, the better solution is to tune the packet sending period directly with an efficient rate control mechanism. However, rate control can also lead to another situation of continuous loss: when congestion occurs, the data source may continue to send out data before it receives a loss report or a timeout event. Therefore, a supportive window control should be used together with rate control to specify a threshold on the number of unacknowledged packets.

UDT combines these two mechanisms. Rate control is the major mechanism used to tune the packet sending period, whereas window control is a supportive mechanism used to specify a dynamic threshold that limits the number of unacknowledged packets. This window control is also called flow control because it incorporates a simple flow control mechanism by feeding back the minimum value between the congestion window size and the current available receiver buffer size (UDT's flow control computation is done at the receiver side).

The congestion window size (W) is dynamically updated to the product of packet arrival speed (AS) and the sum of SYN and RTT: W = AS * (SYN + RTT).

The packet arrival speed is calculated through a median filter on the packet arrival intervals. Note that using a mean arrival interval during certain periods does not work because the data sending may not be continuous.

For protocols that acknowledge every data packet, the maximum amount of data packets on the fly is the product of sending speed and RTT. In UDT, however, acknowledgement is triggered every SYN time, so the value should be the product of sending rate and (SYN + RTT). However, we use the receiving speed rather than the sending speed because the former can reflect the network situation more precisely.

C. Control Equations

The most important part of congestion control is the control equations. Chiu and Jain's work shows that AIMD-based algorithms are stable and fair [30]. However, the AIMD

algorithm used in TCP, which increases approximately 1 segment per RTT, does not meet the efficiency objective in high BDP links.

UDT uses a modified AIMD algorithm as follows.

Every SYN time, if there is no NAK, but there are ACKs received in the past SYN time, the number of packets to be increased in the next SYN time (*inc*) is calculated by:

$$inc = \max(10^{\lceil \log_{10} B \rceil - 9}, 1/1500) \times 1500 / MSS \qquad (1)$$

where B is the estimated available bandwidth (Section III.D) in bits per second and MSS is the maximum segmentation size in bytes, which is also the fixed UDT packet size. The constant SYN value in UDT is 0.01 second.

The easiest way to understand (1) is through Table 1, which gives examples of inc, when MSS is 1500 bytes. If MSS is not 1500 bytes, the increments listed above will be corrected by the ratio of 1500/MSS.

TABLE 1
UDT INCREASE PARAMETER COMPUTTAION EXAMPLE

B (Mb/s)	inc (packets)
B ≤ 0.1	0.00067
0.1 < B ≤ 1	0.001
1 < B ≤ 10	0.01
10 < B ≤ 100	0.1
100 < B ≤ 1000	1
…	…

The packet sending period P is then recalculated according to equation (2), where P' is the current packet sending period:

$$SYN / P = SYN / P' + inc \qquad (2)$$

According to formula (1), UDT can recover 90% of the available bandwidth after a single loss in 7.5 seconds or less. Indeed, suppose the link capacity is L ($10^{n-1} < L \leq 10^n$), then to reach 90% of the available bandwidth needs:

$(0.9 * L) / (10^{n-9} * 1500 * 8) * 0.01$

$\leq (0.9 * 10^n) / (10^{n-9} * 1500 * 8) * 0.01 = 750\ SYN = 7.5s.$

In contrast, at 200ms RTT, TCP needs 28 minutes to recover from a single loss to 1Gb/s, or 4 hours 43 minutes to recover to 10Gb/s, etc.

Once a NAK is received, the packet sending period is increased by 1/8:

$$P = P' * 1.125 \qquad (3)$$

To help clear the congestion, the sender stops sending packets in the next SYN time if the largest sequence number in this NAK is greater than the largest sequence number sent when the last decrease occurred.

D. Bandwidth Estimation

To utilize the bandwidth efficiently, it is necessary to have certain information about the link. Allman and Paxson proposed to determine the TCP increase parameter according to the current available bandwidth [31]. Bic TCP uses a binary search to probe the bandwidth, whereas XCP puts the control at the routers, so it knows everything about the link. In contrast, UDT uses a bandwidth estimation technique.

There are several bandwidth estimation algorithms available, which can be roughly classified into end-to-end link capacity estimation and available bandwidth estimation, respectively.

The link capacity is easy to estimate. For example, receiver

based packet pair (RBPP) [32] can estimate it on most of the current high-speed networks. The downside of using link capacity is that when large amounts of concurrent flows are presented, the increase of each flow should be proportional to its fair share according to the number of flows and available bandwidth, not the total link capacity. However, it is difficult to estimate the number of concurrent flows with an end-to-end approach, especially when flows of heterogeneous protocols coexist together.

UDT uses the following algorithm to estimate available bandwidth: Every N packets, the sender does not wait until the next packet sending time but sends out two consecutive packets without inter-packet delay to form a packet pair. The receiver, after receiving the packet pair, will calculate the link capacity and send it back in the next acknowledgement. Here N can be a constant number or a variable generated by a certain function agreed upon by both the sender and the receiver. We use $N = 16$, a number derived from experience.

Suppose L is the estimated link capacity using RBPP, C is the current sending speed, and d is the rate decrease factor, which is $1/9$ (= $1 - 1/1.125$, see formula (3)). For any flow, when the current sending rate is greater than it was when the last decrease occurred, the estimated bandwidth is $L - C$. After a rate decrease and before the sending rate recovers to the value prior to the decrease, the estimated value is min$\{L \times d, L - C\}$. The rationale of $L \times d$ is that all flows decrease their sending rate by d on a loss event (assuming they all experience the congestion), so the surplus bandwidth is $L \times d$. Note that L can be less than C, and in such a case, a minimum positive value should be used (to adapt to formula (1)).

According to this bandwidth estimation and formula (1), in networks with a single bottleneck (which means that all concurrent flows have an approximately equivalent L), flows with higher sending rates cannot increase faster. Therefore, in this case UDT will converge to fairness, even faster than the AIMD algorithm in standard TCP[3]. Fig. 2 shows Jain's fairness index [33] for both UDT and TCP (a larger index means fairer and 1 is the ideal fairness).
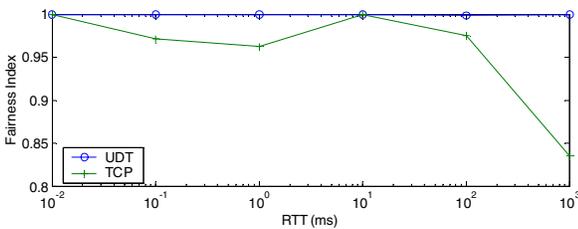

Fig 2. Fairness index of UDT and TCP. This simulation uses 10 concurrent flows running for 100 seconds on a 100 Mb/s link. DropTail queue is used and the queue size is set to max{10, BDP}.

E. Avoiding Congestion Collapse

Congestion collapse refers to the undesirable situation of obtaining low bandwidth utilization as the load increases [34]. It is a common error in the design of transport protocols.

During our development of UDT, we found that one possible form of congestion collapse, specific to high performance data transfer, is from increasing control traffic, which can cost both substantial bandwidth and CPU time. The situation becomes more serious if the processing time is so large that no packets will be sent out because the CPU is busy processing control traffic (and more control traffic may be generated because of timeouts).

The basic mechanism used to avoid this type of congestion collapse is to increase the expiration time after each timeout event for the same packet. For example, the receiver needs to report a lost packet in NAK. If it does not receive any retransmission of the packet in a certain period of time related to RTT, the report will be sent again. However, this timeout period should be increased each time, otherwise, the sender may be blocked by incoming feedback if the processing time is longer than the feedback interval.

This mechanism prevents this type of congestion collapse from happening. However, it is still desirable to reduce the processing time for any single packet in the implementation.

Congestion collapse can also arise from a sole rate control algorithm without any mechanism to stop packet sending until a timeout event occurs if the sender has not received any acknowledgements. The sender may continue sending more packets, which will worsen the congestion and cause further control packets to be discarded. The flow control of UDT is used to avoid this kind of congestion collapse, as stated in Section III.B.

F. Stability

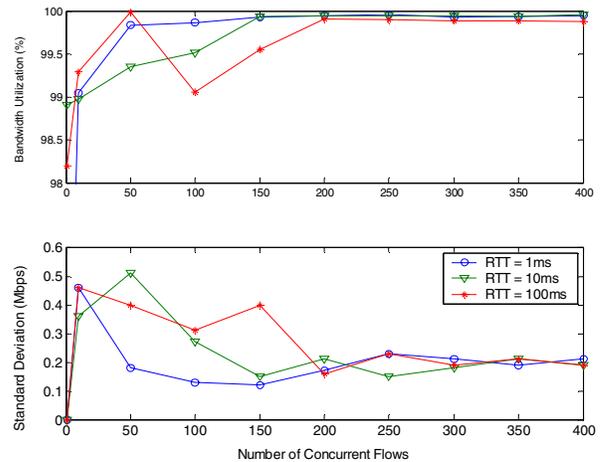The tradeoff of this design is that it does not support very

Fig. 3. Relationship between UDT Performance and number of parallel flows. The figure shows aggregate multiplexed UDT flows, bandwidth utilization and standard deviation of per flow performance (in Mb/s). The link has 100 Mb/s capacity and the gateway uses DropTail queue management.

The link capacity estimation scheme of RBPP does not work well in certain cases such as in multi-channel links. This problem can be overcome by using a more complicated packet bunch mechanism [32]. However, RBPP generally works well on high-speed optical links.

If the bandwidth is underestimated by RBPP, UDT may become more conservative. If the bandwidth is overestimated,

---

[3] On multi-bottleneck topologies, a UDT flow can reach at least half of its max-min fair share. This is the functionality of the logarithm smoothing filter in formula (1). Due to space limitations, the proof is omitted in this paper.

UDT may become more aggressive. However, there is an upper limit to this aggressiveness: suppose the estimated bandwidth is infinitely large, then UDT turns into a sole window-based control protocol, and the congestion window is determined by the packet arrival rate at the receiver side.

The constant control interval may cause problems with stability. As the RTT increases, the increment per RTT also increases and can become very large. However, in this situation the flow window will limit packet sending and help to maintain stability. That is, although UDT becomes more aggressive as RTT increases (in contrast to protocols with control intervals based on RTT), its stability is maintained.

Fig. 4 shows the stability characteristics of UDT and TCP against RTT (smaller values are more stable, and 0 is the ideal). UDT is more stable than TCP in most cases, except when the RTT is between 1 and 10 ms. The stability of TCP can be affected by queue size, and BDP is an optimal size for TCP's performance.

The stability index [16] is defined as:

$$ S = \frac{1}{n} \sum_{i=1}^{n} \left(\frac{1}{\bar{x}_i} \sqrt{\frac{1}{m-1} \sum_{k=1}^{m} (x_i(k) - \bar{x}_i)^2} \right) $$

where n is the number of concurrent flows; m is the number of throughput samples for each flow; $x_i(k)$ is the k-th sample value of flow i; and $\bar{x}_i$ is the average throughput of flow i.
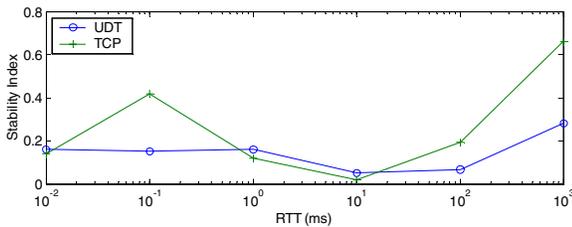

Fig. 4. Stability index of UDT and TCP. This simulation uses 10 concurrent flows running for 100 seconds on a 100Mb/s link. The sample interval is 1 second. DropTail queue is used and the queue size is set to max{10, BDP}.

G. TCP Friendliness

It is hard to define a quantitative TCP friendliness rule for a protocol whose objective is to utilize the extremely high bandwidth that TCP cannot utilize. Although there are models to quantify TCP's throughput (such as Padhye, et al.'s model [5]), using this model to limit the new protocol's throughput is improper because TCP is inefficient in high BDP links and not fair between flows with different RTTs.

Many high-speed protocols, including Scalable TCP and HighSpeed TCP, use the following strategy: in low BDP environments where TCP can work well, new protocols should not be more aggressive than TCP, whereas they can increase aggressiveness as the congestion window exceeds a threshold.

Similarly, UDT uses the constant rate control interval as the functionality of this threshold. Because TCP increases its congestion window approximately 1 segment per RTT, in short RTT links TCP is more aggressive than UDT. As the RTT increases, TCP becomes less aggressive and may become inefficient. In such situations, UDT may overrun TCP and

utilize the bandwidth that TCP fails to occupy.

The value of 0.01 seconds used for SYN relates to the tradeoff between TCP friendliness, efficiency, and stability. For example, if you decrease this value, you increase efficiency, but decrease friendliness and stability. Conversely, if you increase the value of SYN, you increase friendliness and stability but decrease efficiency.

Another element contributing to TCP friendliness is rate-based packet sending. Window-based control is more aggressive because it pours all the data into the network at the same time.[4]

Meanwhile, the smaller increase parameter at low bandwidth (UDT increases at a rate not less than 1 segment per SYN only with available bandwidth higher than 100Mb/s) and the gentler decrease factor have mixed impacts on TCP friendliness.

Suppose there are m UDT and n TCP flows coexisting in the network. With the same network configuration, we start $m+n$ TCP flows separately. The average throughput for the i-th TCP flow in each run is $\bar{x}_i$ and $\bar{y}_i$, respectively. We define the TCP friendliness index as:

$$ T = \frac{1}{n} \sum_{i=1}^{n} \bar{x}_i \left/ \frac{1}{m+n} \sum_{i=1}^{m+n} \bar{y}_i \right. $$

where the denominator is the fair share of TCP.

$T = 1$ is the ideal friendliness; $T > 1$ means UDT is too friendly; and $T < 1$ means UDT overruns TCP. Fig. 5 shows the friendliness index in a simulation of 5 UDT and 10 TCP flows. Even at 100ms RTT, TCP still gets more than 20% of its fair share. (Note that the exceptional lower point at 0.1ms RTT is caused by the impact of queuing.)
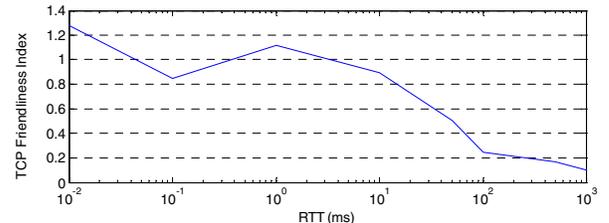

Fig. 5. TCP friendliness index. The simulation starts 5 UDT and 10 TCP flows on a 100Mb/s link. DropTail queue is used and the queue size is set as max{BDP, 100}.

H. Summary

In UDT, rate control is the main congestion control method. The receiver-based packet pair scheme is used to estimate end-to-end link capacity, which will decide the parameters of the AIMD algorithm.

Constant SYN interval and packet pair based bandwidth estimation techniques contribute to the overall efficiency.

The bandwidth estimation scheme and the AIMD control algorithm contribute to intra-protocol fairness. Particularly, the constant SYN leads to RTT fairness. Fig. 6 shows the RTT fairness characteristics of UDT. The two flows with varying

---

[4] For a similar reason, TCP's performance can be heavily affected by queuing, which, however, have little impact on UDT's rate control. In this paper, we omit the discussion of the queuing impacts.

RTTs from 1 to 1000 ms have little difference in their throughput within 10 percent.

TCP friendliness is achieved by the constant SYN interval and rate-based packet sending.

The flow control helps to maintain stability and reduce oscillations. Fig. 7 shows the UDT performance with and without flow control.
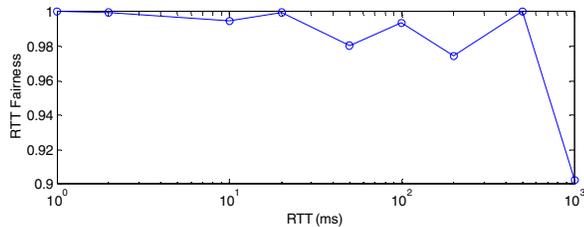

Fig. 6. RTT fairness of UDT. Two concurrent UDT flows are simulated in a network topology similar to that of Fig. 1, with one link having a RTT of 1ms (flow 1) and the other having a RTT from 1ms to 1000ms (flow 2). This figure shows the average throughput of flow 2 over that of flow 1.
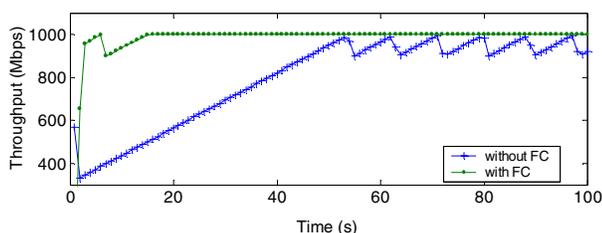

Fig. 7. Comparison between UDT performance with and without flow control. The simulation runs on NS-2. The network configuration is 1Gb/s link capacity, 100ms RTT, DropTail queue with queue size set to BDP.

IV. IMPLEMENTATION ISSUES

The special difficulty in processing Gb/s speed data transfer was noticed by Jain, et al. a decade ago [35]. Although the need for multi-processor or special parallel hardware no longer exists today, the implementation of an application level transport protocol is still sensitive to its performance.

This section will discuss those implementation issues from the software point of view and give practical solutions.

A. Even Distribution of Processing

One of the most common problems of high-speed data transfer is that the generation of control information and application data reading at the receiver side can take a relatively long time, compared to the high packet arrival rate. A poor implementation can cause frequent packet drops, timeouts, and even packet loss avalanches (loss processing that causes even more loss).

One example comes from the Linux implementation of SACK TCP and is identified by Leith [36]. Linux uses a linked list to record unacknowledged packets, which is scanned upon receiving SACK information. In high BDP links, this list is so long that the scanning can cause unnecessary timeouts.

Similarly, because UDT uses explicit loss feedback, the receiver maintains a loss list to record the loss information. Access to the loss list, which contains up to tens of thousands of packets, may take such a long time that the arriving packets overflow the protocol buffer.

To handle this type of problem, it is necessary to evenly distribute the processing into small pieces even if this leads to higher aggregate processing time.

In the following two sections, we will describe how UDT manages the information of in flight packets and handles the memory copy when applications call the *recv* method.

B. Loss Information Management

Lost packets are generally represented as holes in a sliding window, e.g., using a bit array or bit map. However, in high BDP networks, this window is very large and may take significant time to scan. The number of lost packets during congestion can also be very large, and to report the loss will take several packets. In addition, the *insert*, *delete*, and *query* operations to the loss storage need substantial time in a simple array or list data structure.

Considering the fact that loss is often continuous during high congestion, we can use two values to represent a loss event, instead of using all the sequence numbers. For example, if packets from sequence number 200 to 500 are lost, the pair of [200, 500] can be used to record the loss, rather than using 301 numbers. Fig. 8 shows the loss pattern during a heavy congestion over a long haul 1Gb/s link. Each loss event contains up to 3000 lost packets.
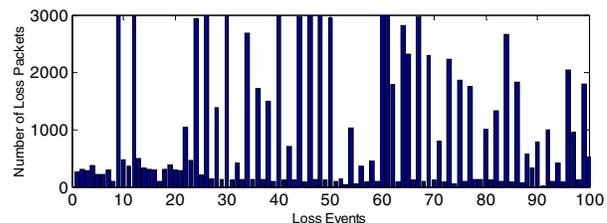

Fig. 8. Loss pattern during congestion. The figure shows the number of lost packet for each loss event during high congestions in a 1Gb/s link with 110ms RTT. The data is obtained by injecting a bursting UDP flow into the network.

Furthermore, with each loss event, loss information is stored in one node. The main access operations are to split and combine the nodes. The practical scanning complexity is much smaller than that of scanning a regular array or list, because there are much fewer loss events than lost packets. Meanwhile, each access takes a similar amount of time.

Fig. 9 shows the algorithm's performance. From Fig. 9 we can see that most of the accesses are finished in 1 microsecond, independent of the number of losses. Detailed data structure and algorithm information can be found in the Appendix.
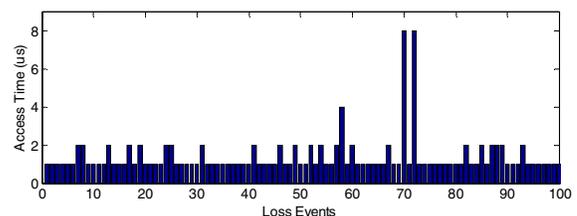

Fig. 9. Access time to the Loss List. The figure shows insert, delete, and query time (in microseconds) to the loss list formed by loss scenario in Fig. 8. Testing is run on a Linux machine with dual 2.4GHz Xeon CPU.

Note that with this loss information storage, no other bit array or map is needed for data reliability.

C. Memory Copy Avoidance

Due to the large amount of data transferred, copy avoidance in high performance transport protocols is much more critical than that in protocols for conventional Internet applications. Here the major motivation is to reduce the packet loss caused when the CPU or the system bus is busy copying large data block, rather than to save CPU time from unnecessary copying.

The copy avoidance between user space and kernel space has been done elsewhere, such as Fast Sockets [24] and Zero Copy TCP [25]. However, at the application level, there is another memory copy that should be avoided, i.e., between the protocol buffer and the application. UDT implements overlapped IO to reduce this copying in a best effort manner.

Fig. 10 illustrates the UDT overlapped IO at the receiver side. The basic idea is to insert the application buffer into the protocol buffer as a logical extension.
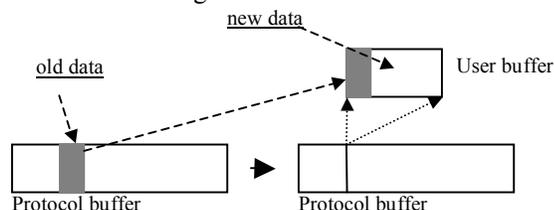

Fig. 10. Demonstration of inserting application buffer into protocol buffer. The left part is the protocol buffer before the application buffer is inserted, whereas the right part is the result status. The gray area presents the data that is already in the buffer. New data will be written directly into the user buffer.

D. Preventing Rate Control from Being Impaired

In UDT, rate control is the major mechanism that manages efficiency and fairness, whereas window control only plays a supportive role. However, there is a potential hazard that window control could become the dominating mechanism and rate control becomes impaired. This situation must be avoided.

This situation may be caused when the packet sending period decreases to below the actual packet sending time. Once this happens, the packet sending is completely controlled by the flow window and the packet sending period may continue to drop. Although most theoretical work assumes an instant packet sending time, this assumption does not hold at high transfer rates. For example, to send a 1500-byte packet out from a 1GigE NIC will take about 13 microseconds, which is comparable to the packet sending period when transfer speed is reaching 1Gb/s.

To avoid this problem, before updating the packet sending period using formula (2) and (3), the value of P' (current packet sending period) should be corrected using the real sending rate.

E. High Precision Timer

Very high precision (microsecond or more precise) timers are not available in most general-purpose operating systems. However, to support rate control at Gb/s speed data transfer, the timing precision should be at least at the microsecond level, and it is better at the CPU frequency level. A simple implementation can use busy waiting to query CPU clock cycles. There is also hardware support that uses interrupt, such as APIC [37] on Intel architecture, as well as software-based approaches such as Soft Timer [38].

Busy waiting, although it may consume 100 percent time on one CPU, may be scheduled to a lower priority so that other jobs are allowed to continue. Due to the blocking manner of UDP sending, higher speeds need less CPU time for the busy waiting implementation.

Many implementations of rate-based protocols use an additional variable of *burst* [10] to control the number of packets that can be sent out continuously. They then sleep for a longer time, hoping that it is long enough to meet the minimum sleep interval that operating systems can provide. However, at high speeds, the value of *burst* can be very large and can make rate control meaningless. For example, Linux provides 0.01 second minimum sleep time, during which time 833 1500-byte packets can be sent out at 1 Gb/s.

Using hardware interrupts may not be a better solution because too frequent interrupts can cause substantial context switches and reduce system performance. Meanwhile, software-based approaches may not guarantee the desired precision. Special network processors may be used in the future.

F. Speculation of Next Packet

If the incoming data can be placed directly into its destination buffer position, the need for a temporary buffer can be eliminated and memory copy is further reduced. The key problem is to guess the sequence number of the next arrival packet, which will decide where to put the incoming data.

Due to the fact that most packets arrive in order, speculation is easy when no loss occurs. During congestion, when the receiver receives a retransmitted packet, it is very likely that the next incoming packet will have the lowest sequence number greater than the current one among those that have not yet been received.

However, because retransmission is only a very small portion of the traffic flow, UDT always speculates that the next packet will be the next consecutive number after the largest sequence number already received. This scheme has the advantage of computation and storage simplicity. The accuracy of the speculation is in reverse proportion to the loss, because 1 loss can cause 2 speculation errors (when it is lost and when the retransmission arrives).

This technique, and the data scattering/gathering technique (i.e., receive or send data that are stored in different memory locations together), need to work together.

G. API Design and Implementation

The API (application programming interface) is an important consideration when implementing a transport protocol. Generally, it is a good practice to comply with the socket semantics. However, due to the special requirements and use scenarios in high performance applications, additional modifications to the original API are necessary.

In the past several years, network programmers have welcomed the new *sendfile* method [39]. It is also an important method in data intensive applications, as these are often involved with disk-network IO. In addition to *sendfile*, a new *recvfile* method is also added, to receive data directly onto disk. The *sendfile/recvfile* interfaces and *send/recv* interfaces are orthogonal.

UDT also implements overlapped IO at both the sender and the receiver sides. Related methods and parameters are added into the API.

Some lower level APIs should be exposed to applications by an upper level protocol. For example, if the transport layer knows whether a packet loss is due to congestion or link error from the network layer, it will be very helpful for congestion control on links with high bit error rates. UDT exposes many UDP interfaces to give applications the most flexibility for configuring their transport facilities.

H. Summary

We implemented UDT using C++ as open source software [3]. The UDT library supports most popular platforms, including Linux, BSD, UNIX, and Windows.

The UDT library is a duplex transport service. Each UDT entity has both a sender and a receiver, which are two threads for packet sending and receiving, respectively. The main thread processes the application request.

The sender is only responsible for sending data packets according to the limit of flow control and rate control. It always sends the lost packets with higher priority, if there are any. The receiver checks the ACK, NAK, SYN, and EXP timers, which will trigger acknowledging, loss information retransmission, rate control, and timeout events, respectively. These four timers are self-clocked in UDT. We use these because high precision is not required in this instance, and the timers use less CPU time. A time bounded UDP receiving call (using RCV_TIMEO option in socket API) is then started to receive new packets, which will be processed if they arrive before the UDP receiving call expires on its timer.

V. EXPERIMENTAL STUDY

In this section, we report on some experimental studies we performed. The experiments were performed on three high-speed testbeds located at Chicago (StarLight), Ottawa (Canarie), and Amsterdam (SARA), respectively. All end hosts run Linux on dual Xeon 2.4GHz CPUs with 1GigE NICs. Between Ottawa and Chicago the network is OC-12 (622 Mb/s) with 16ms RTT; Between Chicago and Amsterdam the network is 1Gb/s with 110ms RTT; Ottawa and Amsterdam are connected via StarLight.

A. Efficiency and Fairness Experiments

In Fig. 11 three UDT flows send data from Chicago to another local machine, Ottawa, and Amsterdam over separate links. The network bandwidth is almost fully utilized in all three cases: 940Mb/s, 580Mb/s and 940Mb/s respectively. In contrast, TCP only reaches about 128 Mb/s from Chicago to

Amsterdam after a thorough tuning for performance.

Fig. 12 shows a similar experiment to that in Fig. 11 but the three flows share the same 1Gb/s link. This experiment demonstrates the fairness property among UDT flows with different bottleneck bandwidths and RTTs. All the three flows reach about 325Mb/s. Using the same configuration, TCP's throughputs are 754Mb/s (to Chicago), 151Mb/s (to Canarie), and 27Mb/s (to Amsterdam), respectively.
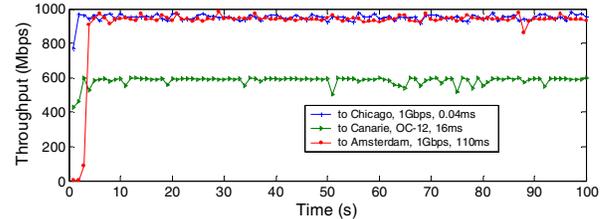

Fig. 11. Single UDT flow performance for several networks. The 3 experiments are done independently in different network environments.
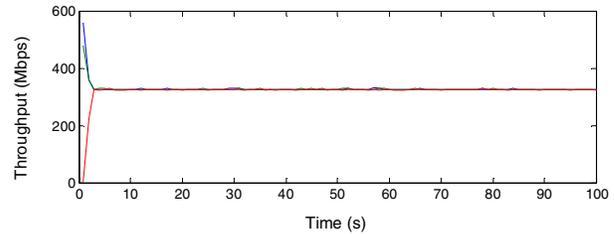

Fig. 12. UDT fairness for several networks. The 3 UDT flows are started simultaneously from Chicago to Ottawa, Amsterdam and another local machine in Chicago.
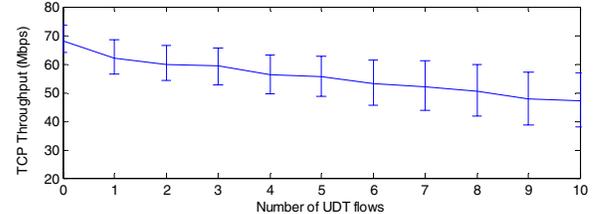

Fig. 13. Aggregate throughput of 500 small TCP flows with different numbers of background UDT flows from 0 to 10.
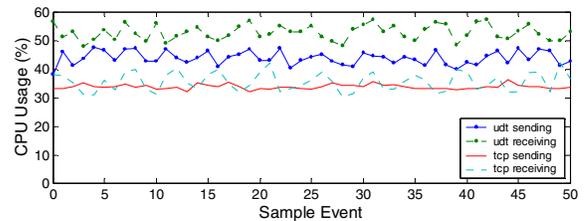

Fig. 14. CPU utilization at sender and receiver sides. The test is between a pair of Linux machines, each having dual 2.4GHz Intel Xeon CPUs. The overall computation ability is 400% (due to hyper-threading). Data is transferred at 970Mb/s between memories.

To examine the TCP friendliness property we set up 500 short-lived TCP flows where each transfers 1MB of data from Chicago to Amsterdam; a varying number of bulk UDT flows are started as background traffics. TCP's throughput should decrease slowly as the number of UDT flows increases. The results are shown in Fig. 13. They decrease from 69Mb/s (without concurrent UDT flows) to 48Mb/s (with 10 UDT concurrent flows).

Fig. 14 shows the CPU utilization of a single UDT flow and a single TCP flow (both sending and receiving) for memory-memory data transfer. The CPU utilization of UDT is slightly higher than that of TCP. UDT averaged 43% (sending) and 52% (receiving). TCP averaged 33% (sending) and 35% receiving. Considering that UDT is implemented at the user level, this performance is acceptable.

B. Comparison with Other Work

In this section we compare UDT with other high-speed protocols using end-to-end congestion controls, including SABUL. We also examine recent developments in TCP to overcome the high BDP problem: Scalable, HighSpeed, FAST, and Bic TCP.

Scalable TCP uses an MIMD approach, whereas HighSpeed TCP has a more moderate increase strategy but also is based on the current congestion window size. According to Chiu and Jain [30], the MIMD algorithm used in Scalable TCP may not converge to fairness equilibrium, which can be seen in Leith and Shorten's experiments [40]. They also show that HighSpeed TCP converges very slowly [40]. According to Xu, et al. [19], both Scalable and HighSpeed TCP can increase the RTT bias of TCP

SABUL's MIMD-like congestion control also converges slowly. Kumazoe, et al. have compared the performance of Scalable TCP, HighSpeed TCP, and SABUL on Japanese Gigabit Networks [41]. It was shown that SABUL maintains a higher throughput with stability.

FAST TCP uses multi-bit information provided by queuing delay to compute the congestion window directly. This method is reported to have better responsiveness, stability, and fairness properties than the 1-bit flag loss indication used in standard, Scalable, and HighSpeed TCP [16]. The major deficiency of FAST TCP is that it has a control parameter (alpha) that needs to be set up manually [16] and its performance can be affected by reverse traffic. Bic TCP has a binary increase scheme to search the available bandwidth efficiently. While reaching high throughput, Bic TCP does not increase the RTT fairness problem of standard TCP [19].

Like these protocols, UDT can also reach a high efficiency. At the same time, it maintains fast convergence to intra-protocol fairness (which is independent of RTT), it is friendly to TCP, and it can tune the control parameter automatically.

C. Effectiveness in Practical Applications

We have incorporated UDT into a streaming join application [8]. Using the same scenario as Fig. 1, the throughput using UDT is about 600 - 800 Mb/s, depending on the data stream contents.

TABLE 2
UDT DISK-DISK PERFORMANCE (ALL DATA IN Mb/s)

To:	Chicago	Ottawa	Amsterdam
From:	disk write 450	disk write 550	disk write 180
Chicago disk read: 600	420	470	145
Ottawa disk read: 800	390	506	138
Amsterdam disk read: 500	377	410	152

Practical applications generally involve more computation and disk IO than simple memory-memory testing. We tested the performance of disk-disk transfer in several environments, which is shown in Table 2. The data shows that UDT can transfer data between disks at nearly the highest speed, which is limited by the disk IO bottleneck.

UDT has been used in many applications, such as file transfer, remote data replication, distributed data mining, and distributed file servers.

VI. LESSONS LEARNED

We have learned many lessons during our 3-year development of SABUL and UDT. We describe in this section several lessons that may be helpful in developing high performance application level protocols.

Using TCP in another transport protocol should be avoided. Using TCP as the control connection accelerated the development of SABUL. However, when we tested SABUL, we found that TCP can have a negative effect on the performance when congestion occurs (Section II.C). More importantly, it is better to design a new protocol based only on a connectionless packet switched layer such that the protocol can be implemented on other layers or networks where TCP may not exist.

Using packet delay as indication of congestion can be hazardous to protocol reliability. The advantages of using delay in congestion control have been recognized by Jain [42]. An increase in the packet delay can indicate congestion earlier, before a real loss occurs. Therefore, it can reduce loss and boost performance. For the same reason, using delay also helps with TCP friendliness, because TCP only uses loss as a congestion indication. TCP-LP [43] is such an example.

UDT once used the method of PCT/PDT described in [44] to detect a packet delay increasing trend and used it as a supportive method to report congestion.

However, packet delays cannot be measured with complete accuracy due to the disturbance of end systems, such as context switch and NIC interrupts coalescing. In addition, packet delay may not always have a high correlation with congestion [45]. Therefore, we no longer use packet delay to indicate congestion.

The obsolete design of UDT that did use packet delay to indicate congestion is friendlier to TCP, but may lead to poor throughputs on certain systems.

Note that this usage of packet delay in transport protocols is not the same as that in FAST TCP, which uses RTT in an equation-based algorithm.

Processing continuous loss is critical to the performance. Continuous loss events can cause multiple decreases in the sending rate, which is lethal to transport protocols depending on packet loss as an indication of congestion. In fact, packet loss can be treated differently and it may be unnecessary to react for all loss events. Currently UDT stops packet sending for SYN time at the first loss event of certain congestions

(Section III.C). Our future work will continue to research this issue.

Knowing how much CPU time each part of the protocol costs helps to make an efficient implementation. One of the code optimization principles is to know which part of the program costs most in term of the resources. All transport protocols have to do a number of similar jobs such as memory copy and network/host bit order conversion. It is useful to know the quantitative CPU time that each job consumes. In UDT, we have done this with Intel VTune performance analyzer [46]. The results, in Table 3, are obtained on a Linux system with dual Intel Xeon 2.4GHz CPU. (Values are the ratio of the CPU utilization of the function unit against total CPU utilization of the UDT entity.)

TABLE 3
CPU UTILIZATION RATIO OF FUNCTIONS IN UDT

Data Sending (%)		Data Receiving (%)	
UDP writing	66.7	UDP reading	90.0
Timing	14.9	Bandwidth/RTT/arrival speed measurement	2.7
Packing data	5.9	Unpacking data	1.9
Processing control packet	5.1	Loss processing	0.6
Application interaction	3.5	Timing	0.4
Other	3.9	Other	4.4

A packet-based scheme is more suitable for high-speed protocols. Packet-based sequencing can reduce the hazard of sequence number wrap collapse. The current 32-bit sequence number in TCP will be wrapped every 17 (= 2^{31} / 125M) seconds at a 1Gb/s transfer rate. It can also reduce the computation overhead of the end systems.

Meanwhile, data that is less than the fixed packet size can still be sent because the UDP socket API can tell the actual size of the packet. In fact, because UDT is supposed to be used for bulk data transfer, most of its data packets can be packed in a fixed size.

In theory, the optimal packet size is the path MTU. Smaller packets cause more overhead of packet headers and packet processing, whereas larger packets have the danger of segmentation collapse [33]. In practice, this is highly affected by the protocol stack implementation of the OS. Fig. 15 shows the relationship between throughput and packet size on a Linux system. The optimal throughout is reached when the MSS is set to 1500 bytes, equivalent to the path MTU. As an example of an exception to this, on Windows XP professional version, we found that the optimal UDP payload size is 1024 bytes, independent of the path MTU.
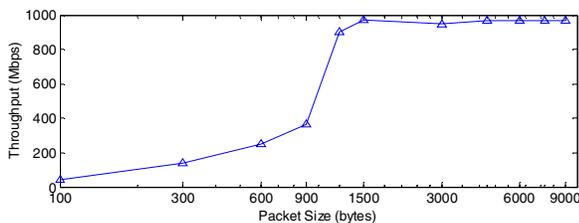

Fig. 15. Relationship between UDT throughput and packet size. The experiments are done using a single UDT flow over a 1Gb/s, 110ms RTT link between a pair of Linux systems. The MTU along this path is 1500 bytes.

VII. CONCLUSIONS

This paper describes a new high performance data transport protocol called UDP-based Data Transport (UDT). UDT is an application level protocol that combines rate-based and window-based congestion control and uses bandwidth estimation techniques to dynamically update the control parameters. This paper also addresses several implementation problems in protocols for high-speed networks. UDT is very suitable for data intensive computing applications where a small number of bulk sources share abundant bandwidth, as has been demonstrated here using both simulations and experiments.

We hope that the experiences described here and the open source implementation of UDT may be useful for future work on transport protocols. First, some ideas that previously appeared mainly in theory and simulations have been tested in UDT, such as the use of bandwidth estimation in transport protocols. Second, the design trade-offs discussed may be useful for other experimental high performance network transport protocols. Third, the next release of UDT is designed so that alternate lost list processing, congestion control algorithms, and bandwidth estimation techniques can be tested.

APPENDIX: LOSS INFORMATION PROCESSING

The data packet sequence number of UDT occupies 32 bits, but only the lowest 31 bits are used, whereas the highest bit is used as a flag. The loss information carried in the loss report is compressed as follows:

If the flag bit of a sequence number is 1, then all the numbers from the current one to the next one are lost; otherwise, the sequence number itself is a lost sequence number.

For example, in the following segment of a loss list:
0x00000003, 0x80000006, 0x0000000F, 0x00000012
the lost sequence numbers are 3, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 18.

The loss list in UDT is a static list (Fig. 16), and each node has two values: the start and the end sequence numbers, which means that all numbers between and including these two numbers are lost. If there is only one single loss, the end number is -1. The location of a node is equal to the position of the head node plus the distance between the start numbers of the two nodes. Continuous losses must be stored in one node. The list is logically circular.

		head				tail			
Link		6				-1		…	
Start		3				7		…	
End		5				-1		…	

Fig. 16. Loss List structure. The figure shows a loss list with loss 3, 4, 5, and 7. Each node on the list has a start value and end value. The list uses a static list data structure.

The major operations on the loss list are *insert*, *delete*, and *query*. Here we only explain how an *insert* is done to this data structure (Fig. 17). The other two algorithms can be obtained

similarly.

Theoretically, the complexity of this algorithm is O(n), where n is the number of loss events, and the time is mainly consumed by searching the prior node (step 4.b). However, according to the locality phenomenon, most searches can be finished in several steps around the near neighbors, so in practice it is fast. *Delete* and *query* operations have the same complexity as *insert*.

In UDT, the operations in the sender and the receiver are slightly different from the above algorithm because more information can be used to simplify them. For example, at the receiver side, insert only happens at the end of the list.

Algorithm: Insert new loss sequence of n (start, end) to loss list L

1. If L is empty, insert (start, end) at position 0; stop.
2. Compute the position of n.start in L (loc) and the offset from the list head (off);
3. If off < 0, insert (start, end) at loc, and loc becomes new head of L.
4. If off > 0:
   a. If L[loc].start = n.start, modify L[loc].end to n.end if n.end > L[loc].end;
   b. Otherwise, search the prior node p, if p and n overlaps or are continuous, modify L[p].end to n.end if n.end > L[p].end, otherwise insert n at loc.
5. If off = 0, modify L[head].end to n.end if n.end > L[head].end.
6. Combine new modified node with its next node if they overlap or are continuous.

Fig. 17. Insert algorithm of loss list.

REFERENCES

[1] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, RTP: *A Transport Protocol for Real-Time Applications*, RFC 1889.
[2] Y. Gu and R. L. Grossman, *SABUL: A Transport Protocol for Grid Computing*, Journal of Grid Computing, to appear.
[3] UDT source release. *http://sourceforge.net/projects/dataspace.*
[4] D. Katabi, M. Hardley, and C. Rohrs, *Internet Congestion Control for Future High Bandwidth-Delay Product Environments*, ACM SIGCOMM '02.
[5] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. *Modeling TCP throughput: a simple model and its empirical validation*. ACM SIGCOMM '98.
[6] Y. Zhang, E. Yan, and S. K. Dao, *A Measurement of TCP over Long-Delay Network*, Proc. of 6th Intl. Conf. on Telecommunication Systems.
[7] W. Feng and P. Tinnakornsrisuphap, *the Failure of TCP in High-Performance Computational Grids*, SC '02.
[8] M. Mazzucco, A. Ananthanarayan, R. Grossman, J. Levera, and G. Bhagavantha Rao, *Merging Multiple Data Streams on Common Keys over High Performance Networks*, SC '02.
[9] NS-2, *http://www.isi.edu/nsnam/ns/.*
[10] D. Clark, M. Lambert, and L. Zhang, *NETBLT: A high throughput transport protocol*, ACM SIGCOMM '87.
[11] D. Cheriton, *VMTP: A transport protocol for the next generation of communication systems*, SIGCOMM '87.
[12] T. Strayer, B. Dempsey, and A. Weaver, *XTP – the Xpress Transfer Protocol.* Addison-Wesley Publishing Company, 1992.
[13] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883, Proposed Standard, July 2000.
[14] M. Gerla, M. Y. Sanadidi, R. Wang, A. Zanella, C. Casetti, and S. Mascolo, *TCP Westwood: Congestion Window Control Using Bandwidth Estimation*, IEEE Globecom 2001, Volume: 3, pp 1698-1702.
[15] L. Brakmo and L. Peterson, *TCP Vegas: End-to-End Congestion Avoidance on a Global Internet*, IEEE Journal on Selected Areas in Communication, Vol 13, No. 8 (October 1995) pages 1465-1480.
[16] C. Jin, D. X. Wei, and S. H. Low, *FAST TCP: motivation, architecture, algorithms, performance*, IEEE Infocom '04.
[17] S. Floyd, *HighSpeed TCP for Large Congestion Windows,* RFC 3649, Experimental Standard, Dec. 2003.

[18] T. Kelly, *Scalable TCP: Improving Performance in Highspeed Wide Area Networks*, Computer Communication Review, Apr. 2003.
[19] L. Xu, K. Harfoush, and I. Rhee, *Binary Increase Congestion Control for Fast Long-Distance Networks*, IEEE Infocom '04.
[20] H. Sivakumar, S. Bailey, R. L. Grossman. *PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks*, SC '00.
[21] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. ACM SIGCOMM '90.
[22] T. Braun and C. Diot, *Protocol Implementation Using Integrated Layer Processing,* ACM SIGCOMM '95.
[23] S. Leue and P. Oechslin, *on parallelizing and optimizing the implementation of communication protocols*. IEEE/ACM Transactions on Networking 4(1): 55-70 (1996).
[24] S. H. Rodrigues, T. E. Anderson, and D. E. Culler, *High-Performance Local Area Communication with Fast Sockets*, USENIX '97, Anaheim, California, January 6-10, 1997.
[25] J. Chu, *Zero-copy TCP in Solaris*. Usenix '96, San Diego, CA, Jan. 1996.
[26] A. Edwards and S. Muir, *Experiences Implementing A High-Performance TCP In User-Space*, ACM SIGCOMM '95.
[27] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. C. Verma, and H. Zhang, *the Tenet real-time protocol suite: Design, implementation, and experiences*, IEEE/ACM Trans. Networking, vol. 4, pp. 1-11, Feb. 1996.
[28] Y. Gu and R. Grossman, *UDT: A Transport Protocol for Data Intensive Applications,* Internet Draft, work in progress.
[29] A. Aggarwal, S. Savage, and T. Anderson, *Understanding the Performance of TCP Pacing*. IEEE Infocom '00.
[30] D. Chiu and R. Jain, *Analysis of the Increase/Decrease Algorithms for Congestion Avoidance in Computer Networks*, Journal of Computer Networks and ISDN, Vol. 17, No. 1, June 1989, pp. 1-14.
[31] M. Allman and V. Paxson. *On Estimating End-to-End Network Path Properties*. ACM SIGCOMM '99.
[32] V. Paxson, *End-to-End Internet Packet Dynamics*, IEEE/ACM Transactions on Networking, Vol.7, No.3, pp. 277-292, June 1999.
[33] R. Jain, *the Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley- Interscience, New York, NY, April 1991.
[34] S. Floyd and K. Fall. *Promoting the use of end-to -end congestion control in the Internet*. IEEE/ACM Transactions on Networking, 7(4): 458-472, 1999.
[35] N. Jain, M. Schawrtz, T. Bashkow, *Transport protocol processing at GBPS rates*, SIGCOMM '90.
[36] D. Leith, *Linux TCP Implementation Issues in High-Speed Networks*, technical report, *http://www.hamilton.ie/net/LinuxHighSpeed.pdf.*
[37] Intel 82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC), *http://www.intel.com/design/chipsets/datashts/290566.htm.*
[38] M. Aron and P. Druschel. *Soft timers: Efficient microsecond software timer support for network processing*. ACM Transactions on Computer Systems 18, 3 (2000), 197- 228.
[39] P. Joubert, R. King, R. Neves, M. Russinovich, and J. Tracey, *High-Performance Memory-Based Web Servers: Kernel and User-Space Performance*, USENIX '01, Boston, Massachusetts, June 2001.
[40] D. J. Leith and R. Shorten, H-TCP Protocol for High-Speed Long Distance Networks, PFLDnet '04, Feb. 2004. *http://dsd.lbl.gov/DIDC/PFLDnet2004/talks/Leith-slides.pdf.*
[41] K. Kumazoe, Y. Hori, M. Tsuru, and Y. Oie, *Transport Protocol for Fast Long Distance Networks: Comparison of Their Performances in JGN*, SAINT '04.
[42] R. Jain, *A Delay Based Approach for Congestion Avoidance in Interconnected Heterogeneous Computer Networks*, ACM SIGCOMM '89.
[43] A. Kuzmanovic and E. W. Knightly, *TCP-LP: A Distributed Algorithm for Low Priority Data Transfer*, IEEE Infocom '03.
[44] M. Jain and C. Dovrolis, *Pathload: A Measurement Tool for End-to-End Available Bandwidth*, Proceedings of Passive and Active Measurements (PAM) 2002 workshop, pages 14-25, Fort Collins, CO.
[45] J. Martin, A. Nilsson, and I. Rhee, *Delay-based congestion avoidance for TCP*. ACM/IEEE Transactions on Networks, June 2003.
[46] Intel VTune Performance Analyzer, *http://www.intel.com/software/products/vtune.*