

UDT: A HIGH PERFORMANCE DATA TRANSPORT PROTOCOL

BY

YUNHONG GU

B.E., Hangzhou Institute of Electronic Engineering, China, 1998

M.E., Beijing University of Aeronautics and Astronautics, China, 2001

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2005

Chicago, Illinois

ACKNOWLEDGMENT

I have been fortunate to have Bob Grossman as my advisor. Bob gave me both the freedom to work on my research interest and the enlightening guidance to help me succeed on my work. He taught me systematically how to conduct and evaluate research work, from which I believe I will benefit greatly in my future career.

I would also like to thank members of my preliminary and dissertation committees: Jason Leigh, Ashfaq Kohkor, Jon Solworth, and Charles Tier. They provided insightful feedback on my PhD project and dissertation.

I owe many thanks to the LAC staff members, who have made my life in UIC much easier. Shirley Connelly has helped me proofread and refine every single paper I wrote during my PhD study, including this dissertation. Shirley, Dave Turkington, Anakany Alveraz, and Jana Wichelecki also helped me through many other non-academic matters, including but not limited to assistantships, personal immigration status, and travel arrangements. Mike Sabala was always ready to help me when I met system and network problems.

Marco Mazzucco instructed my research work during my first year at UIC. He guided me into networking research and let me start a project (SABUL) that led to my PhD research work. I also worked happily with Xinwei Hong in the past four years on networking projects. I have learned a lot from both of them.

Many people have provided valuable feedback on the UDT implementation. Without them there would not be today's fully functional UDT library. In particular, Dave Hanley gave many critiques on the UDT API and through the use of UDT in his projects he has found numerous bugs. Jason Vroustouris and Joe Love spent a summer testing the implementation. Hans Blom and Raj Kettimuthu provided many suggestions in both protocol design and implementation. Kazumi Kumazoe tested UDT on JGN2 and helped promote the use of UDT in the Asia Pacific region.

YG

TABLE OF CONTENTS

1.	INTRODUCTION	1
1.1	Transport Protocol and Congestion Control.....	2
1.2	TCP	4
1.3	Previous Work	5
1.3.1	TCP Modifications	5
1.3.2	XCP.....	6
1.3.3	Application Level Solutions.....	7
1.3.4	SABUL	7
1.4	UDT	8
1.5	Contributions	9
1.6	Organization.....	10
2.	THE UDT PROTOCOL	11
2.1	Overview.....	11
2.2	Design	12
2.2.1	Packet Structures.....	12
2.2.2	Connection Setup and Teardown	13
2.2.3	Reliability Control / Acknowledging	14
2.2.4	Flow Control	15
2.2.5	Congestion Control	16
2.2.6	The Sending and Receiving Algorithms	17
2.3	Implementation	18
2.3.1	Software Architecture	18
2.3.2	Implementation Details	20
2.3.3	User Interface.....	27
2.4	Related Work	27
2.5	Concluding Remarks.....	29
3.	THE UDT CONTROL ALGORITHM	30
3.1	AIMD with Decreasing Increases	30
3.2	The UDT algorithm.....	33
3.3	Fairness of UDT.....	36
3.3.1	Max-min Fairness	36
3.3.2	TCP Friendliness.....	37
3.4	Bandwidth Estimation.....	38
3.5	Dealing with Packet Loss.....	39
3.5.1	Loss Synchronization	40
3.5.2	Noisy Link	41
3.6	Related Work	42
3.7	Concluding Remarks.....	43
4.	PERFORMANCE EVALUATION	45
4.1	Simulations	47

TABLE OF CONTENTS (Continued)

4.1.1	Efficiency, Fairness, and Stability	47
4.1.2	TCP Friendliness.....	54
4.1.3	Impact of Queue Size and Management.....	55
4.1.4	Impact of Link Error	58
4.2	Experimental Studies	60
4.2.1	Efficiency, Fairness, and Stability	61
4.2.2	TCP Friendliness.....	65
4.2.3	Implementation Efficiency.....	66
4.2.4	Performance in Real World Applications.....	68
4.3	Concluding Remarks.....	70
5.	COMPOSABLE UDT	71
5.1	Design.....	72
5.1.1	Overview.....	72
5.1.2	The CCC Interface	73
5.1.3	Inside The UDT Protocol.....	75
5.2	Implementation	77
5.3	Expressiveness	77
5.3.1	Rate-based UDP	78
5.3.2	Standard TCP (TCP NewReno)	78
5.3.3	New TCP Algorithms (Loss-based)	80
5.3.4	New TCP Algorithms (Delay-based)	80
5.3.5	Group Transport Control.....	80
5.3.6	Summary	80
5.4	Performance	82
5.4.1	Similarity	82
5.4.2	CPU Usage Overhead	84
5.5	Related Work	86
5.6	Concluding Remarks.....	87
6.	CONCLUSION.....	89
6.1	Contributions	89
6.1.1	A High Performance Data Transport Protocol and Associated Implementation	89
6.1.2	An Efficient and Fair Congestion Control Algorithm	90
6.1.3	A Configurable Transport Protocol Framework.....	90
6.2	Limitations and Future Research Direction	90
6.2.1	Bandwidth Estimation in the Context of Transport Protocol	90
6.2.2	Implementation Optimization	91
6.2.3	Configurability.....	91
6.3	Final Remarks.....	91
	CITED LITERATURE	93
	VITA	99

LISTS OF TABLES

Table 2-1: UDT increase parameter computation example.	16
Table 3-1: Increase/Decrease and Response Functions of UDT and Various TCP algorithms.....	42
Table 4-1: UDT performance (in mbps) in the scenario of Figure 4-9.	54
Table 4-2: UDT performance (in mbps) of Figure 4-10.	54
Table 4-3: Concurrent UDT flow experiment results.	65
Table 4-4: UDT disk-disk performance (all data in Mb/s).....	70
Table 5-1. Lines of code (LOC) of implementations of TCP algorithms.....	81
Table 5-2: Performance characteristics of TCP and CTCP with various parallel flows.	83
Table 5-3: CPU usage of TCP and CTCP with various parallel flows.	84
Table 5-4: CPU utilization of CTCP against number of parallel flows and ACK intervals.....	85

LISTS OF FIGURES

Figure 1-1: Internet layered architecture.....	2
Figure 1-2: A streaming join example.....	5
Figure 2-1: Layered architecture of UDT.....	11
Figure 2-2: Relationship between UDT sender and receiver.....	12
Figure 2-3: UDT packet header structures.....	13
Figure 2-4: UDT sending algorithm.....	17
Figure 2-5: UDT receiving algorithm.....	18
Figure 2-6: UDT/CCC implementation.....	19
Figure 2-7: Loss pattern during congestion.....	21
Figure 2-8: UDT Loss list structure.....	21
Figure 2-9: Insert algorithm of the UDT loss list.....	22
Figure 2-10: Access time to the loss list.....	22
Figure 2-11: Sender's buffer.....	23
Figure 2-12: Receiver's buffer.....	24
Figure 2-13: Inserting an application buffer into the protocol buffer.....	24
Figure 3-1: Function of increase parameter of DAIMD and several TCP variants.....	32
Figure 3-2: A piecewise $a(x)$ with breakpoints.....	33
Figure 3-3: Sending rate changes over time.....	34
Figure 3-4: Two UDT flows with different link capacities.....	37
Figure 3-5: De-synchronization of UDT control algorithm.....	40
Figure 3-6: Random loss-based decrease algorithm.....	41
Figure 4-1: Simulation network configuration.....	47
Figure 4-2: Bandwidth utilization of a single UDT flow with DropTail queue management.....	48
Figure 4-3: Relationship between UDT Performance and number of parallelism.....	49
Figure 4-4: Fairness of UDT flows with different start time.....	50
Figure 4-5: Network configuration for RTT fairness simulation.....	51
Figure 4-6: RTT independence.....	51
Figure 4-7: Stability index of UDT and TCP.....	52
Figure 4-8: UDT robustness and convergence with rapidly changing background CBR flow.....	53
Figure 4-9: UDT performance in multi-bottleneck topology network.....	53
Figure 4-10: UDT performance in complex topology network.....	54
Figure 4-11: Bandwidth allocation between UDT and TCP.....	55
Figure 4-12: Relationship between UDT throughput and queue size (DropTail).....	56
Figure 4-13: Relationship between TCP friendliness and queue size under different queue management schemes.....	57
Figure 4-14: UDT throughput with RED queue.....	58
Figure 4-15: Impact of link error.....	59
Figure 4-16: Effectiveness of loss resilience algorithm.....	60
Figure 4-17: Experiment network configuration.....	61
Figure 4-18: UDT performance over real high-speed network testbeds.....	62
Figure 4-19: UDT fairness in real networks.....	63
Figure 4-20: Fairness testing configuration.....	63

LISTS OF FIGURES (Continued)

Figure 4-21: Flow start and stop configuration.....	64
Figure 4-22: UDT efficiency and fairness.	64
Figure 4-23: Aggregate throughput of 500 small TCP flows with different numbers of background UDT flows.....	65
Figure 4-24: TCP friendliness in LAN.	66
Figure 4-25: CPU utilization at sender and receiver sides.	67
Figure 4-26: UDT CPU utilization by modules.	68
Figure 4-27: UDT CPU utilization by functionalities.....	68
Figure 4-28: Transferring SDSS data using UDT.....	69
Figure 5-1: UDT/CCC architecture.	73
Figure 5-2: UDT/CCC sending algorithm.	75
Figure 5-3: UDT/CCC receiving algorithm.	76
Figure 5-4: UDT/CCC based protocols.	82

LIST OF ABBREVIATIONS

ACK	Acknowledgment
AIMD	Additive Increase Multiplicative Decrease
API	Application Programming Interface
BDP	Bandwidth-Delay Product
CCC	Configurable Congestion Control
CM	Congestion Manager
GTP	Group Transport Protocol
MIMD	Multiplicative Increase Multiplicative Decrease
NAK	Negative Acknowledgment
RTO	Retransmission Time Out
RTT	Round Trip Time
SYN	SYNchronization interval of the UDT protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UDT	UDP-based Data Transfer Protocol
XCP	eXplicit Control Protocol

SUMMARY

This dissertation proposes a new data transport protocol for bulk data transfer over high-speed wide area networks. This new protocol is called UDT, or UDP-based Data Transfer Protocol. Through extensive simulations and real world experiments, this dissertation shows that UDT satisfies both the objectives of efficiency and fairness (including intra-protocol fairness, RTT fairness, and TCP friendliness). UDT can support data transfer at very high speeds (e.g., 1Gb/s and above) with a single flow while sharing the network resources fairly with concurrent flows. This is not feasible by using existing Internet data transport protocols (e.g., TCP). UDT can be used in the emerging distributed data intensive applications such as grid computing, where a small number of data flows share the abundant optical network bandwidth. UDT is at the application level and it is easy to deploy without the need for administrative privileges.

There are two highly related but orthogonal parts in the work presented by this dissertation: the UDT protocol and its congestion control algorithm. UDT is an application level, end-to-end, unicast, reliable, connection-oriented streaming data transport protocol. The protocol is specially designed for efficient high-speed data transfer. In addition, the UDT protocol is designed to be able to accommodate a large variety of control algorithms such that it can also be used as a protocol framework for the implementation and evaluation of new control algorithms. Finally, UDT's default congestion control algorithm enables UDT to be used in shared network environments while realizing high throughput. This dissertation will introduce the design, implementation, and evaluation of the UDT protocol and its congestion control algorithm.

1. INTRODUCTION

The rapid increase of network bandwidth and the emergence of new distributed applications are the two reciprocal motivations of networking research and development. On the one hand, network bandwidth today has been expanded to 10Gb/s with 40Gb/s emerging, which enables many data intensive applications that were impossible in the past. On the other hand, new applications, such as scientific data distribution, expedite the deployment of high-speed wide-area networks.

Today, national or international high-speed networks have connected most developed regions in the world with fibers [22]. Data can be moved at up to 10 Gb/s among these networks and often at a higher speed inside the networks themselves. For example, in the United States, there are national multi-10Gb/s networks, such as Lambda National Rail, Internet2/Abilene, Teragrid, ESNNet, etc. They can connect to many international networks such as Canada's CA Net, Netherland's SurfNet, UK's UKLight, and Japan's JGN/JGN2.

Meanwhile, we are living in a world of exponentially increasing data. The old way of storing data in disk or tape storage and delivering them by transport vehicles is no longer efficient. In many situations, this old fashioned method of shipping disks with data on them makes it impossible to meet the applications' requirement (e.g., online data analysis and processing).

Researchers in high-energy physics, astronomy, bioinformatics, and other high performance computing areas have started to use these high-speed wide area optical networks to transfer their terabytes of data. We expect that home Internet users will also be able to make use of the high-speed networks in the near future for applications with high-resolution streaming video, for example.

One real world research example is the Sloan Digital Sky Survey (SDSS) project [90], which is mapping in detail one-quarter of the entire sky, determining the positions and brightness of more than 300 million celestial objects. It will also measure the distances to more than a million galaxies and quasars. The data from the SDSS project so far has increased to 2 terabytes and continues to grow. Currently, the 2 terabytes data is being delivered to the Asia-Pacific region, including Australia, Japan, South Korea, and China. Astronomers also want to execute online analysis on multiple datasets stored in geographically distributed locations.

Unfortunately, the high-speed networks have not been smoothly used by these applications. The Transmission Control Protocol (TCP), the *de facto* transport protocol of the Internet, substantially underutilizes network bandwidth over high-speed connections with long delays [17, 27, 50, 101]. For example, a single TCP flow with default parameter settings on Linux 2.4 can only reach about 5 Mb/s over a 1Gb/s link between Chicago and Amsterdam. A new transport protocol as a timely solution is required to address this challenge. The new protocol is expected to be easily deployed and easily integrated with the applications, in addition to utilizing the bandwidth efficiently and fairly.

This dissertation recognizes these emerging application requirements and proposes a new high performance data transport protocol, namely UDT, or UDP-based Data Transfer protocol. This dissertation will describe in detail the design and implementation of UDT, and through extensive theoretical, simulation, and experimental work this dissertation will demonstrate that UDT satisfies those requirements from distributed data intensive applications.

This first chapter provides necessary background information and an overview of the whole dissertation. Section 1.1 gives preliminary information on the Internet transport protocols and network congestion control. Section 1.2 introduces the most typical Internet transport protocol, TCP, and discusses its shortcomings in high-speed wide area networks. Section 1.3 introduces related work on improving TCP. In section 1.4 we will give a brief overview of the UDT protocol. The contributions of this dissertation are listed in section 1.5. Finally, we describe the contents and organization of this dissertation in section 1.6.

1.1 Transport Protocol and Congestion Control

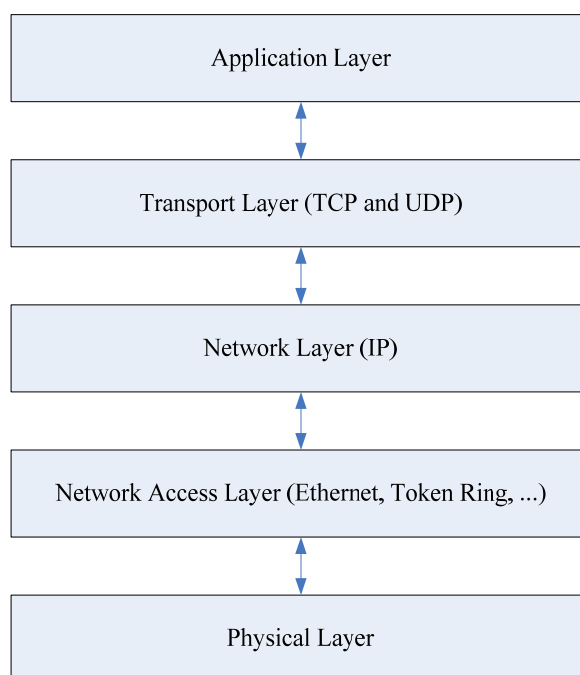


Figure 1-1: Internet layered architecture.

This figure demonstrates the layered architecture of the TCP/IP reference model. Above the physical layer there is the network access layer that provides basic link operation to make use of the physical network. One layer above is the network layer that provides Internet Protocol encapsulation and routing support. In the transport layer are the transport protocols (TCP and UDP) that directly provide data transfer support for applications.

In the layered Internet architecture, the transport layer is the fourth layer above the network layer and right below the application layer (Figure 1-1). Currently there are two transport protocols at this layer: TCP (Transmission Control Protocol) and

UDP (User Datagram Protocol), while there are new protocols emerging, including SCTP [87] and DCCP [54]. TCP is a connection-oriented reliable data streaming protocol, whereas UDP is a connection-less unreliable messaging protocol.

A transport protocol provides various functionalities to the applications, including but not limited to data delivery, data reliability control, and streaming or messaging service. Meanwhile, general-purpose transport protocols have four fundamental objectives that are usually transparent to applications: efficiency, fairness, convergence, and distributedness.

A transport protocol needs to be efficient, or to utilize the available bandwidth as efficient as possible. In detail, to be efficient, a protocol must accomplish the following two tasks in a short time: a) probe the maximum available bandwidth, and b) recover to the maximum speed after a drop in the sending rate due to congestion or packet loss. Meanwhile, after it reaches maximum speed, it should remain at the current state until the network situation changes, i.e., oscillations should be as small as possible.

The network bandwidth is expected to be shared fairly among all concurrent flows. The measurement of fairness can have different standards. The most common one is the max-min fairness, whose objective is to maximize the minimum throughput. The fairness property among all flows belonging to the same protocol is called the intra-protocol fairness of that protocol. In particular, RTT (Round Trip Time) independence is used to describe the special case of fairness over topology with different RTTs, which is not satisfied by TCP. The fairness problem becomes more difficult when heterogeneous protocols coexist. A new transport protocol is required to consider the situation when it coexists with TCP before it is widely deployed on the Internet. The fairness between TCP and the new protocol is called TCP friendliness.

The data sending rate must converge to a unique equilibrium from any starting point, given any specific network situation. It is acceptable that the throughput oscillates around a fixed point because binary feedbacks are usually used to notify changes in the network situation. This is the global stability property of Internet transport protocols.

Finally, because the Internet is such a large loosely coupled system, it is impossible to have a center server to dispatch the bandwidth. Transport protocols must control their data sending rate at the end hosts with or without assistance from the routers that the traffic passing through. The end-to-end principle states that, whenever possible, transport protocols operations should only occur at end hosts. The end-to-end principle greatly increases the system's scalability. Moreover, even at the existence of gateway operators, it is still necessary to have congestion control functionalities at end hosts [33].

Congestion control is the critical component in a transport protocol in order to realize these objectives. The transport protocol adjusts the data sending rate using a certain congestion control algorithm. Network congestion control is usually a feedback system. The feedback can be either explicitly generated from intermediate nodes such as routers; or it can be estimated by packet losses, increase trends in packet delay, or timeout events. Explicit feedback from routers brings more accurate information but also requires higher computation and deployment costs.

The data sending rate can be tuned through either the inter-packet time or the number of outstanding packets. The former method is called rate-based congestion control and the latter is called window-based congestion control. Both of the methods can be applied at the same time. A linear system is often applied in a control scheme to tune these parameters because of its simplicity. The most famous control algorithm is TCP's AIMD algorithm, or additive increase multiplicative decrease.

1.2 TCP

TCP is the most widely used transport protocol and the *de facto* standard Internet data transport protocol. TCP provides reliable data streaming service to applications. It was first proposed during the 70's and since then there have been many updates that are proposed to improve its performance or fix the problems found in previous versions. So far, four major versions have been widely deployed: Tahoe [43], Reno [88], NewReno [31], and SACK [34, 69]. TCP NewReno and TCP SACK are today's most used TCP versions.

TCP's AIMD-based control algorithm [4] increases the sending rate (via congestion window size) approximately 1 segment per RTT, but halves it once there is a loss event. According to this scheme, at 200ms RTT (which is approximately the network distance between US and Japan), after a loss event, over 83,333 RTTs are required for TCP to increase its window for full utilization of 10 Gb/s with 1500-byte packets (166,666 packets per RTT), which is approximately 4.6 hours. In other words, to maintain its maximum utilization of 10 Gb/s (75% of the peak throughput), the loss rate cannot be more than 1 loss event per 18,500,000,000 packets, which is less than the theoretical limit of the network's bit error rates, let alone that we have not considered the impact of the concurrent flows and other system noises.

In fact, the throughput of a TCP flow can be approximately modeled by [73]

$$T = \frac{S}{R\sqrt{\frac{2p}{3}} + t_{RTO} \left(3\sqrt{\frac{3p}{8}} \right) p(1 + 32p^2)} \quad (1-1)$$

where S is the TCP segment size, R is the network RTT, p is the loss rate, and t_{RTO} is the TCP timeout value.

This model indicates that TCP becomes ineffective as the network bandwidth and delay increases [15, 27, 50, 101]. Furthermore, the existence of the RTT in the TCP throughput model means that concurrent flows with different RTTs may have different throughputs, which is also known as RTT bias.

The success of TCP is mainly due to its stability and the widespread presence of short lived, web-like flows on the Internet. However, the usage of network resources in high performance distributed 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, 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 only 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 [71]. Also, assume that the two data streams are composed of records of the same size. Figure 1-2 illustrates the network topology.

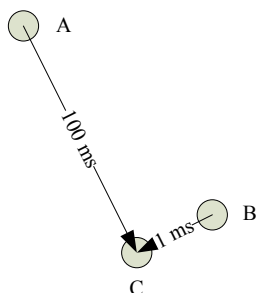


Figure 1-2: A streaming join example.

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 to transfer the streams, in both a real network and the NS-2 (Network Simulator) [106] simulator. The throughputs of the two streams are 3.52 and 863 Mb/s in the real network and 80.5 and 807 Mb/s in the simulation environment, respectively. The slower stream (AC) limits the join throughput to $AC*2$, or 7 Mb/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 in most cases.

1.3 Previous Work

1.3.1 TCP Modifications

Researchers have continually worked to improve TCP. A straightforward approach is to use a larger increase parameter and smaller decrease factor in the AIMD algorithm than those used in the standard TCP algorithm. Scalable TCP [53] and High Speed TCP [29] are the two typical examples of this class.

Scalable TCP increases its sending rate proportional to the current value, whereas it only decreases the sending rate by $1/8$ when there is packet loss. HighSpeed TCP uses logarithmic increase and decreases functions based on the current sending rates. Both of the two TCP variants have better bandwidth utilization, but suffer from serious fairness problems. The MIMD

(multiplicative increase multiplicative decrease) algorithm used in Scalable TCP may not converge to fairness equilibrium, whereas HighSpeed TCP converges very slowly.

BiC TCP [98] uses a similar strategy but proposes a more complicated method to increase the sending rate. Achieving good bandwidth utilization, BiC TCP also has a better fairness characteristic than Scalable and HighSpeed TCP. Unfortunately, none of the above three TCP variants address the RTT bias problem; instead, the problem becomes more serious in these three TCP versions, especially for Scalable TCP and HighSpeed TCP. In addition, BiC TCP has an upper limit on the increase parameter, thus it is less scalable.

TCP Westwood [35] tries to estimate the network situation (available bandwidth) and then tunes the increase parameter accordingly. The estimation is made through the timestamps of acknowledgments. This strategy demonstrates a good idea for using a bandwidth estimation technique in end-to-end congestion control algorithms. However, the Westwood method may be seriously damaged by the impact of ACK compression [99], which can occur at the existence of reverse traffic or NIC interrupt coalescence.

Other recently proposed loss-based TCP control algorithms also include Layered TCP (L-TCP) [11] and Hamilton TCP (H-TCP) [83]. L-TCP uses a similar strategy as HighSpeed TCP by simulating the performance of multiple TCP connections to realize higher bandwidth utilization. H-TCP tunes the increase parameter and the decrease factor according to the elapsed time since the last rate decrease.

Delay-based approaches have also been investigated. The most well known TCP variant of this kind is probably the TCP Vegas algorithm. TCP Vegas compares the current packet delay with the minimum packet delay that has been observed. If the current packet delay is greater, then it means that in some place the queue is filling up, which indicates network congestion. Recently, a new method that follows the Vegas' strategy called FAST TCP was proposed. FAST uses an equation-based approach in order to react to the network situation faster. Although there has been much theoretical work on Vegas and FAST, many of their performance characteristics on real networks are yet to be investigated. In particular, the delay information needed by these algorithms can be heavily affected by reverse traffic. As a consequence, the performance of the two protocols is very vulnerable to the existence of reverse traffic.

1.3.2 XCP

XCP [50], which adds explicit feedback from routers, is a more radical change to the current Internet transport protocol. While those TCP variants mentioned in sub-section 1.3.1 tried many methods to estimate the network situation, XCP takes advantage of explicit information from the routers. As an XCP data packet passes each router, the router calculates an increase parameter or a decrease factor and updates the related information in the data packet header. After the data packet reaches its destination, the receiver sends the information back through acknowledgments.

An XCP router uses an MIMD efficiency controller to tune the aggregate data rate according to the current available bandwidth at the bottleneck node. Meanwhile, it still uses an AIMD fairness controller to distribute the bandwidth fairly among all concurrent flows.

XCP demonstrates very good performance characteristics. However, it suffers more serious deployment problems than the TCP variants because it requires changes in the routers, in addition to the operating systems of end hosts. In addition, recent work showed that gradual deployment (to update the Internet routers gradually) has a significant performance drop [13].

1.3.3 Application Level Solutions

While TCP variants and new protocols such as XCP suffer from deployment difficulties, application level solutions emerge as a favorite timely solution.

A common approach is to use parallel TCP, such as Pockets [84] and GridFTP [2]. Using multiple TCP flows may utilize the network more efficiently, but this is not guaranteed. Performance of parallel TCP relies on many factors from end hosts to networks. For example, the number of parallel flows and the buffer sizes of each flow have significant impact on the performance. The optimal values vary on specific networks and end hosts and are hard to tune. In addition, parallel TCP inherits the RTT fairness problem of TCP.

Using rate-based UDP has also been proposed as a scheme for high performance data transfer to overcome TCP's inefficiency. There is some ongoing work including SABUL [37], FOBS [23], RBUDP [41], FRTP [102], and Hurricane [96]. All of these protocols are designed for private or QoS-enabled networks. They have no congestion control algorithm or have algorithms only for the purpose of high utilization of bandwidth.

1.3.4 SABUL

SABUL (Simple Available Bandwidth Utilization Library) was our prototype for UDT. The experiences obtained from SABUL encouraged us to develop a new protocol with better protocol design and congestion control algorithm.

SABUL is an application level protocol that uses UDP to transfer data and TCP to transfer control information. SABUL has a rate-based congestion control algorithm as well as a reliability control mechanism to provide efficient and reliable data transport service.

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

Later 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. Our further investigation on the SABUL implementation encourages us to re-implement it from scratch with a new protocol design.

Another reason for the redesign is the use of TCP in SABUL. TCP was used for the simplicity of design and implementation. However, TCP's own reliability and congestion control mechanism can cause unnecessary delay of control information in other protocols that have their own 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 be even longer due to TCP's congestion control.

1.4 UDT

UDT is the data transport protocol proposed by this dissertation to support the distributed data intensive applications in wide area high-speed networks. UDT addresses the solution by investigating two orthogonal research problems: 1) the design and implementation of transport protocols with respect to throughput and CPU usage; and, 2) the Internet congestion control algorithm with respect to efficiency, fairness, and stability.

UDT is an application level, end-to-end, unicast, reliable, connection-oriented streaming data transport protocol. The UDT protocol is completely at user space above UDP, i.e., it uses UDP to transfer user data and protocol control information. UDT uses packet-based sequencing to check packet loss and guarantee data reliability. It is specially designed for high-speed bulk data transfer by aiming to remove or reduce the overhead of memory copy, loss information processing, acknowledging, etc. UDT provides reliable streaming data transfer service, similar to TCP.

The UDT protocol supports a large variety of control algorithms. Moreover, it supports congestion control algorithms to be configured at run time, thus each UDT flow can have its own control algorithm and it can change the algorithm at any time.

The built-in (default) UDT congestion control algorithm is proposed to utilize high bandwidth efficiently and fairly. The UDT algorithm uses a loss-based AIMD mechanism. Bandwidth estimation technique is used to optimize its increase parameter dynamically. A random decrease factor is used to remove the negative effect of loss synchronization.

UDT is not used to replace TCP on the Internet where the bottleneck bandwidth is relatively small and there are large amounts of multiplexed short life flows. However, when coexisting with TCP flows, UDT is designed not to occupy more bandwidth than TCP does unless the TCP flows fail to utilize their fair share due to TCP's efficiency problems in high bandwidth-delay product (BDP) environments. This design goal is due to the fact that TCP will still be used in these high BDP networks, and application that uses UDT may sometimes run on public networks.

UDT distinguishes itself from the related work described in Section 1.3 in three major aspects:

- UDT is at the application level. This promotes a much better deployment method than in-kernel protocols including XCP and TCP variants. This also addresses many different research problems, especially in implementations.
- UDT comes with an efficient and fair congestion control algorithm. Therefore, it is a better approach than the other UDP-based protocols that either have no, or very limited congestion control abilities.
- UDT itself is also a protocol framework with configurable congestion control. This feature does not only support application awareness, but also makes UDT a research tool for the evaluation of new congestion control algorithms.

1.5 Contributions

Through the design and implementation of UDT, we recognized and addressed numerous research problems in data transport protocols. Along the way, this dissertation makes the following specific contributions.

- UDT provides a timely and practical solution to the problem of transferring bulk data in high-speed wide area networks. UDT is easily deployable. This is important as there are only four versions of TCP that get widely deployed in the past three decades because of the long time lag of standardization, implementation, and deployment of kernel space protocols. Although there were numerous TCP variants proposed at the same time as UDT was developed, these protocols are not expected to be deployed widely in the near future. In addition, bandwidth estimation techniques are used in the UDT congestion control mechanism such that there is no need for manual tuning of the control parameters.
- Our work systematically investigated the design and implementation issues of high performance data transport protocol at the application level [39]. Although they were often neglected, protocol design and implementation have a significant impact on efficiency. In the UDT project we identified the overhead arising from acknowledgments, loss processing, threading, and memory copy, and proposed appropriate solutions.
- UDT's congestion control algorithm addresses both efficiency and fairness objectives [38]. UDT's algorithm takes approximately a constant time to converge to 90% of available bandwidth. UDT flows are fair to each other, even if they have different RTTs. While UDT is highly efficient, it is not that aggressive. It is friendly to concurrent TCP flows. In addition, the UDT algorithm also solves the loss synchronization problem using a random decreasing method. Finally, UDT can also handle limited non-congestion packet losses.
- UDT's approach is highly scalable. Given that there is enough CPU power, UDT can support up to unlimited bandwidth within terrestrial ranges. The timer-based selective acknowledgment generates a constant number of acknowledgments (ACKs) no matter how fast the data transfer rate is. The congestion control algorithm and the bandwidth estimation technique allow UDT to increase to 90% of the available bandwidth no matter how large it is. In addition, the constant rate control interval helps realize RTT fairness.

- Composable UDT offers more to application development and network research by allowing configurable congestion control algorithms. This feature enables easy development of application or network specific control mechanisms, as well as easy evaluation of new control algorithms.
- Finally, we have developed a productivity quality open source UDT library that can be used in real world applications and research work [108].

1.6 Organization

The details of our UDT research work will be explored in the remainder of this dissertation. Chapter 2 describes the details of UDT protocol design and implementation. We will introduce the design rationale, the details of major protocol functionalities, and critical implementation details. Chapter 3 analyzes the UDT congestion control algorithm by deducing the algorithm from a generalized class of AIMD algorithms. In addition, we will also introduce the bandwidth estimation technique used in UDT and several points in dealing with packet losses. Chapter 4 evaluates UDT's performance using both simulations and real world experiments. We define a set of quantitative measurements including efficiency, fairness, TCP friendliness, and stability, and use them to quantify the results of the simulations and experiments. Chapter 5 introduces the Composable UDT framework. We focus on summarizing of the control events and the updates to be applied into the original UDT design. An evaluation on the expressiveness and performance is also provided. Finally, Chapter 6 concludes the dissertation with the contributions, limitations and future work, and some final remarks.

2. THE UDT PROTOCOL

We present in this chapter how UDT works through its design and implementation, respectively. We will first give an overview of the UDT protocol in section 2.1. Then in section 2.2 we describe in detail the UDT protocol, including packet structures, connection maintenance, packet sequencing, acknowledging, and reliability control. We will also introduce UDT's flow and congestion control in this section, but the analysis of the control algorithm is left to the next chapter. In section 2.3 we will introduce the implementation details of the UDT protocol. Finally, some brief concluding remarks are given in section 2.4.

2.1 Overview

UDT adapts itself into the layered network protocol architecture (Figure 2-1). UDT uses UDP through the socket interface provided by operating systems. Meanwhile, it provides a UDT socket interface to applications. Applications can call the UDT socket API in the same way they call the system socket API.

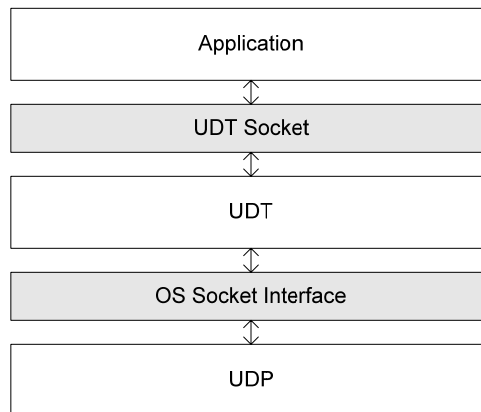


Figure 2-1: Layered architecture of UDT.

In this layered architecture, the UDT layer is completely in user space above the network transport layer of UDP, whereas the UDT layer itself provides transport functionalities to applications.

UDT is a duplex transport protocol. Each UDT entity has two logical parts: the sender and the receiver. The sender sends (and retransmits) application data according to flow control and rate control. The receiver receives both data packets and control packets, and sends out control packets according to the received packets as well.

Figure 2-2 describes the relationship between the UDT sender and the receiver. In Figure 2-2, the UDT entity A sends application data to the UDT entity B. The data is sent from A's sender to B's receiver, whereas the control flow is exchanged between the two receivers.

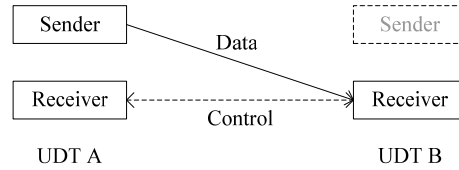


Figure 2-2: Relationship between UDT sender and receiver.

All UDT entities have the same architectures, each having both a sender and receiver. This figure demonstrates the situation when a UDT entity A sends data to another UDT entity B. Data is transferred from A's sender to B's receiver, whereas control information is exchanged between the two receivers.

The receiver is also responsible for triggering and processing all control events, including congestion control and reliability control, and their related mechanisms as well.

UDT uses rate-based congestion control (rate control) and window-based flow control to regulate the outgoing data traffic. Rate control updates the packet-sending period¹ every constant interval, whereas flow control updates the flow window size each time an acknowledgment packet is received.

UDT always tries to pack application data into fixed size packets, unless there is not enough data to be sent. Since UDT is supposed to be used to transfer bulk data streams, we assume that there is only a very small portion of irregular sized packets in a UDT session. The fixed size can be set up by applications and the optimal value is the path MTU (including all packet headers). The actual size of a UDT packet can be known from the UDP header [77].

2.2 Design

2.2.1 Packet Structures

There are two kinds of packets in UDT: the data packets and the control packets. They are distinguished by the 1st bit (flag bit) of the packet header.

A UDT data packet contains a packet-based sequence number and a relative timestamp (which starts counting once the connection is set up) in the resolution of microseconds (Figure 2-3), in addition to the UDP header information. We believe that this information is sufficient for most control algorithms.

¹ In this dissertation we mix the using of inter-packet time and packet sending period, whichever is more proper in the context. Inter-packet time does not include the time needed for sending out a data packet, whereas packet-sending period does include the packet sending time.

Note that UDT's packet-based sequencing with the packet size information provided by UDP is equivalent to TCP's byte-based sequencing and can also support data streaming.

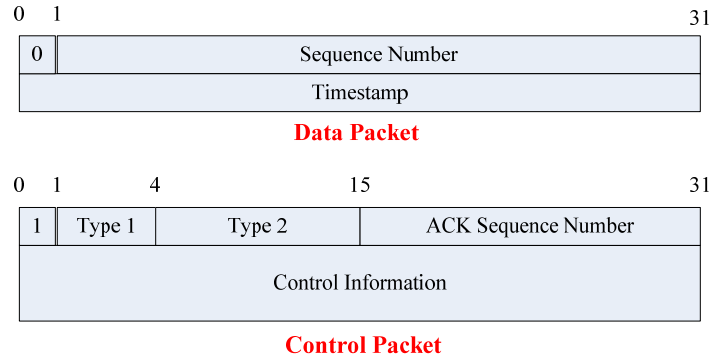


Figure 2-3: UDT packet header structures.

The first bit of the packet header is a flag to indicate if this is a data packet (0) or a control packet (1). Data packets contain a 31-bit sequence number and a 32-bit timestamp. In the control packet header, bit 1- 4 is the packet type (type 1) information. Type 0 – 6 are used by UDT, whereas type 7 is used for user defined types, whose detail type information is put in bit 5 – 15 (type 2). The detailed control information depends on the packet type.

There are 6 types of control packets in UDT and the type information is put in bit field 1 - 3 of the packet header. The contents of the following fields depend on the packet type. The first 32 bits must exist in the packet header, whereas there may be an empty control information field, depending on the packet type.

Particularly, UDT uses sub-sequencing for ACK packet. Each ACK packet is assigned a unique increasing 16-bit sequence number, which is independent of the data packet sequence number. The ACK sequence number uses bits 16 - 31 in the control packet header. The ACK sequence number ranges from 0 to $(2^{16} - 1)$. Bits 16 - 31 are undefined in other control packets.

The 6 types of control packets are: handshake (connection setup information), ACK (acknowledgment), ACK2 (acknowledgment of acknowledgment), NAK (negative acknowledgment, or loss report), keep-alive, and shutdown.

The type 2 field is reserved for users to define their own control packets in the Composable UDT framework (Chapter 5).

2.2.2 Connection Setup and Teardown

One UDT entity starts first as the server, and its peer side (the client) that wants to connect to it will send a handshake packet first. The client should keep on sending the handshake packet every constant interval (the implementation should decide this interval according to the balance between response time and system overhead) until it receives a response handshake from the server or a time-out timer expires.

The handshake packet has the following information:

- UDT version: this value is for compatibility purposes. The current version is 2.
- Initial Sequence Number: It is the initial data packet sequence number that the UDT entity that sends this handshake will use to send out data packets. This is a random value.
- Packet Size: the fixed size of a data packet (including all headers) that the UDT entity will always try to pack.
- Maximum Flow Window Size: The maximum flow window size is the maximum number of packets that the UDT entity can hold in its buffer.

The server, when receiving a handshake packet, compares the packet size with its own value and sets its own value as the smaller one. The result value is also sent back to the client by a response handshake packet, together with the server's version, initial sequence number, and maximum flow window size. The server is ready for sending/receiving data right after this step. However, it must send back a response packet as long as it receives any further handshakes from the same client, in case the client does not receive the previous response.

The client can start sending/receiving data once it gets a response handshake packet from the server. Further response handshake messages, if any are received, should be omitted.

If one of the connected UDT entities is being closed, it will send a shutdown message to the peer side. The peer side, after receiving this message, will also be closed. This shutdown message, delivered using UDP, is only sent once and not guaranteed to be received. If the message is not received, the peer side will be closed by a timeout mechanism (through the keep-alive packets).

Keep-alive packets are generated periodically if there is no other data or control packets sent to the peer side. A UDT entity can detect a broken connection if it does not receive any packets in a certain time.

2.2.3 Reliability Control / Acknowledging

Acknowledgment is necessary for congestion control and data reliability. In high-speed networks, generating and processing acknowledgments for every received packet may take a substantial amount of time. Meanwhile, acknowledgment itself also consumes some bandwidth. (This problem is more serious if the gateway queues use packets rather than bytes as the minimum processing unit, which is very common today.) The detailed relationship between performance and frequency of acknowledgment will be further discussed in Chapter 5, section 5.4.

UDT uses timer-based selective acknowledgment, which generates an acknowledgment at a fixed interval, if there are new continuously received data packets. 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 that acknowledge every data packet.

UDT uses ACK sub-sequencing to avoid sending repeated ACKs as well as to calculate RTT. An ACK2 packet is generated each time an ACK is received. When the receiver side gets this ACK2, it learns that the related ACK has reached its destination

and only a larger ACK will be sent later. Furthermore, the UDT receiver can also use the departure time of the ACK and the arrival time of the ACK2 to calculate RTT. (Note that it is not necessary to generate an ACK2 for every ACK packet.)

The ACK interval of UDT is 0.01 seconds, which means that a UDT receiver will generate 1 acknowledgment per 833 packets at 1 Gb/s data transfer speed.

To support this scheme, negative acknowledgment (NAK) is used to explicitly feed back packet loss. NAK is generated 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. By informing the sender of explicit loss information, UDT provides a similar mechanism to TCP SACK, but the NAK packet can bring more information than the TCP SACK field.

2.2.4 Flow Control

Window control sends data in bursts, and may have sent a large number of packets by the time when 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. [1]. Note that TCP has a self-clocking mechanism to regulate packet sending, but this scheme is often compromised by reverse traffic and NIC interrupt coalescence.

In high BDP links, the better solution may be 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 and the NAK packets get lost, 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: $W = AS * (SYN + RTT)$. Here SYN is the constant rate control interval, which is defined as 0.01 seconds in the current protocol specification.

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, acknowledgment is triggered every SYN time, so the value should be the product of sending rate and (SYN + RTT). In addition, we use the receiving speed rather than the sending speed because the former can reflect the network situation more precisely.

2.2.5 Congestion Control

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:

$$inc = \max(10^{\lceil \log_{10} B \rceil - 9}, 1/1500) \times 1500 / MSS \quad (2-1)$$

where B is the estimated available bandwidth in bits per second and MSS is the maximum segmentation size in bytes, which is also the fixed UDT packet size.

The easiest way to understand (1) is through Table 2-1, which gives examples of inc , when MSS is 1500 bytes. If MSS is not 1500 bytes, the increments listed in table 2-1 will be corrected by the ratio of $1500/MSS$.

Table 2-1: UDT increase parameter computation example.

The first column represents the estimated available bandwidth and the second column represents the increase in packets per SYN. While the available bandwidth increases to the next scope of 10's integral power, the increase parameter also increases by 10 times.

B (Mb/s)	inc (packets/SYN)
$B \leq 0.1$	0.00067
$0.1 < B \leq 1$	0.001
$1 < B \leq 10$	0.01
$10 < B \leq 100$	0.1
$100 < B \leq 1000$	1
...	...

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

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

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

$$P = P' * 1.125 \quad (2-3)$$

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.

The UDT congestion control described above is not enabled until the first NAK is received or the flow window size has reached the maximum flow window size. This is the slow start period of the UDT congestion control. During this time the inter-packet time is kept as zero. The initial flow window size is 2 and it is doubled each time an ACK is received. The slow start only happens at the beginning of a UDT connection, and once the above congestion control scheme is enabled, it will not happen again.

2.2.6 *The Sending and Receiving Algorithms*

Since we have described the majority of UDT's functioning mechanisms, we summarize the algorithms used in the sender and the receiver in this sub-section.

Figure 2-4 describes the abstract sending algorithm. In this algorithm, a sender's loss list is a data structure that records the lost data packets indicated by loss reports from the receiver or by sender side timeouts. ACK and NAK are the abbreviations of acknowledgment and loss report (negative acknowledgment), respectively.

-
- 1) If there is no application data to send, sleep until it is activated by the application.
 - 2) Packet sending:
 - a) If the sender's loss list is not empty, remove the first lost sequence number from the list and pack the corresponding packet.
 - b) Otherwise, if the number of unacknowledged packets does not exceed the flow window sizes, pack a new packet.
 - c) Otherwise, wait here until an ACK or NAK is received, or a timeout event occurs. Go to Step 1.
 - 3) Send the new packet out.
 - 4) Wait until the next packet sending time. Go to Step 1.
-

Figure 2-4: UDT sending algorithm.

Step 2.b is the flow control, which limits the number of unacknowledged packets. The limit is equal to either the congestion window size or the flow window size, whichever one is smallest.

Step 2.c implements self-clocking. This step is useful when the burst-sleep method is used to reduce CPU time (Section 2.3.2.5). Note that the timeout in this step is used to break the deadlock when there is no feedback. It is different from the retransmission timeout.

Step 4 is the rate control, which suspends the data sending until the next sending time.

Figure 2-5 describes the receiving algorithm. In this algorithm, the receiver's loss list is a data structure to store the sequence numbers of the lost packets. EXP is the abbreviation for retransmission timeout (expiration).

-
- 1) Query the timers
 - a) If ACK timer is expired and there are new packets to acknowledge, send back an ACK report; otherwise, if the user-defined ACK interval is reached, send back a lightweight ACK report.
 - b) If NAK timer is expired and the receiver's loss list is not empty, send back a NAK report;
 - c) If EXP timer is expired and there are sent but unacknowledged packets, put the sequence numbers of these packets into the sender's loss list;
 - d) Reset the expired timers.
 - 2) Start time bounded UDP receiving. If nothing is received before the UDP timer expires, go to Step 1.
 - 3) If there is no unacknowledged packet, reset the EXP timer. If the received packet is a control packet, process it, and reset EXP timer if it is an ACK or NAK; Go to Step 1.
 - 4) Check packet loss. If there are packet losses, insert the sequence numbers of the lost packets into the receiver's loss list and generate a loss report (NAK).
 - 5) Go to Step 1.
-

Figure 2-5: UDT receiving algorithm.

The receiver uses self-clocked timers to trigger acknowledgment, loss reports, and timeout events (step 1 and 2). This timing mechanism takes advantage of time-bounded UDP receiving using the `SO_RCVTIMEO` option. On systems where `SO_RCVTIMEO` is unavailable, `select` can be used.

As soon as a packet is received, the receiver processes the new packet according to its type (step 4 and 5).

If there is no unacknowledged packet, the receiver simply resets the EXP timer (step 3); otherwise, the EXP timer only resets when it is an ACK or NAK packet (step 4).

Step 5 is to check packet loss. Various loss detection techniques such as robust reordering [100] can be used here.

2.3 Implementation

The special difficulty in processing Gb/s speed data transfer was noticed by Jain, et al. a decade ago [47]. 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. Overheads of memory copies and context switches bring more difficulty for application level implementations.

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

2.3.1 Software Architecture

Figure 2-6 depicts the UDT software architecture. The UDT layer has five function components: the API module, the sender, the receiver, the listener, and the UDP channel, as well as four data components: sender's protocol buffer, receiver's protocol buffer, sender's loss list, and receiver's loss list.

Because UDT is bi-directional, all UDT entities have the same structure. The sender and receiver in Figure 2-6 have the same relationship as that in Figure 2-2.

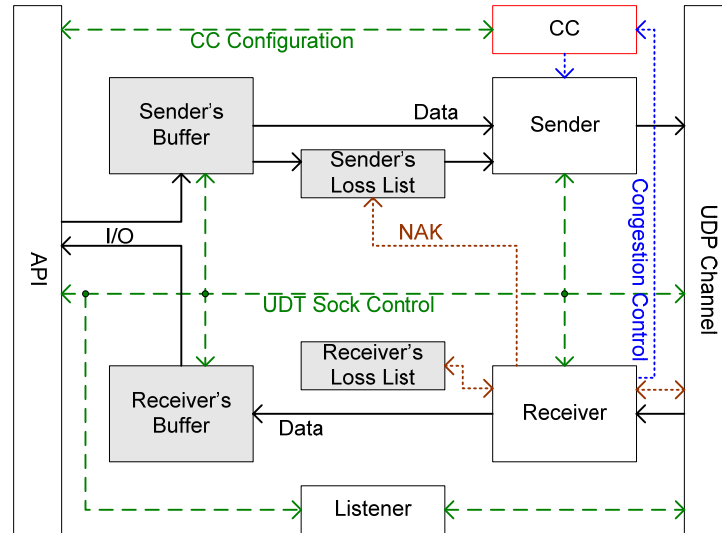


Figure 2-6: UDT/CCC implementation.

The solid line represents the data flow, and the dashed line represents the control flow. The shading blocks (buffers and loss lists) are the four data components, whereas the blank blocks (API, UDP channel, sender, receiver, and listener) are function components.

The API module is responsible for interacting with applications. The data to be sent is passed to the sender's buffer and sent out by the sender into the UDP channel. At the other side of the connection (not shown in this figure but it has the same architecture), the receiver reads data from the UDP channel into the receiver's buffer, reorders the data, and checks packet losses. Applications can read the received data from the receiver's buffer.

The receiver also processes received control information. It will update the sender's loss list (when NAK is received) and the receiver's loss list (when loss is detected). Certain control events will trigger the receiver to update the congestion control module, which is in charge of the sender's packet sending.

The UDT socket options are passed to the sender/receiver (synchronization mode), the buffer management modules (buffer size), the UDP channel (UDP socket option), the listener (backlog), and CC (the congestion control algorithm, which is only used in Composable UDT). Options can also be read from these modules and provided to applications by the API module.

2.3.2 Implementation Details

2.3.2.1 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 (kernel 2.4.18) and was identified by Leith [61]. Linux TCP 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 UDP 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 sub-sections, we will describe how UDT manages the information of in-flight packets and handles the memory copy when applications call the *recv* method.

2.3.2.2 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. Figure 2-7 shows the loss pattern during a heavy congestion over a long haul 1Gb/s link. Each loss event contains up to 3000+ lost packets.

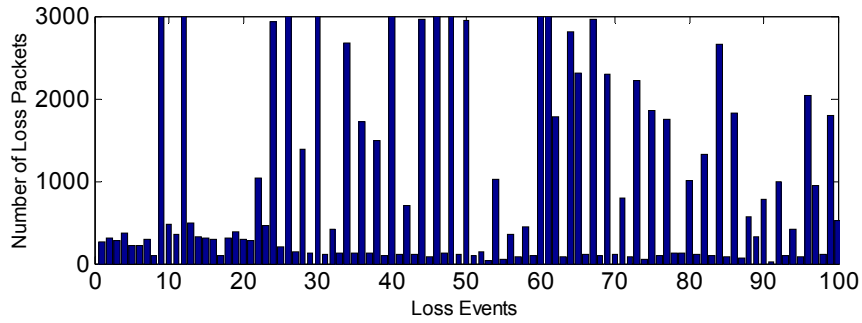


Figure 2-7: Loss pattern during congestion.

The figure shows the number of lost packets for each loss event during high congestion 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.

The loss information carried in the loss report is compressed as follows (the flag bit of the first 32-bit packet header is reused here): 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 (Figure 2-8), 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	...

Figure 2-8: UDT 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 an 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 (Figure 2-9). The other two algorithms can be obtained similarly.

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.

Figure 2-9: Insert algorithm of the UDT loss list.

Theoretically, the complexity of this algorithm is $O(n)$, where n is the number of nodes, 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.

Figure 2-10 shows the algorithm's performance. From Figure 2-10 we can see that most of the accesses are finished in 1 microsecond, independent of the number of losses. Note that with this loss information storage, no other bit array or map is needed for data reliability.

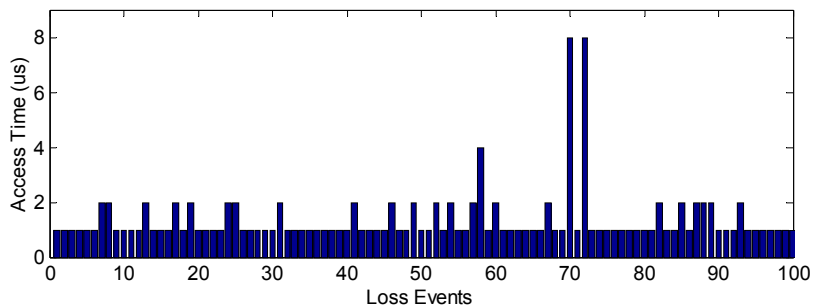


Figure 2-10: Access time to the loss list.

The figure shows the access time (in microseconds) to the loss list formed by loss scenario in Figure 2-7. Testing is run on a Linux machine with dual 2.4GHz Xeon CPU.

2.3.2.3 Memory Copy Avoidance

The buffer management modules are responsible for temporally storing the outgoing or incoming data.

The sender's buffer (Figure 2-11) is a list of outgoing data blocks. When a user calls *send*, the data to be sent is copied into a newly allocated memory block and linked on the list in the order of the send calls (concurrent calls are synchronized). If overlapped IO is enabled, however, the data block is linked to the list directly. After all the data in a block is sent and acknowledged, it is removed from the list and released in non-overlapped IO mode or processed by a user-defined routine (e.g., release the buffer) in overlapped IO mode. Insertion of the new block always occurs at the tail of the list and removal always happens at the head of the list. Note that since the buffer blocks are allocated dynamically, the sender's buffer does not waste any memory space.

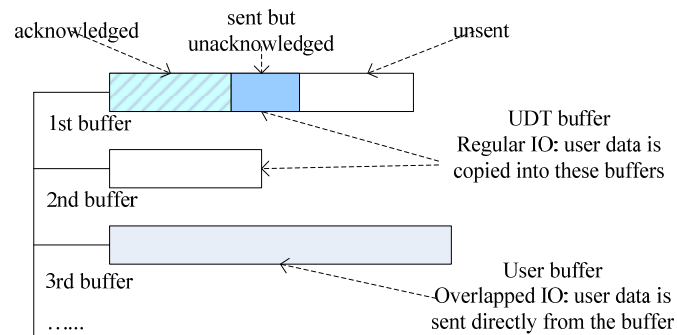


Figure 2-11: Sender's buffer.

The sender's buffer is a list of both protocol allocated buffers (non-overlapped IO) and user buffers (overlapped IO). The UDT sender reads and sends out data in the order that the buffers are linked on the list. There are three regions in the sender's buffer: acknowledged data, sent but unacknowledged data, and unsent data.

The receiver's buffer (Figure 2-12) consists of a list of user buffers and a block of protocol buffers. The list of user buffers is presented when overlapped IO is enabled. In this case, incoming data will be written into the user buffer directly when applicable. In all the other situations, incoming data is written into the protocol buffer and read into the user buffer when *recv* is called. The protocol buffer is dynamically resized according to usage.

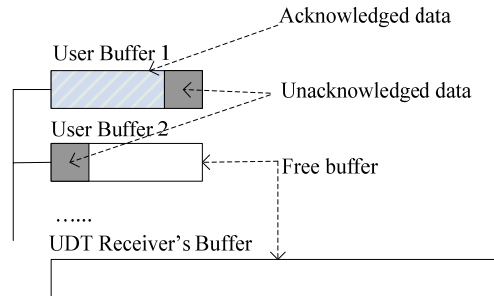


Figure 2-12: Receiver's buffer.

The receiver's buffer has two parts: a linked user buffer list when overlapped IO is used and a protocol buffer. Incoming data is put in the user buffers in the order that they are linked on the list, if there is at least one user buffer and there are spaces for the new data; otherwise the new data is written into the protocol buffer. There are three regions in the receiver's buffer: the acknowledged data, the unacknowledged data, and the free buffer space.

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 motivations include reducing the delay, jitter, timeout, and packet loss when the CPU or the system bus is busy copying a large data block, in addition to saving CPU time from unnecessary copying.

The copy avoidance between user space and kernel space has been done elsewhere, such as Fast Sockets [81] and Zero Copy TCP [19]. 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.

Figure 2-13 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.

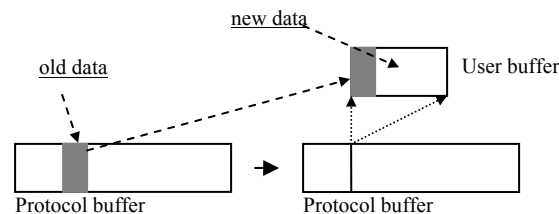


Figure 2-13: Inserting an application buffer into the protocol buffer.

The left part is the protocol buffer before the application buffer is inserted, whereas the right part is the result status. The gray area represents the data already in the buffer. New data will be written directly into the user buffer.

2.3.2.4 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 12 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.

2.3.2.5 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 [104] on Intel architecture, as well as software-based approaches such as Soft Timer [7].

Busy waiting, although it may consume 100 percent of the 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.

An alternative implementation can use an additional variable of *burst* [20] to control the number of packets that can be sent out continuously. The sender then sleeps for a longer time, hoping that it is long enough to meet the minimum sleep interval that operating systems can provide. We call this a burst-sleep method.

The UDT implementation provides both of the two options. In the second option, the UDT sender will sleep for the minimum time allowed by the host system after it sends out a burst of data packets. To reduce the burst size, the UDT sender will be awakened by any possible UDT events, such as the arrival of data or control packets or an application call.

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.

2.3.2.6 *Speculation of Next Packet*

If the incoming data can be placed directly into its destination buffer position using the data scattering/gathering technique (i.e., receive or send data that are stored in different memory locations together), 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 approximately reverse proportional to the packet loss rate, because 1 loss can cause 2 speculation errors (when it is lost and when the retransmission arrives).

2.3.2.7 *Threading*

Three threads are started in a particular UDT entity: the sending thread, the receiving thread, and the listening thread. All the threads are not started at the same time. The listening thread is only started in a listening UDT entity (when it calls *listen*), in which the sender and the receiver are not started. A regular UDT entity (non-listener) will create a receiving thread at the beginning of the connection, whereas the sending thread is started only after the first *send* call. This lazy startup can save system resources since many connections are only used for one-way data transfer.

We could have used one single thread for both sending and receiving (by using the *select* call), but with two separate threads, UDT can take advantages of multiple processors, considering the computation overhead of high speed data transfer and the fact that high end workstations usually have multiple processors.

The sender and the receiver implement the sending and the receiving algorithms in Figure 2-4 and 2-5, respectively.

The sender sends a data packet from the sender's loss list if it is not empty or from the sender's buffer if there is user data to be sent. The sending is controlled by congestion control.

The receiver receives both data and control packets from the UDP channel, whereas it also sends out control information. The in-order data is put into the receiver's buffer, and any detected loss is recorded in the receiver's loss list. It also updates the sender's loss list when a loss report (NAK) is received. Finally, the receiver is responsible for triggering congestion control events and updating control parameters.

The listener accepts connection requests and maintains the accepted sockets in the globally shared UDT descriptors queue. UDT identifies different connections by the random initial sequence number and the *<ip:port>* address.

2.3.3 User Interface

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 [49]. 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 functions 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.

An application can make use of the UDT library in four ways. The library provides a set of C++ API that is very similar to the system socket API. Network programmers can learn it easily and use it in a similar way as using TCP sockets.

When used in applications written by languages other than C/C++, an API wrapper can be used. So far, both Java and Python UDT API wrappers have been developed.

Certain applications have a data transport middleware to make use of multiple transport protocols. In this situation, a new UDT driver can be added to this middleware, and then used by the applications transparently. For example, a UDT XIO driver has been developed so that the library can be used in Globus applications.

Finally, our library also provides a set of C API that has exactly the same semantics as the system socket API. An existing application can be re-compiled and linked against the UDT/CCC C library. In this way, the applications use our library transparently without any changes to the source codes. There is one limitation, though. UDT does not support multi-process models (e.g., using *fork* system call) due to efficiency considerations, so this method does not work if the existing application uses the same sockets in multiple processes.

2.4 Related Work

Several transport protocols for high-speed data transfer have been proposed in the past, including NETBLT [20], VMTP [16], and XTP [89]. 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 [69], which is currently the most supported TCP version, uses selective acknowledgment to alleviate the TCP performance degradation from multiple continuous losses. TCP Westwood [35] is another example designed to recover quickly from packet loss by using bandwidth estimation techniques on the ACK packets. TCP Vegas [14] and FAST TCP [48] 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 [29], Scalable TCP [53], and BiC TCP [98] are focusing on fast probing of available bandwidth.

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

XCP [50], which adds explicit feedback from routers, is a more radical change to the current transport protocol. It uses an MIMD efficiency controller to tune the sending rate according to the current available bandwidth at the bottleneck node. Meanwhile, it still uses an AIMD fairness controller to distribute the bandwidth fairly among all concurrent flows.

People in the high performance computing field have been looking for application level solutions. One of the common solutions is to use parallel TCP [84] 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 [15]). 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 [62]. 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 terms of CPU time for high-speed data transfer. Rodrigues, et al. [81] and Chu [19] 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 [25], and Banerjee, et al. present the Tenet protocol design and implementation in [8].

2.5 Concluding Remarks

High-speed data transfer creates many challenges for the design and implementation of different transport protocols. For example, to achieve 10 Gb/s data transfer speed an end host needs to process 833,333 regular sized (1500 bytes) packets per second. Any additional action on per packet processing can lead to a significant increase in CPU usage, whereas a bursting of CPU usage can further lead to packet loss. Moreover, on long distance links, the number of on flight packets is also huge and requires large data storage to temporally record their information. Access to such data storages is also critical.

This chapter describes the design and implementation of UDT. We presented the details of the UDT protocol and in particular those related to high performance data transfer. This chapter also addresses several implementation problems in protocols for high-speed networks.

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 UDT implementation is designed so that alternate loss list processing, congestion control algorithms, and bandwidth estimation techniques can be tested. These can be reused to reduce the implementation work for related research and development.

3. THE UDT CONTROL ALGORITHM

We explain the rationale of the UDT congestion control algorithm and analyze its performance in this chapter. We generalize a new class of AIMD algorithms in which the size of the additive increases will decrease as the data sending rate increases. We call these algorithms AIMD algorithms with decreasing increases or DAIMD algorithms. We show that DAIMD algorithms are stable and fair, and can be efficient, as well, with proper parameters. The UDT algorithm is a special case of the DAIMD algorithm. In particular, the increase parameter is proportional to the available bandwidth that is estimated by a bandwidth estimation technique. This chapter will also describe the way that UDT deals with packet loss, which is also critical to the performance.

This chapter begins with the introduction of the DAIMD algorithm in section 3.1, and with an analysis of its performance. We introduce the UDT algorithm in section 3.2 as a special case. Section 3.3 further evaluates UDT's fairness property. The bandwidth estimation technique used in UDT will be explained in section 3.4. The packet loss handling algorithms are described in section 3.5. Finally, section 3.6 gives some brief concluding remarks.

3.1 AIMD with Decreasing Increases

We consider a general class of the following AIMD rate control algorithm:

For every rate control interval, if there is no negative feedback from the receiver (loss, increasing delay, etc.), but there are positive feedbacks (acknowledgments), then the packet-sending rate (x) is increased by $\alpha(x)$.

$$x \leftarrow x + \alpha(x) \tag{3-1}$$

$\alpha(x)$ is non-increasing and it approaches 0 as x increases, i.e., $\lim_{x \rightarrow +\infty} \alpha(x) = 0$.

For any negative feedback, the sending rate is decreased by a constant factor β ($0 < \beta < 1$):

$$x \leftarrow (1 - \beta) \cdot x \tag{3-2}$$

Note that formula (1) is based on a fixed control interval, e.g., the network round trip time (RTT). This is different from TCP control, in which every acknowledgment triggers an increase.

By varying $\alpha(x)$, we can get a class of rate control algorithm that we name the DAIMD algorithm, because the additive parameter is decreasing.

If we use the rate control interval as a unit of time, then from time t to $t+1$, the increase to the sending rate from (3-1) is:

$$x(t+1) = x(t) + \alpha(x(t))$$

and the decrease from (2) is:

$$x(t+1) = (1-\beta)^n \cdot x(t)$$

where n is the number of negative feedbacks.

Thus, the net change (contributed by both the increases and the decreases) of the sending rate x is approximated by:

$$x(t+1) - x(t) = P(0) \cdot \alpha(x(t)) - \sum_{i=1}^{x(t-D)} (P(i) \cdot (1 - (1-\beta)^i)) \cdot x(t) \quad (3-3)$$

where $P(i)$ is the probability that there are i packet loss events during the period of $(t, t+1)$, and D is the network round trip delay. In equation (3-3), $x(t-D)$ is the number of packets that can be fed back at period $(t, t+1)$, and $P(i) \cdot (1 - (1-\beta)^i)$ means the possible decrease when i packets are lost.

We define $p(t)$ as the possibility that a data packet gets dropped at time t (i.e., the packet loss rate). To simplify the analysis, we assume that the $p(t)$ is very small and there is, at most, one negative feedback during one unit of time ($P(0) + P(1) = 1$).

$$P(0) = (1 - p(t))^{x(t-D)} \approx 1 - p(t) \cdot x(t-D)$$

$$P(1) = 1 - P(0) = p(t) \cdot x(t-D)$$

In addition, at the stable state, the difference between $x(t)$ and $x(t-D)$ is small and we assume $x(t) = x(t-D)$. Equations (3-3) can be simplified as:

$$\dot{x} = (1 - x(t) \cdot p(t)) \cdot \alpha(x(t)) - x(t) \cdot p(t) \cdot \beta \cdot x(t) \quad (3-4)$$

The differential function (3-4) can be written in the form of:

$$\dot{x} = k(x)(U'(x) - p(t)) \quad (3-5)$$

where $k(x) = x \cdot \alpha(x) + \beta \cdot x^2$ is positive and non-decreasing for any $x (x>0)^2$, and

$$U(x) = \int \frac{\alpha(x)}{x \cdot \alpha(x) + \beta \cdot x^2} dx \quad (3-6)$$

is called the utility function [64] of the above congestion control algorithm.

$U(x)$ is concave because $U'(x)$ is strictly decreasing and hence $U''(x) < 0$. According to Srikant [86] (page 26, theorem 3.4), the congestion control algorithm (3-5) (hence the DAIMD algorithm) is globally asymptotically stable and will converge to an equilibrium state.

² Strictly speaking, $k'(x) = \alpha(x) + x \alpha'(x) + 2\beta x$ may be less than 0, so $k(x)$ may not be strictly non-decreasing. However, because $\alpha(x)$ is non-increasing and it is infinitely close to 0, there exists a const c , such that $k'(x) > 0$ for any $x (x>c)$. Therefore, we can construct a new variable: $y = x - c$, and $k(y)$ is non-decreasing for any $y (y>0)$. We can replace x using y in formula (3-4) - (3-6).

We further show that the equilibrium of the DAIMD algorithm described above satisfies max-min fairness. We use Jain's fairness index (7) to evaluate the max-min fairness among multiple flows.

$$FI = \frac{(\sum x_i)^2}{n \sum x_i^2} \quad (3-7)$$

where n is the number of concurrent flows and x_i is the sending rate of the i th flow at equilibrium. FI is a value between 0 and 1, and $FI = 1$ is perfectly fair. Following the methods used in [18], it can be easily seen that a decrease of x according to (3-2) will not affect the value of FI , but an increase of x according to (3-1) increases FI .

Figure 3-1 illustrates the increase function in TCP Reno, Scalable TCP, HighSpeed TCP, and the DAIMD algorithm. If $\alpha(x) \equiv \alpha$, DAIMD turns into AIMD.

There is one fundamental difference between DAIMD and some TCP variants that use loss as a congestion signal: as the window size becomes larger, both Scalable TCP and HighSpeed TCP increase faster, whereas the increase of BiC TCP may be independent of the absolute sending rate but it is determined by the distance between the current sending rate and a target rate.

In fact, the increment of an XCP flow may also decrease as its sending rate increases, depending on the entering or leaving of coexisting flows, because XCP uses available bandwidth to determine the overall increment. If no flow enters or leaves, this is always true.

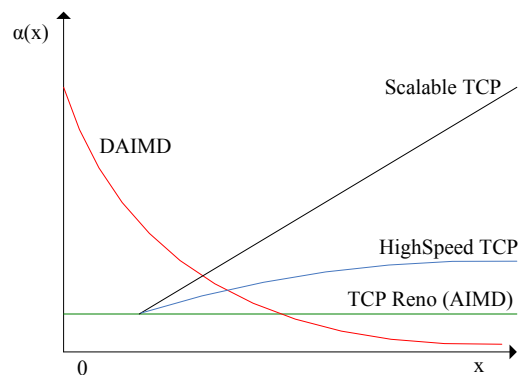


Figure 3-1: Function of increase parameter of DAIMD and several TCP variants.

This figure demonstrates the changes of increase parameters ($\alpha(x)$) of DAIMD, Scalable TCP, HighSpeed TCP, and TCP Reno as the data sending rate (x) increases.

In addition to stability and fairness, the function of $\alpha(x)$ has to be large around $\alpha(0)$ to be efficient and it has to decrease quickly to reduce oscillations. An important special case is provided by an $\alpha(x)$ of the following form (Figure 3-2).

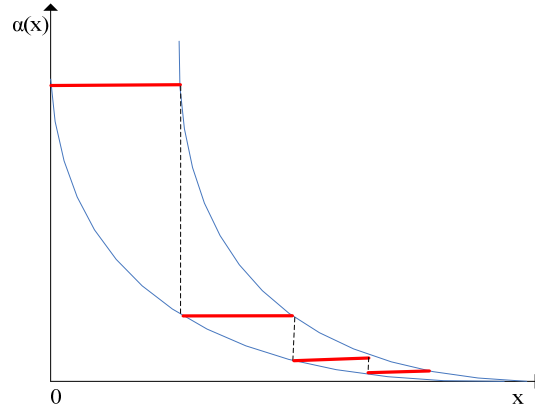


Figure 3-2: A piecewise $a(x)$ with breakpoints.

This figure shows a piecewise increase function, of which the increase parameter is limited between two continuous increase functions.

The first stage in the piecewise function in Figure 3-2 decides how quickly a DAIMD flow can probe the available bandwidth at the beginning, and the length of the stage determines its aggressiveness. The longer the stage is, the more aggressive it will be. Each later stage has a smaller increment as the flow approaches available bandwidth. This will reduce the oscillations at the equilibrium.

Specifically, to achieve efficiency, the increment at each stage should be proportional to the available bandwidth (similar to the mechanism of the XCP efficiency controller [50]).

3.2 The UDT algorithm

UDT adopts this efficiency idea and specifies a piecewise $a(x)$ that is related to the link capacity.

The UDT rate control directly tunes the packet-sending period (T), which indirectly determines the packet-sending rate (x):

$$T \times x = 1$$

We therefore can write the rate control formula in the form of the sending rate.

The fixed rate control interval of UDT is SYN , or the synchronization time interval, which is 0.01 second.

UDT rate control is a special DAIMD algorithm by specifying $a(x)$ as:

$$\alpha(x) = 10^{\lceil \log(L - C(x)) \rceil - \tau} \times \frac{1500}{S} \cdot \frac{1}{SYN} \quad (3-8)$$

In formula (3-8), x has the unit of packets/second. L is the link capacity measured by bits/second. S is the UDT packet size (in terms of IP payload) in bytes. $C(x)$ is a function that converts the unit of the current sending rate x from packets/second to bits/second ($C(x) = x * S * 8$). τ is a protocol parameter, which is 9 in the current protocol specification.

The factor of $(1500/S)$ in function (3-8) is to balance the impact of flows with different packet sizes. UDT treats 1500 bytes as a standard packet size.

Due to the ceiling function in (3-8), the UDT congestion control has multiple stages, as shown in Figure 3-3. UDT increases its sending rate quickly at the beginning and slows down as it is approaching the link capacity. In addition, every stage has the same time span, except for the first stage, if L is not an integer power of 10.

To simplify, we suppose $S = 1500$ and use packets/ SYN as the time unit of $x(t)$. Equation (3-8) can be rewritten as

$$\alpha(x) = 10^{\lceil \log(L - C(x)) \rceil - \tau} \quad (3-9)$$

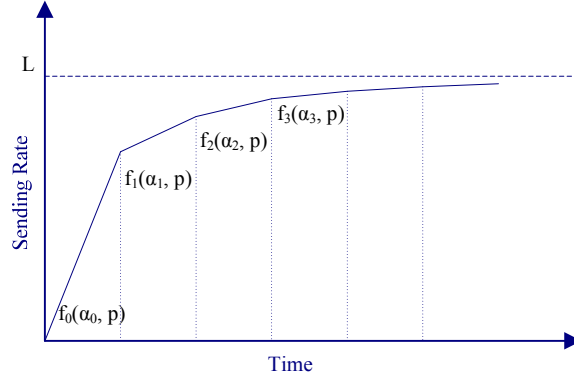


Figure 3-3: Sending rate changes over time.

This figure shows the sending rate of a single DAIMD flow changes over time. This is the situation when there is no non-congestion loss and no other flows in the system; otherwise there will be oscillations in the sending rate.

Thus, UDT implements a piecewise $a(x)$ and according to Section 3.1, it is stable and fair (given that the value of L is the same for all flows. We will discuss this further in Section 3.3). We now discuss its efficiency characteristic.

Suppose in stage k ($k = 0, 1, 2, \dots$), the throughput function is f_k , the increase parameter is α_k , and the loss rate is p . Let e be an integer that satisfies $10^{e-1} < L \leq 10^e$.

According to the rate differential function (3-4), the equilibrium solution ($\dot{x} = 0$) of UDT for any stage k (x_{k^*}) is:

$$x_{k^*} = \frac{1}{2\beta} \left(-\alpha_k + \sqrt{\alpha_k^2 + \frac{4\beta\alpha_k}{p}} \right) \approx \sqrt{\frac{\alpha_k}{\beta \cdot p}} \quad (3-10)$$

The approximation is due to the fact that α_k is very small compared to $1/p$. The result of (3-10) shows that at each stage UDT acts as an AIMD control (the response function is proportional to $p^{-0.5}$), and its increase parameter decreases as the sending rate increases, whereas its decrease factor is a constant.

The increase parameter of each stage decreases by 1/10, and $\alpha_0 = \alpha(0)$, therefore,

$$\alpha_k = 10^{-k} \cdot \alpha_0 = 10^{-k + \lceil \log L \rceil - \tau} \quad (3-11)$$

Recall that

$$\tau = 9,$$

and UDT defines the decrease factor as³

$$\beta = 1/9$$

We finally reach

$$x_k^* = \frac{3}{\sqrt{p}} \cdot 10^{\frac{-k+e-9}{2}} \quad (3-12)$$

Note that x_k has units that are measured by packets/*SYN*. Suppose X_k is the throughput function whose units are bits/second, then

$$X_k^* = \frac{1}{SYN} \cdot \frac{1500}{S} \cdot x_k^* \cdot S \cdot 8 = \frac{3.6}{SYN} \cdot \frac{1}{\sqrt{p}} \cdot 10^{\frac{e-k+1}{2}} \quad (3-13)$$

Once the sending rate increases to a certain value such that $(L-C)$ falls into the next class of the power of 10, i.e., $L - C < 10^{e-k-1}$, the UDT congestion control enters the next stage. However, as k increases, the throughput at stable state (x_k^*) decreases, and k will stop increasing when

$$L - \frac{3.6}{SYN} \cdot \frac{1}{\sqrt{p}} \cdot 10^{\frac{e-k+1}{2}} \geq 10^{e-k-1}$$

The minimum k that satisfies the above condition is the stable stage of a UDT flow. (The operator $[op]^+$ is equivalent to $\max\{op, 0\}$.)

$$k^* = \left[\left[e - 1 - 2 \log \left(\sqrt{d^2 + L - d} \right) \right]^+ \right] \quad (3-14)$$

$$d = \frac{18}{SYN \cdot \sqrt{p}}$$

Furthermore, when $p = 0$, equation (3-3) turns into:

$$x_k(t+1) = x_k(t) + \alpha_k \quad (3-15)$$

This linear increase shows that each stage will need a fixed time interval to increase to the next stage. Specifically, there is a fixed time interval for a UDT flow to increase from 0 to 90% of the link capacity.

Suppose at the end of the first stage, UDT reaches rate R_0 ($R_0 \leq 0.9L$), then to reach $0.9L$ it takes

$$\left(\frac{R_0}{\alpha_0} + \frac{0.9L - R_0}{\alpha_1} \right) \cdot \frac{SYN}{1500 \times 8} = \frac{9(L - R_0)}{\alpha_0} \cdot \frac{SYN}{1500 \times 8}$$

Since $L - R_0 = 10^{e-1}$, $\alpha_0 = 10^{e-9}$, and $SYN = 0.01$, the above formula yields 750 *SYN*, which is 7.5 seconds.

³ We actually increase the packet-sending period by 1/8 in UDT, and it is a decreases factor of 1/9 on the packet-sending rate.

The above conclusion does not apply to the slow start phase, in which the sending rate increase is related to RTT (see Section 2.2.5). In the slow start phase, UDT needs $(\log_2 B * \max\{\text{RTT}, \text{SYN}\})$ time to probe the whole available bandwidth, where B is the available bandwidth measured in packets. This time is usually much less than 7.5 seconds, unless the bandwidth and RTT are exceptionally large. (For example, if $\text{RTT} = \text{SYN}$, then B needs to be greater than 2^{750} packets to let the slow start time exceed 7.5 seconds).

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

3.3 Fairness of UDT

3.3.1 Max-min Fairness

If all concurrent flows have the same L , then the increment function of each flow will satisfy the condition of the $\alpha(x)$ in the DAIMD algorithm. Therefore, in this situation UDT satisfies max-min fairness. In addition, this fairness is independent of RTT, since UDT uses a constant rate control interval.

We now discuss the situation when two flows F_1 and F_2 have different bottleneck link capacities. Suppose the bottleneck link capacities for F_1 and F_2 are L_1 and L_2 ($L_1 > L_2$), respectively. The equilibrium bandwidth allocation is (x_1, x_2) . The following condition should stand:

$$L_1 - x_1 \geq L_2 - x_2 \tag{3-16}$$

Otherwise F_2 has smaller decrements but has higher increments so that (x_1, x_2) cannot be the equilibrium. If the loss rate is small and

$$x_1 + x_2 \approx L_1 \tag{3-17}$$

then according to the equation (3-16) and (3-17) we can conclude that F_2 will take at least half of L_2 ($x_2 \geq L_2/2$).

Figure 3-4 illustrates the details of the competition between the two flows. Suppose at equilibrium, the two flows stay at stage k_1 and k_2 , respectively. According to (3-9), $\alpha_{k_1} \geq \alpha_{k_2}$, otherwise F_2 will occupy more bandwidth, which is impossible. If $k_2 \geq 2$, then F_2 has already occupied more than 90% of L_2 , so it is approximately fair. If $k_2 = 1$, then either F_1 stays at the same stage such that $\alpha_{k_1} = \alpha_{k_2}$, which means the two flows share the bandwidth equally, or it stays below $(L_1 - L_2)$, which means all the bandwidth of L_2 is left for F_2 .

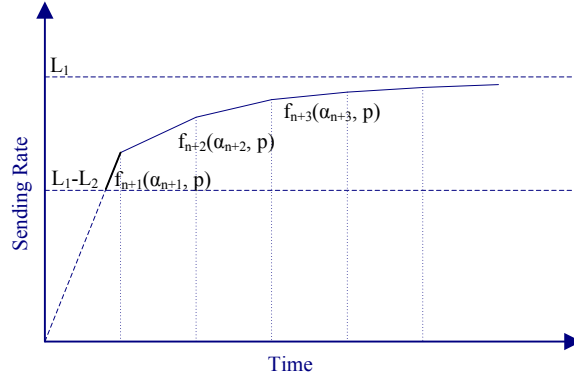


Figure 3-4: Two UDT flows with different link capacities.

The figure demonstrates the convergence to fairness equilibrium of two concurrent UDT flows, each having a bottleneck capacity of L_1 and L_2 , respectively.

The only situation that can cause unfairness is that F_2 stays at stage 2, and F_1 stays at the first stage above $(L_1 - L_2)$. In this case, F_2 is still more competitive than F_1 and will obtain more bandwidth of L_2 . Therefore, the lower bound of the throughput of F_2 is $L_2/2$.

3.3.2 TCP Friendliness

Because UDT uses a fixed rate control interval, when it competes with TCP, the network RTT will play an important role in the bandwidth sharing. Meanwhile, the increase parameter is decided by the parameter of link capacity L , which also affects the TCP friendliness.

If $SYN = RTT$, UDT increases no less than 1 packet per RTT ($\alpha_0 \geq 1$) only at $L > 100$ Mb/s; If $SYN < RTT$, UDT increases at a lower frequency than TCP.

Specifically, according to (3-12) (note that it has to be converted to use the units of packets/second) and the simple version of the TCP throughput model ($\sqrt{1.5/p}/RTT$) [70], the relationship between UDT and TCP (TF) can be written in the equation

(3-18):

$$TF = \left(\frac{3}{SYN} \cdot \frac{1}{\sqrt{p}} \cdot 10^{\frac{-k+e-9}{2}} \right) \bigg/ \left(\frac{1}{RTT} \cdot \sqrt{\frac{1.5}{p}} \right) = \frac{RTT}{SYN} \cdot 10^{\frac{-k+e-9}{2}} \cdot \sqrt{6} \quad (3-18)$$

UDT will obtain less bandwidth than coexisting TCP if $TF \leq 1$. The first stage of UDT is the most aggressive one, so $k = 0$ yields a sufficient condition for TCP friendliness, i.e., any UDT flow that satisfies the following condition must be friendly to TCP:

$$\frac{RTT}{SYN} \cdot 10^{\frac{e-9}{2}} \cdot \sqrt{6} \leq 1$$

Since $10^{e-1} < L \leq 10^e$, the above equation is satisfied if

$$RTT^2 \cdot L \leq SYN^2 \cdot 10^8 / 6 \quad (3-19)$$

Condition (3-19) is sufficient to guarantee that UDT is less aggressive than TCP and it shows that UDT is very friendly to TCP in low BDP environments. In addition, RTT has more impact on TCP friendliness than bandwidth.

3.4 Bandwidth Estimation

The parameters of link capacity L can be manually configured by applications if the network topology is known or it can be set up to be the upper limit of the sending rate of a certain UDT flow. In this section, we discuss how to estimate L automatically.

UDT uses receiver-based packet pairs [3] to estimate the link capacity L . The UDT sender sends out a packet pair (by omitting the inter-packet waiting time) every 16 data packets. If the burst-sleep method (see Section 2.3.2.5: High Precision Timer) is used, more continuous packets (packet train) can be sent out. Whether the incoming packets consist of a packet pair or trains can be determined from their timestamp information.

The receiver records the inter-arrival time of each packet pair or train and uses a median filter (more complex mechanisms can be found in [60, 75]) on them to compute link capacity. Suppose the median inter-arrival time is T and the average packet size in the measure period is S , then the link capacity can be estimated by S/T .

There are two major concerns in using packet pairs to estimate link capacity. One is the impact of cross traffic. The existence of cross traffic can cause the capacity be under estimated. Dovrolis, et al. point out that using packet pairs leads to a value referred to as Asymptotic Dispersion Rate [24], which is a value between available bandwidth and link capacity.

The other concern is the NIC interrupt coalescence. High speed NIC often has the functionality of interrupt coalescence to avoid too frequent interrupts. This can cause multiple packet arrivals to be notified by one single interrupt and hence the link capacity may be overestimated. This error can be eliminated by using the average inter-arrival time of multiple packet pairs. Prasad, et al. have a detailed discussion about the impact of interrupt coalescence on bandwidth measurement in [79].

We have seen that UDT may overestimate the capacity when there is only one flow in the network, whereas it tends to underestimate the capacity when there are multiple flows.

For a single flow, capacity estimation error only affects the convergence time. For multiple flows, it can also affect the fairness. (Note that if all flows have the same estimation error, they can still reach fairness.)

Consider a simple situation where we suppose $L=10^e$, L' is the estimated value, and the estimation error is ε , i.e., $L' = (1+\varepsilon)*L$. We can safely assume that $-0.9 < \varepsilon < 9$, because such a large error is very unlikely and we can even use the sending rate history record to eliminate certain extreme error⁴.

When competing with a flow with accurate L estimation, the bandwidth sharing between the two flows will be at most:

$$\begin{array}{ll} \frac{11+2\sqrt{10}+9\varepsilon}{11+2\sqrt{10}-9\varepsilon} & \text{if } (0 < \varepsilon < 1) \\ \sqrt{10} & \text{if } (1 \leq \varepsilon < 9) \\ 1 & \text{if } (-0.5 < \varepsilon < 0) \\ -\varepsilon/(1+\varepsilon) & \text{if } (-0.9 < \varepsilon \leq 0.5) \end{array}$$

We omit the detailed deduction process for these results and only give the following intuitive analysis. Suppose flow 1 has the right estimation of L and flow 2 has the error estimation of L' . For the first case, before flow 2 reaches $L' - L$, it increases $\sqrt{10}$ (according to equation 3-10) faster than flow 1, after which they have the same increments to compete for the rest of the bandwidth. In the second case, flow 2 is always $\sqrt{10}$ faster than flow 1. In the third case, the two flows will reach equal shares because they have the same increments. Finally, if $L' < L/2$, the throughput of flow 2 will be limited by L' .

As a simple example, if two flows share one 100 Mb/s link, flow 1 measures the link capacity as 101 Mb/s and flow 2 measures 99 Mb/s, then the two flows will still share the bandwidth almost equally. After flow 1 reaches 1 Mb/s, it will enter the same stage as flow 2, and both of the two flows will have the same increments and decrements.

3.5 Dealing with Packet Loss

While in most loss-based congestion control work, packet loss is regarded as a simple congestion indication, few of them have investigated the loss pattern in real networks. Because one single loss may cause a multiplicative rate decrease, dealing with packet loss is very important.

There are three particular kinds of situations related to packet loss that need to be addressed: loss synchronization, non-congestion loss, and packet reordering. Loss synchronization is a condition in which all concurrent flows experience packet loss at almost the same time. Non-congestion loss is usually caused by link error and can give transport protocols false signals of network congestion. Finally, packet reordering can mislead the receiver as packet losses.

⁴For example, if a UDT flow does not reaches 100 Mb/s for some time, say, the last 100 RTTs, but the estimation result is 1 Gb/s, such a result is either wrong or there are other limitations such that the flow will not reach 1 Gb/s in the next several RTTs. At this case, UDT can conclude that this estimation is invalid.

In particular, there has been an effective approach for packet reordering [100], so in this subsection we only focus on the first two situations, for which the solutions in literatures do not apply to UDT.

3.5.1 Loss Synchronization

The phenomenon of "loss synchronization" or "global synchronization" means all concurrent flows increase and decrease their sending rate at the same time, thus the aggregate throughput has a very large oscillation and leads to low aggregate utilization of the bandwidth. This is due to the fact that almost all the flows will experience packet drops when congestion occurs and have to drop their sending rates (so it is also called loss synchronization); when there is no congestion, they all increase the sending rate.

We use a randomization method to alleviate this problem. To describe this method, we define three terms. A loss event is the event when packet losses are detected. A UDT sender can detect a loss event when it receives a NAK report. A congestion event is a particular loss event when the largest sequence number of the lost packets in this loss event is greater than the largest sequence number that has been sent when the last rate decrease occurred. We call the period between two continuous congestion events a congestion epoch. Suppose there are M loss events between two continuous congestion events, and N is a random number that satisfies the uniform distribution between 1 and M .

For each congestion epoch, the decrease factor of the UDT control algorithm is randomized starting from $1/9$ to $[1 - (8/9)^N]$. Once a NAK is received, if this NAK starts a new congestion epoch, i.e., this is a congestion event, the packet-sending period is increased by $1/8$ (which is equivalent to decreasing the sending rate by $1/9$), and the packet sending is stopped for the next SYN time. For every N loss events, the packet sending period will also be increased by $1/8$.

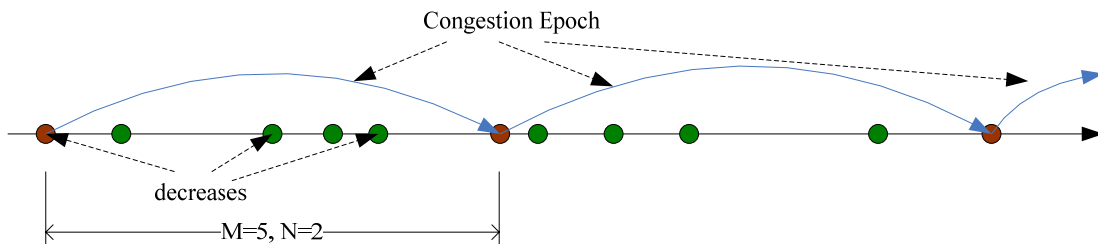


Figure 3-5: De-synchronization of UDT control algorithm.

The figure demonstrates the random loss decrease algorithm. In this figure, each node is a loss event on the time sequence, whereas a congestion epoch is noted as the period between the source and the sink of a directional arrow on the time sequence. In this example sequence, the first congestion epoch contains 5 loss events.

The process above can be described with the following algorithm. In this algorithm, LSD is the largest sequence number ever sent when the last NAK is received, STP is the packet sending period, $AvgNAK$ and $NumNAK$ are two variables used to

record the number of NAKs in the current congestion epoch and its smooth average value (the M value), and DR is the random number between 1 and $AvgNAK$ (the N value).

Algorithm: Random Loss Decrease

- 1) If the largest lost sequence number in the NAK is greater than LSD:
 - a) Increase the STP by 1/8: $STP = STP * (1 + 1/8)$.
 - b) Update AvgNAK: $AvgNAK = (AvgNAK * 7 + NumNAK) / 8$.
 - c) Update $DR = rand(AvgNAK)$.
 - d) Reset $NumNAK = 0$;
 - e) Record LSD.
- 2) Otherwise, increase NumNAK by 1, and if $NumNAK \% DR = 0$:
 - a) Increase the STP by 1/8: $STP = STP * (1 + 1/8)$.
 - b) Record LSD.

Figure 3-6: Random loss-based decrease algorithm.

Note that the use of explicit loss report (NAK) is different from the use of duplicate ACKs in TCP. With duplicate ACKs, the sender may not know all the loss events in one congestion event, and usually only the first loss event is detected. In fact, most TCP implementations will not drop the sending rate more than once in each RTT. However, in UDT, all loss events will be reported by NAK. Even with the SACK option enabled, the TCP sender will still regard the situation as one loss, and halve the congestion only once.

3.5.2 Noisy Link

Loss based control algorithms might not work well if there are significant non-congestion packet losses (e.g., due to link error, bad behavior of equipments, etc.), because they regard all packet losses as due to network congestion and will decrease the data sending rate accordingly. Although the link error rate on optical links is extremely small, sometimes there are non-congestion packets losses due to equipment problems and wrong configurations. We use a simple mechanism in UDT to tolerate such problems.

On noisy links, UDT does not react to the first packet loss in a congestion event. However, it will decrease the sending rate if there is more than one packet loss in one congestion event. This scheme is very effective in networks with small non-congestion packet losses, which we will demonstrate in Chapter 4. Not surprisingly, it also works for light packet reordering problems.

This algorithm is equivalent to removing Step 1.a in the random loss decrease algorithm described in Figure 3-6.

3.6 Related Work

Recently there have been several other new end-to-end congestion control algorithms proposed for grid networks. They can be roughly classified into three types.

The first type is to modify TCP by using large increase parameters (especially at large windows). Scalable TCP, High Speed TCP, and BiC TCP belong to this category. They all use binary indication of congestion and either increase or decrease the sending rate (congestion window size). Protocols in this category differ from each other by using different increase/decrease functions.

Scalable TCP [53] uses an MIMD approach to increase the increase parameter in proportion to the current window size: $\alpha(x) = 0.1x$. Its decrease factor is a constant of 1/8. Scalable TCP does not satisfy intra-protocol fairness due to its MIMD nature. HighSpeed TCP [29] redefines the response function of TCP, according to which it computes a series of increase and decrease parameters. Its increase parameter is an increasing function of the current window size, whereas the decrease factor is a decreasing function of the window size. BiC TCP [98] introduces a binary increase stage and it approximately approaches to AIMD(32, 1/8) at large window size.

Table 3-1 lists the increase/decrease function and response function of TCP Reno, Scalable TCP, High Speed TCP, BiC TCP, and UDT. Specifically, the BiC TCP parameter we use in this table is (32, 1/8, 0.01), and *target_win* is the window size at the midway between the current window size and the maximum window size, which is approximately the window size when the last loss occurs or is infinitely large if the current window size exceeds the old maximum window size. The response function of BiC TCP is according to the equation (1.4) in [98]. In Table 3-1, p is the loss rate, w is the congestion window size, x is the sending rate, and $w = x \cdot RTT$. The symbols in the UDT formula have the same meanings as before.

Table 3-1: Increase/Decrease and Response Functions of UDT and Various TCP algorithms.

This table lists the increase parameter, the decrease factor, and the corresponding response function of several AIMD control algorithms.

	Increase α	Decrease β	Response Function
TCP Reno	1	0.5	$1.22 \cdot p^{-0.5}$
Scalable TCP	$0.1w$	0.125	$0.8 \cdot p^{-1}$
HighSpeed TCP	$0.1578 \cdot w^{0.8024} \cdot \beta(w)/(2 - \beta(w))$	$-0.0520 \cdot \ln w + 0.6892$	$0.12 \cdot p^{-0.835}$
BiC TCP	$\min(\text{target_win} - w, 32)$	0.125	$24.65 \cdot p^{-0.5}$
UDT	$10^{\lceil \log(L - C(x)) \rceil - 9}$	0.111	$3 \cdot 10^{\frac{-k + \lceil \log L \rceil - 9}{2}} \cdot p^{-0.5}$

The second type can be seen in FAST TCP [48], which uses queuing delay as a multi-bit congestion flag to tune the congestion window size with an equation-based method. FAST TCP extends TCP Vegas. The analysis of FAST and Vegas can

be found in [48, 65], and more general analysis on delay-based approaches can be found in [45, 67]. According to [48], FAST TCP tunes the congestion window size every two RTTs, according to the ratio of $BaseRTT/RTT$, where $BaseRTT$ is the minimum RTT observed so far. However, on each packet loss event, FAST still decreases its window size by $1/8$. The FAST algorithm converges to weighted proportional fairness [48].

In fact, an early version of UDT made use of delay increasing trend information [46] as an early warning of network congestion. TCP-LP [59] uses a similar strategy.

The third type of congestion control algorithm for high BDP networks is to use explicit router feedback. XCP [50] is such a window-based protocol. In XCP, each router computes an increment or a decrement, which can be updated as it passes a successive router. The increment and decrement information is carried back by acknowledgments. At each router, the XCP efficiency controller computes the aggregate feedback according to the available bandwidth and the persistent queue size. The XCP fairness controller then distributes the aggregate feedback to all flows. If the aggregate feedback is positive, all the flows will have the same increase; if it is negative, each flow decreases in proportion to its own sending rate. The objective of this AIMD fairness controller is to make XCP satisfy max-min fairness. In particular, XCP uses a control interval of the average RTTs of all flows.

The DAIMD algorithm can be classified into the first type. It uses a decreasing function of the increase parameter, and a constant decrease parameter. However, the difference is that DAIMD tunes the sending rate based on time interval, which is similar to the second and the third approaches. UDT uses a constant control interval.

A lot of previous work has focused on the analysis of distributed congestion control algorithms. For example, Low's duality model has been used in analyzing TCP and AQM [64]. Kelly introduced a series of analysis on rate control and proportional fairness [51, 52]. Ott used a fluid model to describe the binary-based congestion control [72]. Bansal and Balakrishnan analyzed a group of binomial algorithms [10]. Gorinsky and Vin pointed out the limitations of Chiu and Jain's AIMD model and provided an extended analysis [36]. Loguinov and Radha analyzed a series of binary-feedback congestion control algorithms in rate-based applications [63]. Srikant summarized the stability and fairness of Internet congestion control in [86].

3.7 Concluding Remarks

In this chapter, we described a general type of AIMD congestion control algorithm, named DAIMD, whose increment decreases as the sending rate increases. This is different from other AIMD-based algorithms recently proposed to improve the performance of TCP at high BDP environments, which generally use large increase parameters. DAIMD is stable and converges to max-min fairness equilibrium.

UDT is a special case of such algorithms. We showed that UDT can converge to 90% of the link capacity in approximately a fixed time interval, independent of the network BDP. This makes UDT very scalable and efficient. UDT is fair when multiple

flows have the same bottleneck capacities, and the unfairness is lower bounded when flows have different bottleneck capacities. Moreover, it is friendly to TCP in low BDP networks.

In addition, we described how to estimate the parameter of link capacity in UDT and discussed the impact of estimation error. Further we discuss the robustness of our method on bandwidth estimation error.

Finally, we proposed two methods to deal with the global loss synchronization problem and the non-congestion packet loss problem, respectively. These loss handling schemes have great impacts on the transport protocol performance in real world.

4. PERFORMANCE EVALUATION

In this chapter, we evaluate UDT's performance in both simulation environments and real networks. The performance characteristics to be examined include efficiency (throughput), intra-protocol fairness, TCP friendliness, and stability. In real networks, we will also evaluate the implementation efficiency (CPU usage).

These performance characteristics and their measurement are listed and explained below.

a) Efficiency (Throughput)

We define the efficiency of UDT as the aggregate throughput of all concurrent UDT flows. Efficiency is one of the major motivations of UDT, which is supposed to utilize the high bandwidth efficiently, that is, utilize as much bandwidth as possible. Particularly, in grid computing, there are usually only a small number of bulk data flows sharing the network. A single UDT flow should reach high efficiency as well.

Suppose there are m UDT flows in the network and the i -th flow has an average throughput of x_i , the efficiency index is defined as

$$E = \sum_{i=1}^m \bar{x}_i$$

b) Intra-protocol Fairness

The fairness characteristic measures how fairly the concurrent UDT flows share the bandwidth. The most frequently used fairness rule is the max-min fairness, which maximizes the throughput of the poorest flows. If there is only one bottleneck in the system, then all the concurrent flows should share the bandwidth equally according to the max-min rule. In this case, we can use Jain's fairness index [44] to quantitatively measure the fairness characteristics of a transport protocol.

$$F = \left(\sum_{i=1}^n \bar{x}_i \right)^2 / n \cdot \sum_{i=1}^n \bar{x}_i^2$$

where n is the number of concurrent flows and x_i is the average throughput of the i -th flow. F is always less than or equal to 1. A larger value of F means better fairness, and $F = 1$ is the best, which means all flows have equivalent throughput.

c) TCP Friendliness

TCP friendliness is rather a more obscure measurement than the others, because it is almost impossible for a protocol with different control algorithms to reach the same performance as TCP and it is not reasonable to limit the throughput of a new protocol in high BDP environments to the throughput of TCP while the latter is very inefficient.

We consider the TCP friendliness separately in different situations, which are related to two factors: the network BDP and the TCP flow lifetime. First, in low BDP environments, where TCP can utilize the bandwidth efficiently, we expect that UDT should at least share the bandwidth with TCP fairly (equally); in high BDP environments, where TCP cannot efficiently use the

bandwidth, we expect UDT to make use of the bandwidth that TCP fails to use but leave enough space for TCP to increase. Second, TCP's behavior can be very different for bulk flows and Short-lived flows (considering the impact of TCP slow start at the beginning of a connection). We consider the situation of short-lived TCP separately because a majority of TCP traffic over the Internet are short-lived flows (e.g., web traffic).

For bulk TCP flows, 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:

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

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.

For short-lived flows, we will compare the aggregate throughput of a large number of small TCP flows under different numbers of background bulk UDT flows.

d) Stability (Oscillations)

We use the term of “stability” in this chapter to describe the oscillation characteristic of a data flow. A smooth flow is regarded as desirable behavior for most situations, and it often (although not necessarily) leads to better throughput. Note that this is different from the meaning of “stable” in control theory, and the latter means the convergence to a unique equilibrium from any start point.

To measure oscillations, we have to consider the average throughput in each unit time interval (a sample). We use standard deviation of the sample values of the throughput of each flow to express its oscillation [48]:

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

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 .

e) CPU Usage

CPU usage is usually measured by the usage percentage. However, sometimes we need to consider the data throughput when comparing CPU utilizations because the CPU may be used to process different sizes of data per unit time in each run. In this case, we use MHz/Mbps to describe the CPU utilization. The measurement of MHz/Mbps equals *CPU percentage * CPU frequency (MHz) / throughput (Mbps)*. Note that both CPU percentage and MHz/Mbps are NOT generic measurements. That is,

these values are only comparable against those values obtained on the same system, or at least systems with the same configuration.

The simulation results and the experimental studies on real networks will be described and discussed in section 4.1 and 4.2, respectively. Concluding remarks are given in section 4.3.

4.1 Simulations

We use the NS-2 [104] simulator to simulate UDT's performance under different bottleneck bandwidth, RTT, topologies, and queuing mechanisms. Because one single experiment can usually measure one or more of the performance characteristics of efficiency, intra-protocol fairness, and stability (e.g., by running 10 concurrent flows in one experiment, all the three measurements can be calculated), we describe these three characteristics together in section 4.1.1. We then describe the TCP friendliness characteristic in section 4.1.2. In the following two sections we discuss the impact of queuing and link error on the performance of UDT, respectively.

4.1.1 Efficiency, Fairness, and Stability

At first we simulate the efficiency of a single UDT flow at different link capacities and RTTs. We start a single UDT flow between two directly connected nodes (Figure 4-1). No link error model is configured, which means that the link will not cause any non-congestion packet loss. The simulation uses DropTail queue and the queue size is set to the maximum between BDP and 10 packets. Most of the simulations in this section will use this configuration unless otherwise explicitly noted.



Figure 4-1: Simulation network configuration.

This is a simulation network configuration diagram. In this configuration, data will be sent from a source node to a sink node via a directly connected link. In each experiment, we may change the number of parallel flows, the link capacity, the RTT, and the queuing parameters.

The bandwidth utilization for a single UDT flow with different end-to-end link capacities and RTTs is shown in Figure 4-2. In all situations UDT has reached more than 94% of bandwidth utilization.

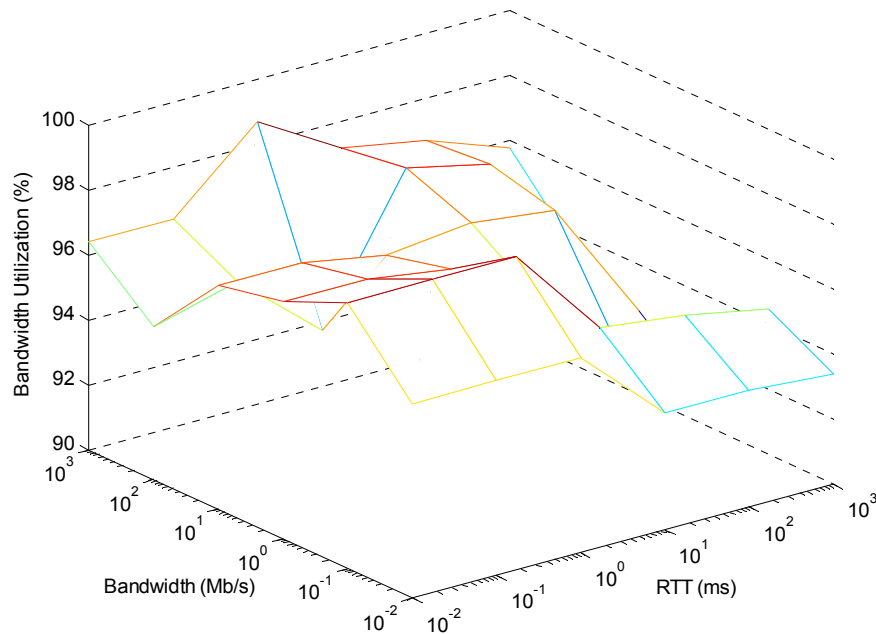
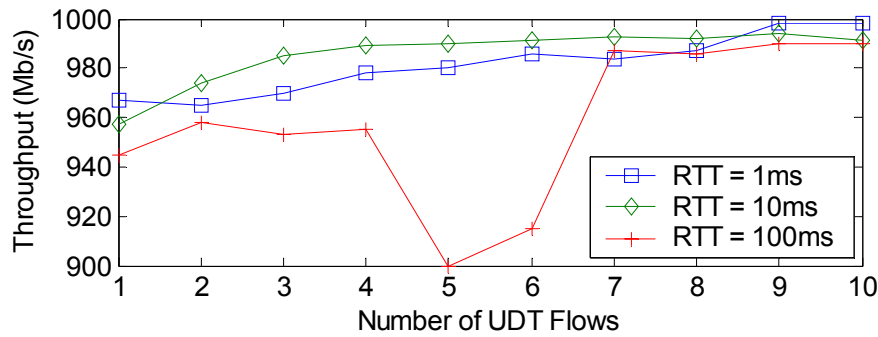


Figure 4-2: Bandwidth utilization of a single UDT flow with DropTail queue management. .

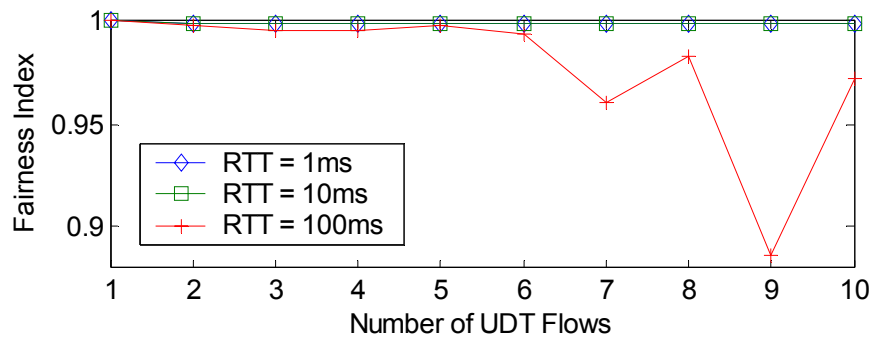
This figure shows the throughput (noted by bandwidth utilization) of a single UDT flow under different bandwidth and RTT. DropTail Queue is used and the queue size is set to $\max(BDP, 10)$.

Next we examine the performance of parallel UDT flows. Because there are multiple concurrent flows, we will also be able to calculate the fairness and stability indexes, in addition to the aggregate throughput. Using the same topology in Figure 4-1, we start a different number of UDT flows in each run.

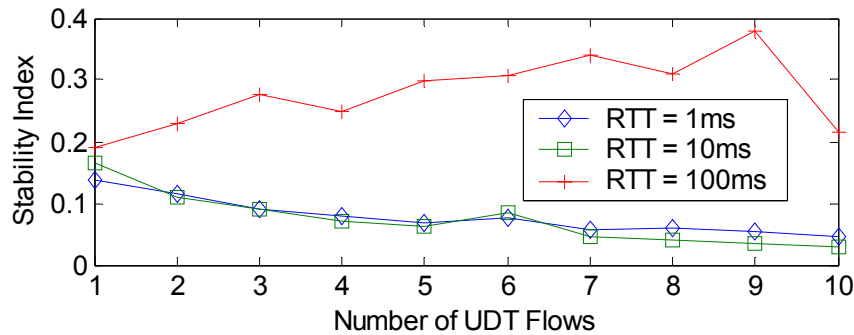
Particularly, when the link capacity is set to 1 Gb/s and the RTT is set to 1ms, 10ms, and 100ms, respectively, we record the aggregate efficiency index (throughput), fairness index, and stability index of different runs from 1 flow to 10 concurrent flows. Figure 4-3 (a) shows that parallel UDT flows can also utilize the bandwidth very efficiently with a little higher utilization than a single flow. Figure 4-3 (b) shows that UDT reaches good intra-protocol fairness, all runs reaching more than 0.9 of fairness indexes. Finally, Figure 4-3 (c) demonstrates the stability index of UDT.



(a)



(b)



(c)

Figure 4-3: Relationship between UDT Performance and number of parallelism.

The figure shows the efficiency index (a), the fairness index (b), and the stability index (c) of multiple concurrent UDT flows. The link capacity is 1 Gb/s and the RTT is 1ms, 10ms, and 100ms for each set of runs. The number of concurrent UDT flows varies from 1 to 10. DropTail Queue is used and the queue size is set to $\max(BDP, 10)$.

Furthermore, we start the UDT flows every 5 seconds to check UDT's convergence and fairness when concurrent flows have different start times. We use two network configurations: one has 100Mb/s link capacity with 1ms RTT, the other has 1Gb/s

link capacity with 100ms RTT. Figure 4-4 shows how 10 UDT flows converge to fairness quickly and stay steady at the equilibrium state. The fairness indexes after 50 seconds in both situation are all greater than 0.999.

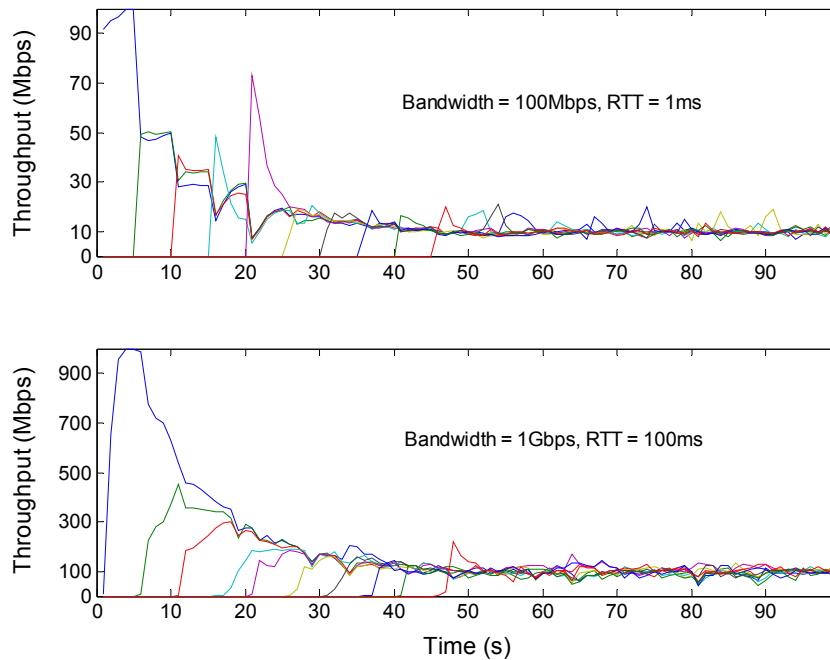


Figure 4-4: Fairness of UDT flows with different start time.

This figure demonstrates how UDT flows converge to the fairness equilibrium. The two network environments are 100Mbps with 1ms RTT (above) and 1Gbps with 100ms RTT (below), respectively. In each run, 10 UDT flows are started and each after 5 seconds. DropTail Queue is used and the queue size is set to $\max(BDP, 10)$.

We continue to examine the fairness property when different UDT flows have different RTTs - the RTT fairness. In this experiment, we start 7 concurrent UDT flows with RTT varying from 1 microsecond to 1 second at the same time. The network configuration is shown in Figure 4-5. Seven connections share the same 100 Mb/s bottleneck link.

The result is shown in Figure 4-6. The 7 flows reach similar average throughput and the fairness index is 0.978, which means that UDT's fairness is almost independent of RTT.

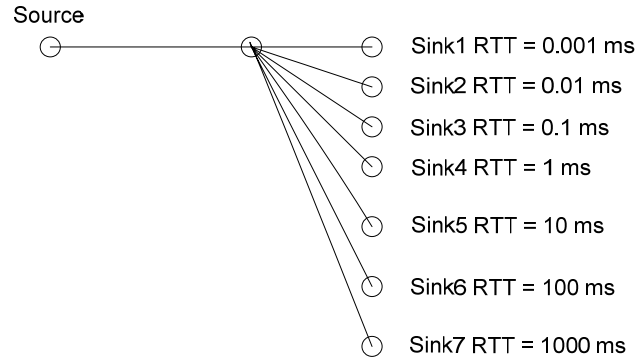


Figure 4-5: Network configuration for RTT fairness simulation.

This is a simulation network configuration to examine RTT fairness. Data will be sent through 7 pairs of sources and sinks and all of the paths share one single bottleneck link. The RTT of the 7 links are 0.001ms, 0.01ms, 0.1ms, 1ms, 10ms, 100ms, and 1000ms.

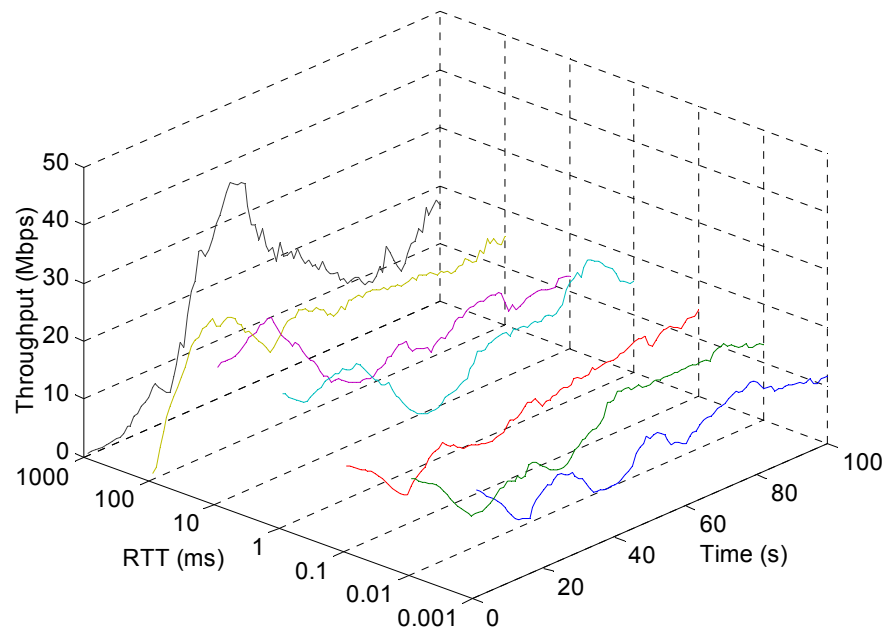


Figure 4-6: RTT independence.

The figure shows that the throughput changes over time when 7 concurrent UDT flows share the same 100Mbps bottleneck link with RTTs differing from 1 microsecond to 1 second. DropTail Queue is used and the queue size is set to $\max(BDP, 10)$.

While both the efficiency and fairness of UDT reaches almost optimal performance, oscillation is usually inevitable in loss-based control protocols that use binary feedbacks. To demonstrate UDT's stability characteristics, we compare the stability index of both UDT and TCP. To do so, we start 10 concurrent flows in each run with different RTTs. Figure 4-7 shows the stability

index 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. Note that the stability of TCP can be affected by queue size, while BDP is an optimal size for TCP's performance.

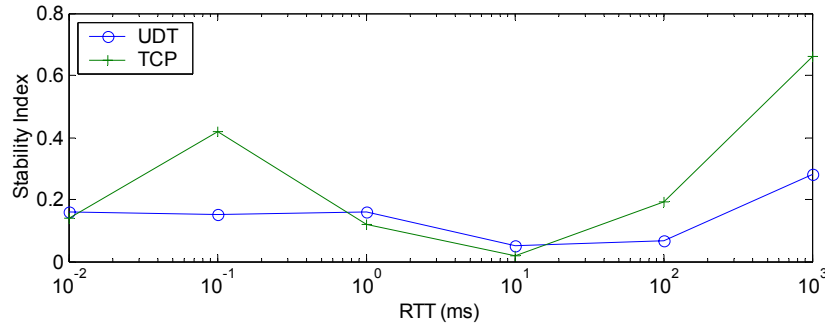


Figure 4-7: Stability index of UDT and TCP.

This figure compares the stability index of UDT and TCP. This simulation uses 10 concurrent flows running for 100 seconds on a 100Mb/s link with RTT varying from 0.01ms to 1s. The sample interval is 1 second. DropTail queue is used and the queue size is set to $\max\{10, BDP\}$.

Finally, we discuss UDT's performance in extreme situations of high congestion, rapidly changing available bandwidth, and complex network topologies.

Figure 4-8 shows how robust and convergent UDT is in a link with rapidly changing capacity. In this simulation, a constant bit rate (CBR) UDP flow is set up as background flow in a 100 Mbps link with 10ms RTT. The network topology is the same as that in Figure 4-1. A single UDT flow is used to observe the convergence. From Figure 4-8 we can see that the UDT flow rapidly responds to the change of the background UDP flow.

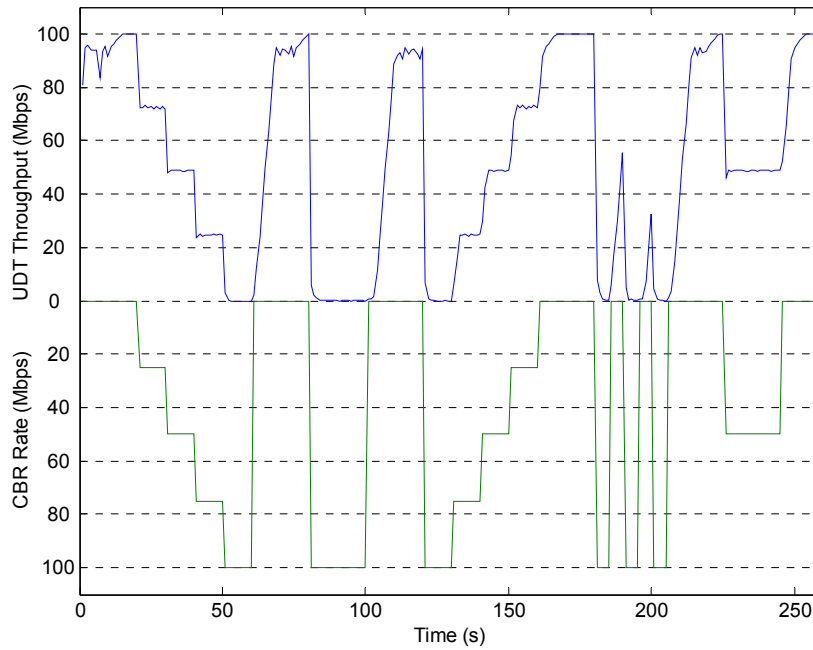


Figure 4-8: UDT robustness and convergence with rapidly changing background CBR flow.

This figure shows the throughput changes of a single UDT flow under drastic changes of a background UDP flow. The network has 100Mbps link capacity and 10ms RTT. The upper part shows the UDT flow's throughput; the lower part shows the UDP flow's throughput. DropTail queue is used and the queue size is set to $\max\{10, BDP\}$.

In multi-bottleneck scenario, UDT may fail to satisfy max-min fairness, since each flow may have different end-to-end link capacity. We now discuss the problem using a simple 2-bottleneck/2-flow scenario (as shown in Figure 4-9). In the 4-node topology of Figure 4-9, we change x between 0 and 200 Mbps to see the fairness between flows through AB and AC. The result in Table 4-1 shows that unfairness can arise when x is between 60 and 100 Mbps, but even at this range of x , flow AC still obtains more than half of its fair share. The worst case appears at $x = 80$ Mb/s, when AC obtains 53.22 Mb/s out of its 80 Mb/s fair share.

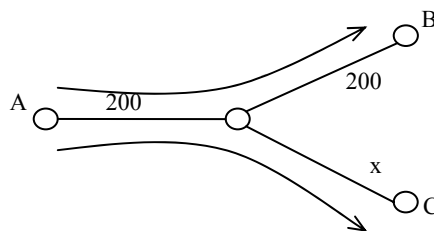


Figure 4-9: UDT performance in multi-bottleneck topology network.

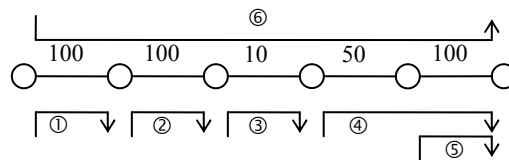
This is the network topology used for simulation of UDT performance in multi-bottleneck topologies. In this topology, the end-to-end capacity of AB is 200Mbps, whereas AC is x ($x < 200$). The MTU is set as 1500 bytes. DropTail queue is used in the network.

Table 4-1: UDT performance (in mbps) in the scenario of Figure 4-9.

In the table lists the throughput of two concurrent UDT flows described in Figure 4-9. The bottleneck capacity of AC (x) changes from 0.1Mb/s to 180Mb/s. The throughput of flow AB and AC is listed under each value of x .

X	0.1	1	10	20	40	60
AB	198.5	189.2	180.1	170.9	152.6	139.7
AC	0.098	0.979	9.955	19.88	39.24	52.56
X	80	100	120	140	160	180
AB	137.7	105.4	100.8	101.3	100.5	100.3
AC	53.22	91.62	98.47	98.20	98.85	99.01

We further examine the UDT performance in a more complex parking lot topology (Figure 4-10). UDT works efficiently and fairly in the system (Table 4-2). The aggregate throughput for each bottleneck link reaches more than 90% of the link capacity, whereas all the flows reach their fair share.

**Figure 4-10: UDT performance in complex topology network.**

The topology consists of 6 nodes, and the capacity is noted above each link. The RTT between any 2 adjacent nodes is 10ms. There are 6 flows in the network and they are noted as arrowed lines in the figure. DropTail queue is used in all nodes.

Table 4-2: UDT performance (in mbps) of Figure 4-10.

The table lists the throughput of each flow described in Figure 4-10. The first row lists the flow ID as in Figure 4-10 and the second row lists the corresponding throughput of each flow.

Flow ID	1	2	3	4	5	6
Throughput (Mb/s)	89.3	90.0	5.18	41.7	50.8	4.78

4.1.2 TCP Friendliness

In this sub-section we examine UDT's friendliness property against bulk TCP flows. We will discuss the situation of short-lived TCP flows in the experiment section.

Using the same network configuration in Figure 4-1, we start 20 concurrent flows (10 UDT flows and 10 TCP flows, or 20 TCP flows) in each run, which differs by RTTs (from 0.01 ms to 100 ms) and bottleneck link capacities (from 0.1 Mb/s to 1Gb/s). The TCP friendliness shown in Figure 4-11 indicates that TCP can obtain more bandwidth than UDT does in low BDP

environments, but as network BDP increases, especially as the RTT increases, UDT becomes able to obtain more bandwidth. However, even at 100 ms RTT and 1 Gb/s link capacity, UDT still only obtains just about 5 times more bandwidth than TCP.

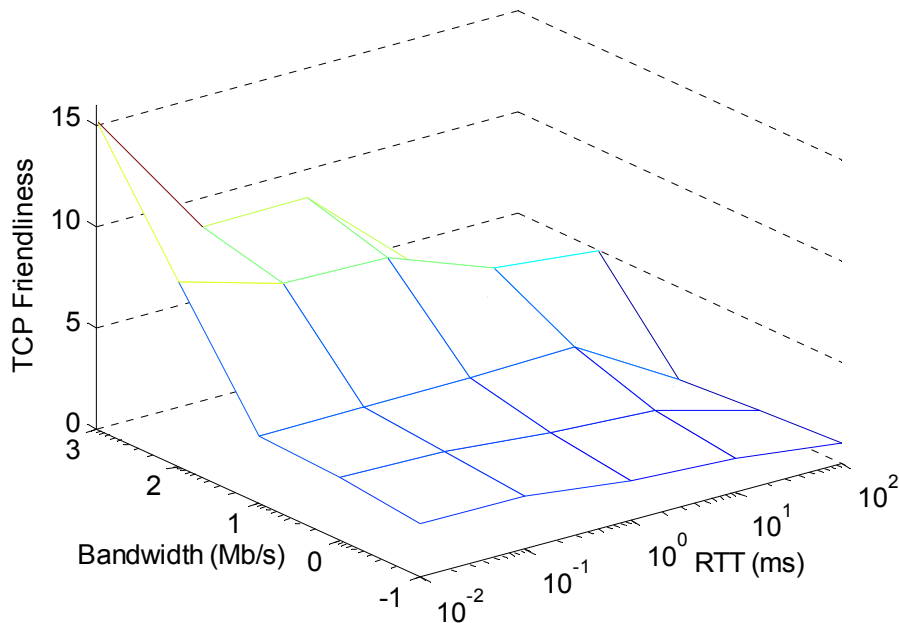


Figure 4-11: Bandwidth allocation between UDT and TCP.

This figure shows the TCP friendliness index of UDT under different bandwidth (varying from 0.1Mb/s to 1000Mb/s) and RTT (varying from 0.01ms to 100ms). DropTail queue management is used and the queue size is set to $\max(BDP, 10)$. No link error model is applied.

4.1.3 Impact of Queue Size and Management

Since UDT sends packets at every inter-packet interval, it does not need large queue size to support high throughput. The relationship between UDT performance and queue size is shown in Figure 4-12. In this simulation, we start a single UDT flow on the network of Figure 4-1. Two sets of simulations have been done. The first set uses 10 Mb/s link capacity whereas the second set uses 1 Gb/s link capacity. In each set we further investigate four situations with different RTTs: 0.1 ms, 1 ms, 10 ms, and 100 ms respectively. The path MTU and UDT packet size are both set to 1500 bytes.

Figure 4-12 shows that UDT can reach high bandwidth utilization with very small queue size. In 1 Gb/s, 100 ms RTT network, it only needs a queue of 10 packets length to reach 75% bandwidth utilization, and 1000 packets length to reach 99% bandwidth utilization. In contrast, the network BDP in this situation is 8333 packets.

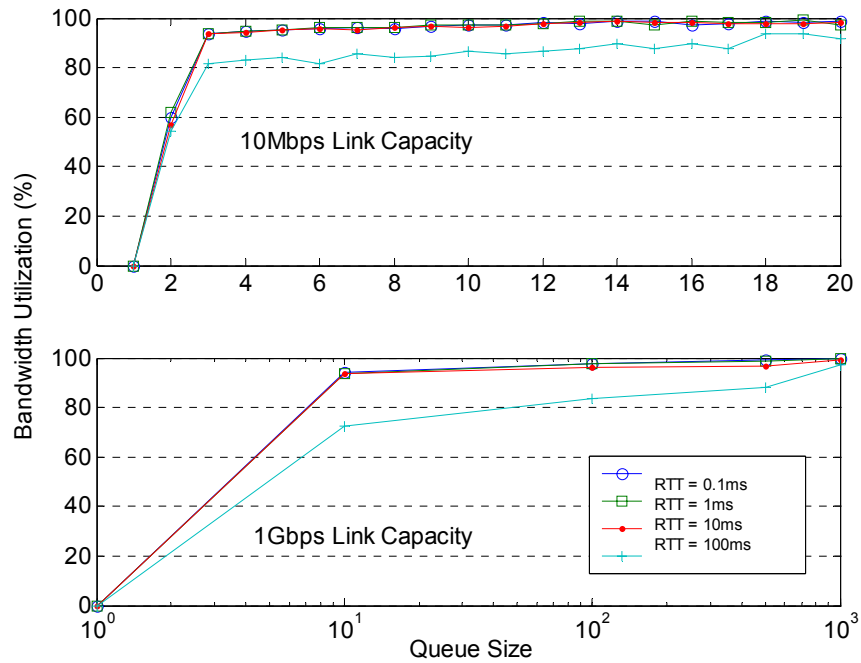


Figure 4-12: Relationship between UDT throughput and queue size (DropTail).

This figure shows the relationship between the throughput of a single UDT flow and the gateway queue size. DropTail queue is used. The two simulations use 10Mbps (above) and 1Gbps (below) link capacity, respectively. RTT varies among 0.1ms, 1ms, 10ms, and 100ms in each simulation. The queue size is measured by packets. The path MTU is 1500 bytes.

However, the queue size does affect TCP's performance and hence affects the TCP friendliness of UDT. If the queue size is too small to store TCP's burst data flow, it will decrease TCP's throughput, therefore, the coexisting UDT can reach higher performance.

Meanwhile, RED queue favors TCP since it does not have bias on burst flows as DropTail does [32]. The bias becomes larger as the queue size increases since the number of continuously dropped packets becomes larger. With DropTail queue, the bandwidth share ratio between TCP and UDT decreases as queue size increases, whereas the queue size has almost no effect in RED queue. This is convinced in the simulation of Figure 4-13, in which we show UDT's TCP friendliness index changing over queue size in both DropTail and RED queues.

TCP flows obtain the highest throughput when the queue size equals to BDP in both situations. After that point, TCP flow with DropTail queue decreases as queue size increases, whereas TCP flow with RED queue remains unchanged after a small drop.

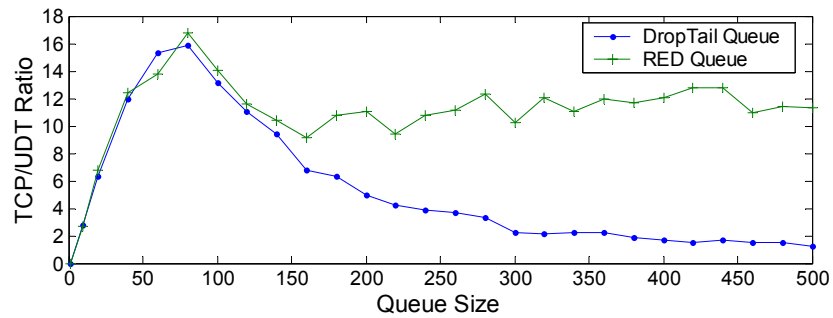


Figure 4-13: Relationship between TCP friendliness and queue size under different queue management schemes.

This figure shows the relationship between the TCP friendliness index of UDT and the gateway queue size. The simulation uses 100Mbps bandwidth link with 10ms RTT. The packet size is 1500 bytes and the BDP is about 83 packets. RED parameters are set as default by NS-2. The queue size is in packets.

Thanks to the rate control and its low queue size requirement, the queue management scheme does not affect UDT much. The throughputs are similar under both DropTail and RED queue managements. However, using RED can help to reduce the average queue size, at little cost of performance drop.

We repeat the simulation in Figure 4-2 but using RED queue this time. The UDT performance with RED queue management is shown in Figure 4-14.

In addition, since RED does not manage per flow dynamics, the fairness issues of RTT should be similar to that that of DropTail management.

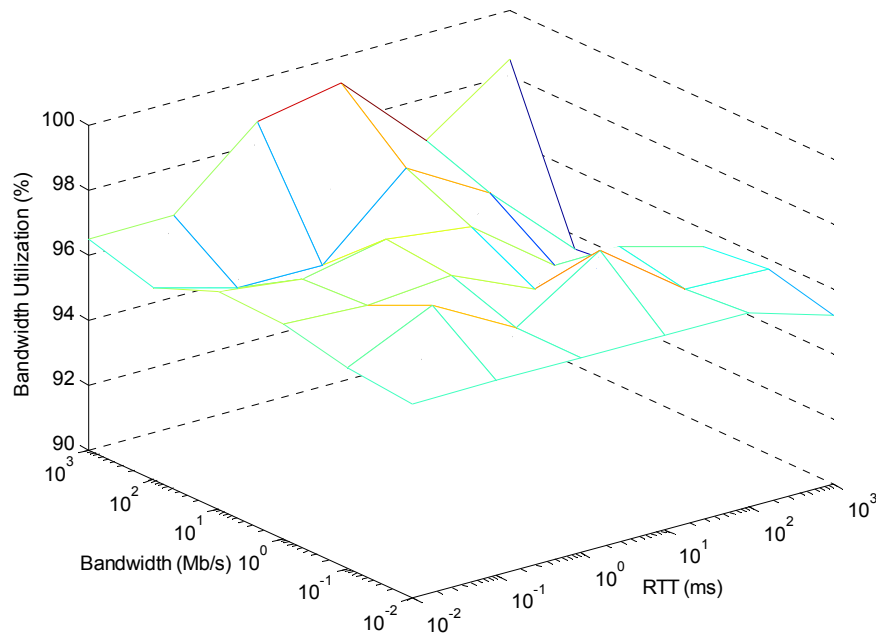


Figure 4-14: UDT throughput with RED queue.

This figure shows the throughput (noted by bandwidth utilization) of a single UDT flow under different bandwidth and RTT. RED queue is used, and the RED parameters are set by the default algorithm of NS-2. Simulation runs for 100 seconds.

4.1.4 Impact of Link Error

Because UDT uses loss-based congestion control algorithms, it can be misled by non-congestion packet losses just like loss-based TCP.

We enable a uniform distribution error model on the link of Figure 4-1. We fix the bandwidth at 100 Mb/s but change the RTT in three setting of 1 ms, 10 ms, and 100 ms. Figure 4-15 shows the bandwidth utilization changes of a single UDT flow against link error rates. The figure also shows the counterpart TCP performance, which performs very poorly in high RTT situations. A useful feature is that RTT has little effect on the link bit error rate, which allows UDT to work well on long haul networks.

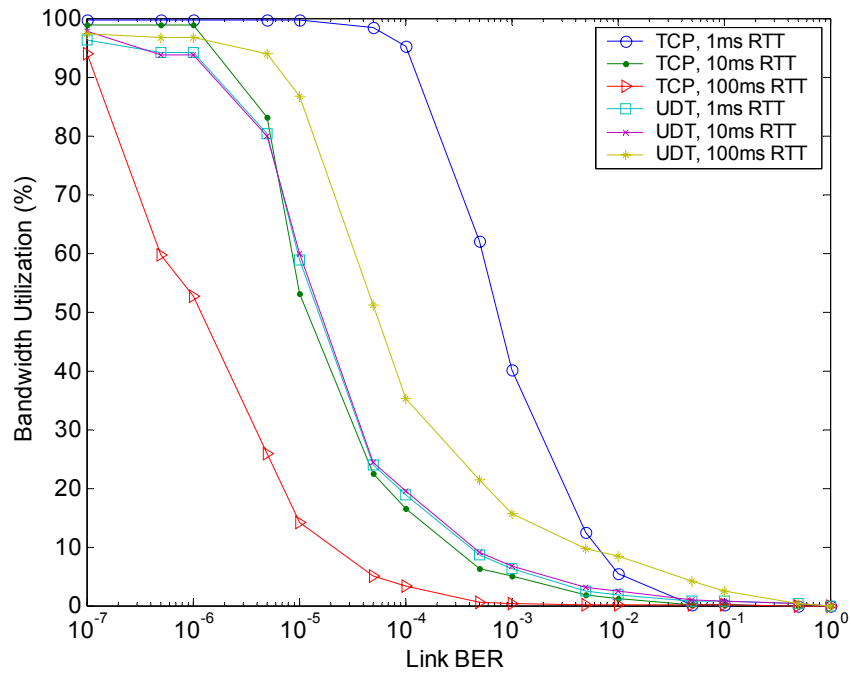


Figure 4-15: Impact of link error.

This figure shows a single UDT and TCP flow's performance under different link bit error rate (BER). Six simulations are done with different RTTs, whereas the bandwidth is fixed at 100 Mb/s. All nodes use DropTail queue with queue size set to $\max\{10, BDP\}$.

As we have mentioned in Chapter 3, Section 3.5.2, to accommodate more use scenarios, UDT provides an option in its congestion control algorithm so that it is more robust to non-congestion losses. Using the same simulation configuration in Figure 4-15, we perform a new set of UDT simulations with the noisy link algorithm. Figure 4-16 demonstrates the effectiveness of this mechanism. We can see that this algorithm significantly improves the bandwidth utilization of UDT at high link error rates. All UDT flows reach above 80% bandwidth utilization even at 1% non-congestion loss rate.

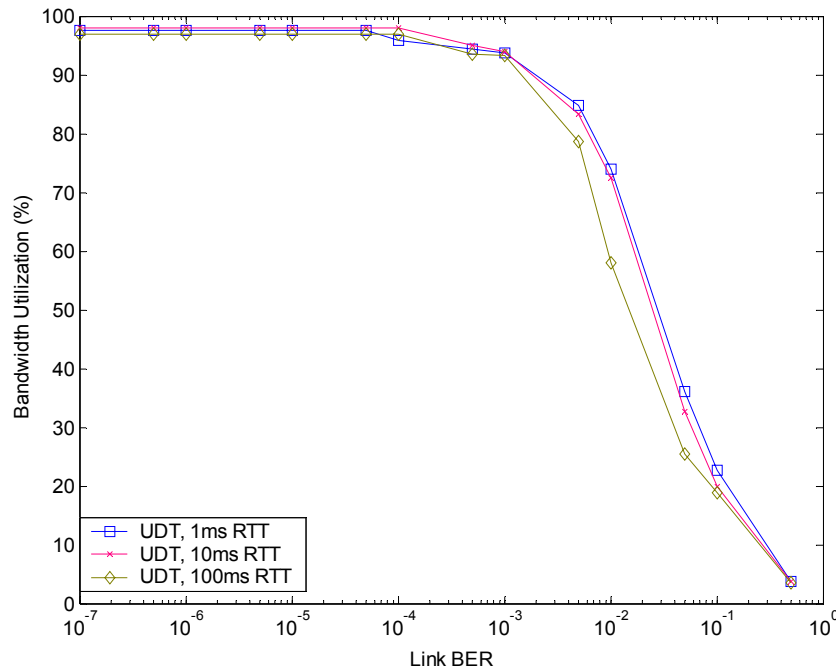


Figure 4-16: Effectiveness of loss resilience algorithm.

This figure shows the performance under different link bit error rate (BER) of single UDT flow when the noisy link algorithm in section 3.5.2 is enabled. Three simulations are done with different RTTs. All nodes use DropTail queue with queue size set to $\max\{10, BDP\}$.

4.2 Experimental Studies

While the simulations cover the majority of network situations, they do not simulate the real world perfectly and some factors such as CPU time usage are omitted. Experiments in real world settings give us more insight into UDT's performance. In this section we examine UDT's performance through experiments on real networks. Similar to the previous section, we discuss throughput or efficiency, intra-protocol fairness, and stability in one sub-section of 4.2.1. TCP friendliness will be covered in section 4.2.2. Section 4.2.3 examines the implementation efficiency (CPU usage). We will also demonstrate UDT's performance in real applications in section 4.2.4.

All of the experiments were performed on NCDM's Teraflow testbed [107]. In particular, we perform most experiments among three major sites: StarLight at Chicago, SARA at Amsterdam, and JGN2 at Tokyo. The distance (measured by RTT) between StarLight and SARA is 110 ms and the distance between StarLight and JGN 2 is 170 ms⁵.

⁵ The distance may have slight changes from time to time due to network reconfiguration. Link capacity also changes over time.

4.2.1 Efficiency, Fairness, and Stability

We performed two groups of experiments in different network settings to examine UDT's efficiency, intra-protocol fairness, and stability property.

In the first group of experiments, we start three UDT flows from a StarLight node to another StarLight local node, a node in Canarie (Ottawa, Canada), and a node in SARA (Amsterdam, the Netherlands), respectively (Figure 4-17).

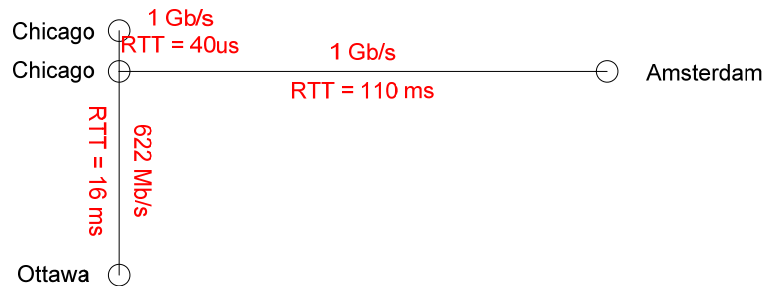


Figure 4-17: Experiment network configuration.

This figure shows the network configuration connecting our machines used for testing at Chicago, Ottawa, and Amsterdam. Between any two local Chicago machines the RTT is about 40us and the bottleneck capacity is 1Gb/s. Between any two machines at Chicago and Amsterdam respectively the RTT is 110ms and the bottleneck capacity is 1Gb/s. Between any two machines at Chicago and Ottawa respectively the RTT is 16ms and the bottleneck capacity is 622Mb/s. Amsterdam and Ottawa are connected via Chicago. The total bandwidth connecting the Chicago cluster is 1Gb/s.

Figure 4-18 shows the throughput of the single UDT flow over each link when the three flows are started separately. A single UDT flow can reach about 940Mbps over 1Gbps link with both 40us short RTT and 110ms long RTT. It can reach about 580Mbps over an OC-12 link with 15.9ms RTT between Canarie and StarLight. In contrast, TCP only reaches about 128 Mb/s from Chicago to Amsterdam after a thorough tuning for performance.

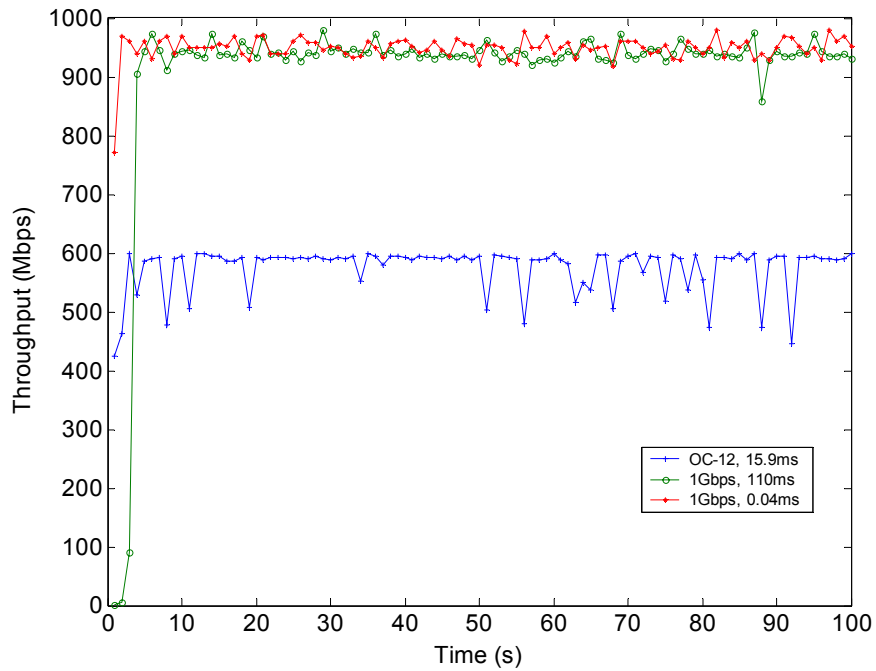


Figure 4-18: UDT performance over real high-speed network testbeds.

This figure shows the throughput of a single UDT flow over three different links described in Figure 4-17. The three flows are started separately and there is no other traffic during the experiment.

Figure 4-19 shows the throughput when the three flows were started at the same time. This experiment demonstrates the fairness property among UDT flows with different bottleneck bandwidths and RTTs. All the three flows reach about 325 Mb/s. Using the same configuration, TCP's throughputs are 754 Mb/s (to Chicago), 151 Mb/s (to Canarie), and 27 Mb/s (to Amsterdam), respectively.

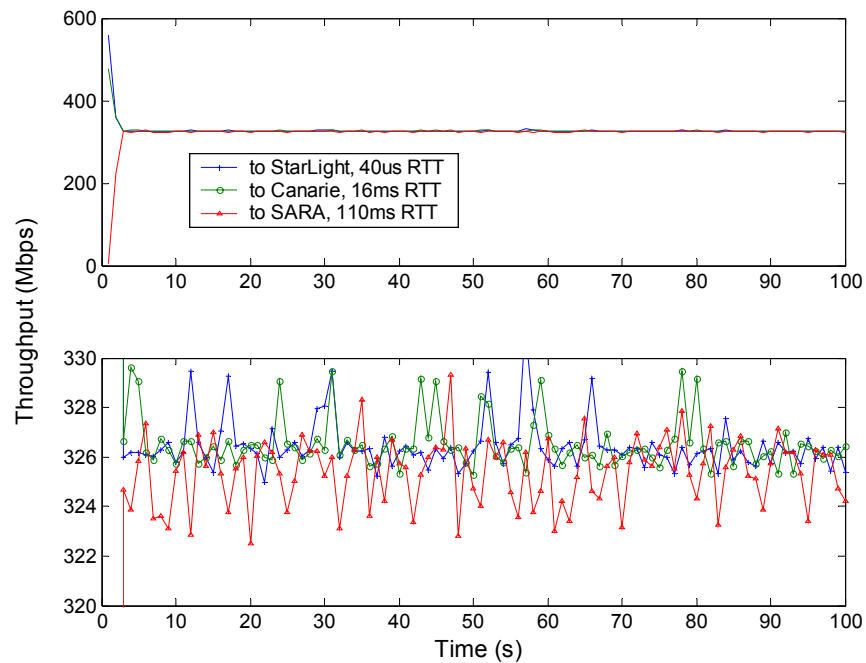


Figure 4-19: UDT fairness in real networks.

This figure shows the throughputs of 3 concurrent UDT flows over the different links described in Figure 4-18. No other traffic exists during the experiment. The sub-figure below is a local expansion of the sub-figure above.

We set up another experiment to check the efficiency, fairness, and stability performance of UDT at the same time. The network configuration is shown in Figure 4-20. Two sites, StarLight (Chicago) and SARA (Amsterdam), are connected with 1 Gb/s link. At each site, four nodes are connected to the gateway switch through 1GigE NIC. The RTT between the two sites is 104ms. All nodes run Linux 2.4.19 SMP on machines with dual Intel Xeon 2.4GHz CPUs.

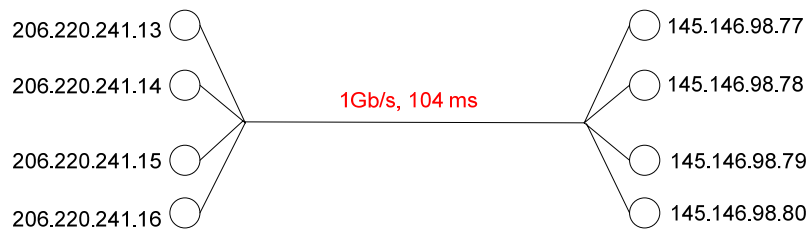


Figure 4-20: Fairness testing configuration.

This figure shows the network topology used in UDT experiments. Four pairs of nodes share 1 Gb/s, 104 ms RTT link connecting two clusters at Chicago and Amsterdam, respectively.

For the four pairs of nodes, we start a UDT flow every 100 seconds, and stop each of them in the reverse order every 100 seconds, as depicted in Figure 4-21.

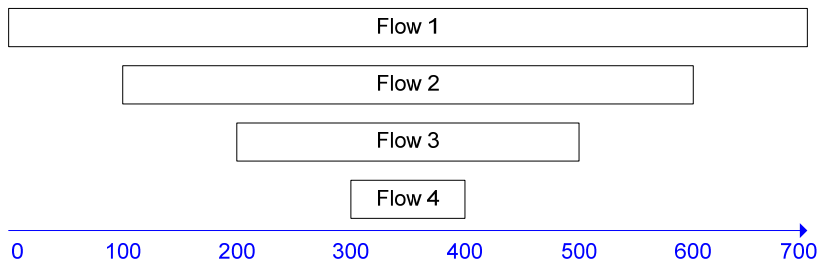


Figure 4-21: Flow start and stop configuration.

This figure shows the UDT flow start/termination sequence in an experiment configuration. There are 4 UDT flows and each flow is started every 100 seconds, and stopped in the reverse order every 100 seconds. The lifetime of each flow is 100, 300, 500, and 700 seconds, respectively.

The results are shown in Figure 4-22 and Table 4-3. Figure 4-22 shows the detailed performance of each flow and the aggregate throughput. Table 4-3 lists the average throughput of each flow, the average RTT and loss rate at each stage, the efficiency index (EI), the fairness index (FI), and the stability index (SI).

All stages achieve good bandwidth utilization. The maximum possible bandwidth is about 940 Mb/s on the link, measured by other benchmark software. The fairness among concurrent UDT flows is very close to 1. The stability index values are very small, which means the sending rate is very stable (few oscillations). Furthermore, UDT causes little increase in the RTT (107 ms vs. 104 ms) and a very small loss rate (no more than 0.1%).

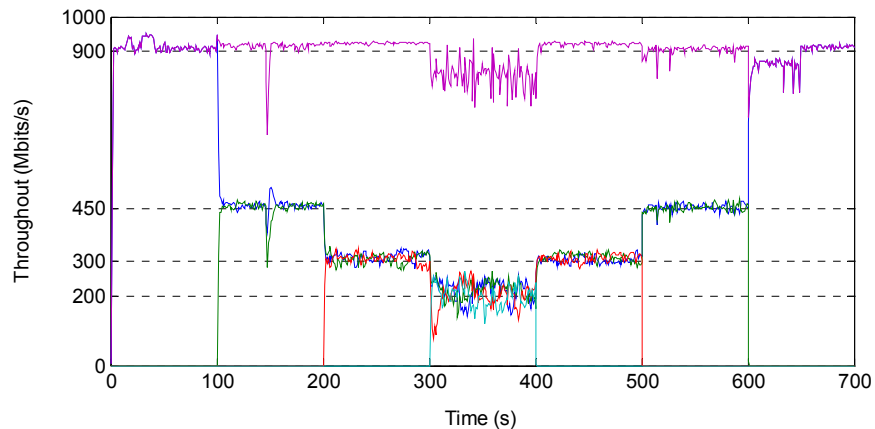


Figure 4-22: UDT efficiency and fairness.

This figure shows the throughput of the 4 UDT flows in Figure 4-21 over the network in Figure 4-20. The highest line is the aggregate throughput.

Table 4-3: Concurrent UDT flow experiment results.

This table lists the per-flow throughput, end-to-end experienced RTT, overall loss rate, the efficiency index, the fairness index, and the stability index of the experiment of Figure 4-22.

Time (sec)	1 - 100	101-200	201-300	301-400	401-500	501-600	601-700
Flow1	902	466	313	215	301	452	885
Flow2		446	308	216	310	452	
Flow3			302	202	307		
Flow4				197			
RTT	106	106	106	106	107	105	105
Loss	0	10^{-6}	10^{-4}	10^{-3}	10^{-3}	0	10^{-6}
EI	902	912	923	830	918	904	885
FI	1	.999	.999	.998	.999	1	1
SI	0.11	0.11	0.08	0.16	0.04	0.02	0.04

4.2.2 TCP Friendliness

Short-lived TCP flows such as web traffic and certain control messages comprise a substantial part of Internet data traffic. To examine the TCP friendliness property against such TCP flows, we set up 500 TCP connections where each transfers 1MB of data from Chicago to Amsterdam; a varying number of bulk UDT flows were started as background traffic when the TCP flows are started. TCP's throughput should decrease slowly as the number of UDT flows increases. The results are shown in Figure 4-23. They decrease from 69 Mb/s (without concurrent UDT flows) to 48 Mb/s (with 10 UDT concurrent flows).

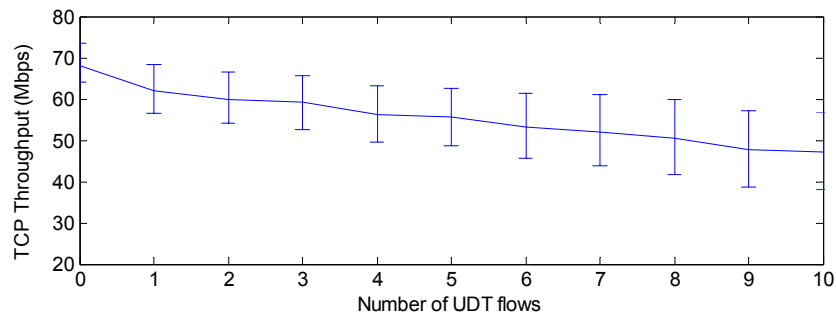


Figure 4-23: Aggregate throughput of 500 small TCP flows with different numbers of background UDT flows.

This figure shows the aggregate throughput of 500 small TCP transactions (each transferring 1MB data), under different numbers of background UDT flows varying from 0 to 10.

In the following experiment, we demonstrate UDT's impact to bulk data TCP flow in local networks where TCP works well. In Figure 4-24 it shows the result of 2 TCP flows and 2 UDT flows coexisting in the StarLight local network, with 1 Gb/s link capacity and 40 μ s RTT. TCP flows utilize slightly higher bandwidth than UDT flows. In real systems, the buffer size of a switch

may be set to a constant value to be used in long distance environments, so it can be much larger than the BDP of a local connection.

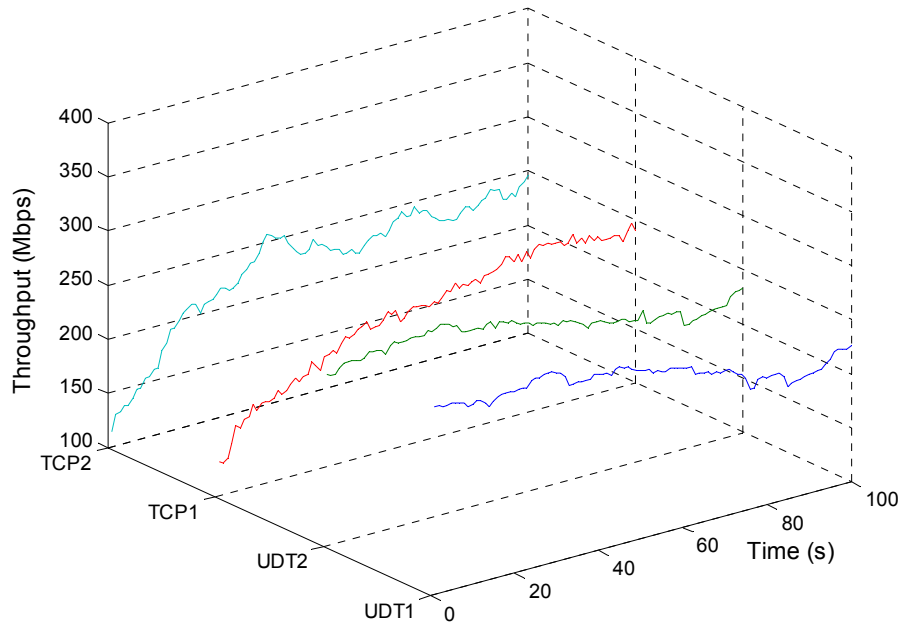


Figure 4-24: TCP friendliness in LAN.

The figure shows the throughput changes over time of 2 TCP flows and 2 UDT flows coexisting in StarLight local networks, with 1Gbps link capacity and 40us RTT.

4.2.3 Implementation Efficiency

In this sub-section we examine UDT's implementation efficiency through the CPU usage. Because UDT is supposed to be used to transfer large volumetric datasets at high speed, the implementation efficiency is very important; otherwise, CPU may be used up before reaching the optimal data transfer speed.

Figure 4-25 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.

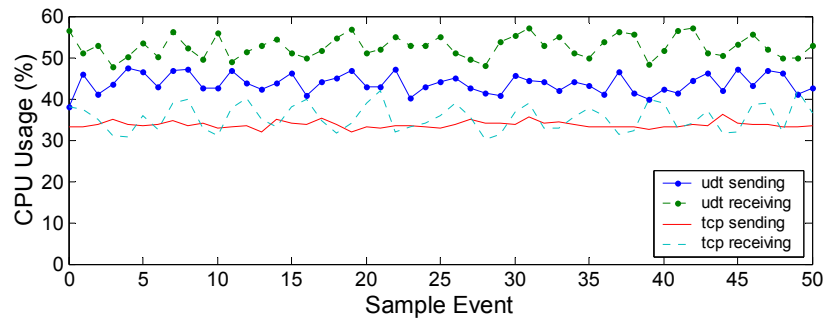


Figure 4-25: CPU utilization at sender and receiver sides.

This figure shows the CPU utilization percentage on the data source machine and the data sink machine, when a single TCP and a single UDT data transfer process is running. 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 970 Mb/s between memories.

Figure 4-26 depicts the CPU utilization of each UDT module in Figure 2-6. The data is obtained using Intel VTune Performance Analyzer [105]. Note that the modules of buffer management, loss processing, congestion/flow control, and UDP IO are called from the sending and receiving threads, so their CPU times are overlapped with the sending threads for sending and with the receiving thread for receiving, respectively. The sender and receiver threads cost most of the CPU time. The UDP IO time mentioned below is only the time consumed in the UDT layer (function calls of OS socket and necessary preparation of the calls) and excludes the data IO time inside the OS kernel.

Buffer management, loss information processing, congestion/flow control, and UDP IO take the majority of the CPU time. Inside the congestion control module, there are functionalities including timing and performance monitoring, which are the major CPU consumers (Figure 4-27) in this module.

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

Figure 4-27 depicts the CPU utilization of each UDT functionality processing. In Figure 4-27 we note that the CPU utilization on the overhead of the congestion control is very small (less than 1%). Therefore, the use of callback functions will not impose significant efficiency impact on UDT.

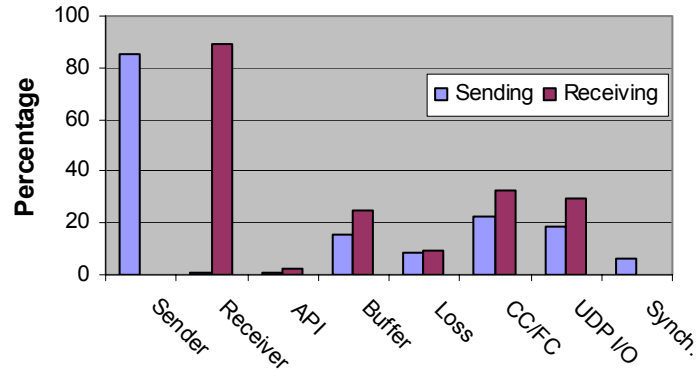


Figure 4-26: UDT CPU utilization by modules.

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

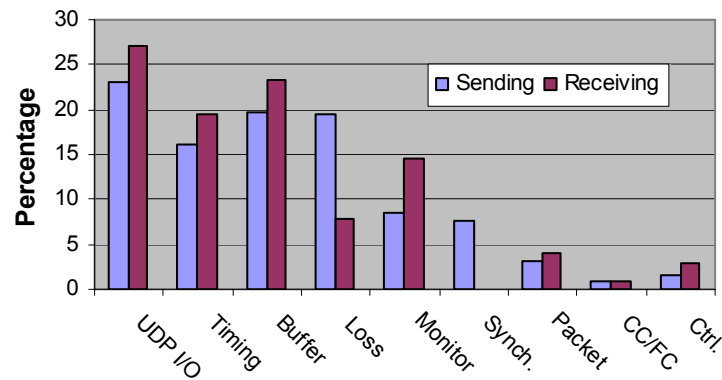


Figure 4-27: UDT CPU utilization by functionalities.

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

4.2.4 Performance in Real World Applications

While we have demonstrated UDT's performance with both extensive simulations and experiments, the real goal of UDT, however, is to benefit real world applications with high performance data transfer support. Therefore, it is very important to examine UDT's performance in real applications.

The major potential users of UDT are those who have large volumetric datasets to transfer. UDT is currently being used in an SDSS project to deliver astronomy data, in GridFTP to access large files on remote site, in the TeraJoin project to transfer multiple concurrent data streams, and so on.

Figure 4-28 shows the throughput when delivering SDSS data using two parallel UDT flows between two pair of machines in Chicago and Tokyo, respectively. The bottleneck is the disk IO, which is about 1.6 Gb/s for two pair of systems. UDT reaches 1.4 Gb/s.

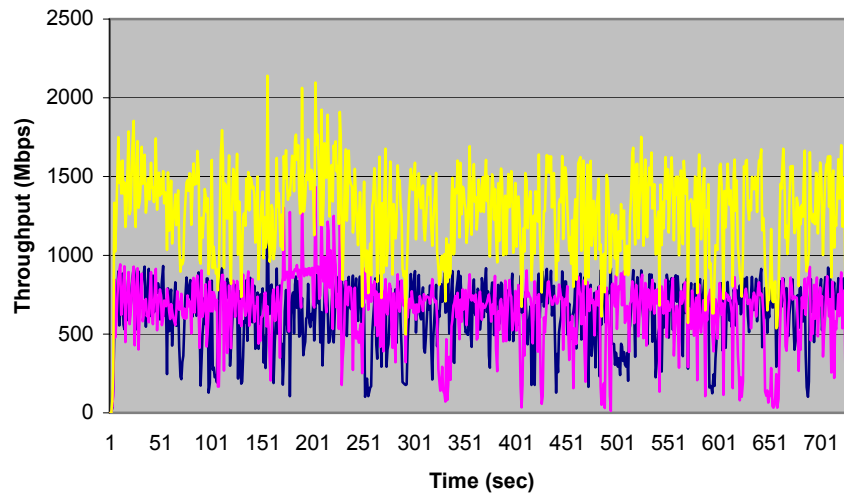


Figure 4-28: Transferring SDSS data using UDT.

This figure shows the throughput of two parallel UDT flows when transferring SDSS data from Chicago to Tokyo. The aggregate throughput is also shown in the figure in the top line.

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 4-4. 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 other applications, such as file transfer, remote data replication, distributed data mining, and distributed file servers.

Table 4-4: UDT disk-disk performance (all data in Mb/s)

This table lists the disk-to-disk file transfer throughput of a single UDT flow over different links between Chicago, Ottawa, and Amsterdam. The bottleneck of the file transfer is the disk IO speed, which is listed under the name of each site.

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

4.3 Concluding Remarks

We have done extensive simulations and experiments to examine UDT's performance in the past several years. These experimental studies demonstrate in a quantitative way that UDT is efficient, fair, stable, and friendly to TCP. They also show that our UDT implementation is efficient and correct. In addition, similar experiments have also been done in many other institutes [6, 9, 28, 57, 58]

We use simulations to check UDT's protocol design and control algorithms in a broad selection of network environments, including different bandwidth, RTT, network topologies, queuing mechanisms, number of concurrent flows, and so on. However, the real world experiment plays a more important role in the UDT project. The fundamental objective of UDT is to develop a transport protocol that can be used in real applications that need to transfer bulk data over high-speed wide area networks, and the best way to examine UDT's performance is to use it in real applications.

5. COMPOSABLE UDT

Most Internet protocols, including TCP, are designed to provide general service to support as many different applications as possible. This design philosophy has a major impact on the transport protocols. The majority of traffic on the Internet is contributed by TCP flows; but there still are applications that TCP cannot support well, in which a specialized protocol is preferred. For example, a GeoWall [56] application may prefer a smooth data transfer rate, whereas it is desirable to move SDSS data at the highest possible speed in private networks.

It is very desirable to create a configurable or reusable user space network stack on which a new congestion control algorithm can be easily implemented, deployed, and evaluated. First, a user space stack is much easier to get deployed, and so is the congestion control algorithms built in it. Second, this stack is useful to support application aware control approaches. An application may prefer to use different congestion control strategies in different situations. Third, this stack can save significant time for network researchers and developers because they can focus on the control algorithm itself rather than the whole protocol implementation. As a sequence, as there are more and more users, this stack can provide good software quality to support application development.

However, this stack is not a replacement for, but a complement to the kernel space network stacks. General protocols like UDP, TCP, DCCP [54], and SCTP [87] should still exist inside the kernel space of operating systems, but OS vendors may be reluctant to support too many protocols and algorithms, especially those application specific or network specific ones.

To address the above requirement, we extended our UDT library with Configurable Congestion Control to a new enhanced version called UDT/CCC. In UDT/CCC, a new control algorithm inherits this base class, redefines the proper control event handlers, and modifies certain control parameters when necessary.

UDT/CCC supports a wide variety of control algorithms, including but not limited to, TCP algorithms (e.g., NewReno [31], Vegas [14], FAST [48], Westwood [35], HighSpeed [29], BiC [98], and Scalable [53]), bulk data transfer algorithms (e.g., SABUL [37, 85], RBUDP [41], LambdaStream [97], CHEETAH [92], and Hurricane [96]), and group transport control algorithms (e.g., CM [5] and GTP [94]).

We envision the following use scenarios for UDT/CCC:

- Implementation and deployment of new control algorithms. Certain control algorithms may not be appropriate to be deployed in kernel space, e.g., a bulk data transfer mechanism used only in private links. These algorithms can be implemented using UDT/CCC.

- Application awareness support and dynamic configuration. An application may choose different congestion control strategies under different networks, different users, and even different time slots. UDT/CCC supports these application aware algorithms.
- Evaluation of new control algorithms. Even if a control algorithm is to be deployed in kernel space, it needs to be tested thoroughly before OS vendors distribute the new version. It is much easier to test the new algorithms using UDT/CCC than modifying an OS kernel.

In this chapter, we present the design, implementation, and evaluation of the UDT/CCC library. The rest of the chapter is organized as follows. We begin with the CCC design and architecture in Section 5.1, followed by the key implementation details in Section 5.2. We then evaluate the UDT/CCC library in the following two sections. In Section 5.3, we demonstrate the expressiveness and simplicity of using UDT/CCC to develop new control algorithms. In Section 5.4, we use experimental studies to examine the performance characteristics. Finally, we give brief concluding remarks in Section 5.5.

5.1 Design

In this section, we focus on how the congestion control interface is provided by the library and how it is implemented inside the UDT layer.

5.1.1 Overview

Figure 5-1 shows how the new CCC feature is inserted into UDT's layered architecture. An application can provide a congestion control class instance (CC in Figure 5-1) for UDT to process the control events, or use the default congestion control algorithm (described in Chapter 3) provided by UDT. The CC instance includes a set of necessary user-defined callback functions (control event handlers) to process certain control events.

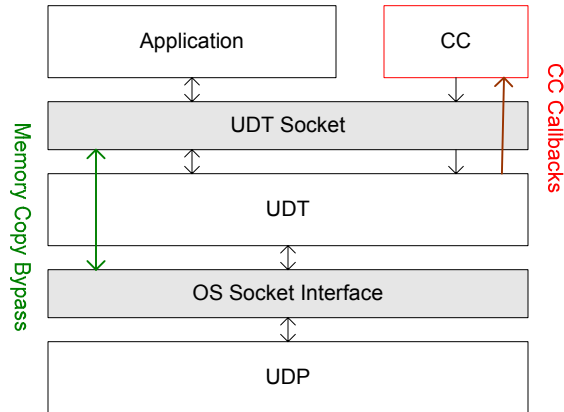


Figure 5-1: UDT/CCC architecture.

UDT/CCC adds a congestion control module (CC) to the application level, which is passed to the UDT layer by applications. The UDT layer, on the other hand, will call the event handlers in CC when related control events occur.

5.1.2 The CCC Interface

We identify four categories of configuration features to support configurable congestion control mechanisms. They are 1) control event handler callbacks, 2) protocol behavior configuration, 3) packet extension, and 4) performance monitoring.

5.1.2.1 Control Event Callbacks

Seven basic callback functions are defined in the base CCC class. They are called by UDT when a control event is triggered.

init and **close**: These two methods are called when a UDT connection is set up and when it is torn down. They can be used to initialize necessary data structures and release them later.

onACK: This handler is called when an ACK (acknowledgment) is received at the sender side. The sequence number of the acknowledged packet can be learned from the parameters of this method.

onLoss: This handler is called when the sender detects a packet loss event. The explicit loss information is given to users as the *onLoss* interface parameters. Note that this method may be redundant for most TCP algorithms that use only duplicate ACKs to detect packet loss.

onTimeout: A timeout event can trigger the action defined by this handler. The timeout value can be assigned by users, otherwise it uses the default value according to the TCP RTO calculation described in RFC 2988 [76].

onPktSent: This is called right before a data packet is sent. The packet information (sequence number, timestamp, size, etc.) is available through the parameters of this method.

onPktReceived: This is called right after a data packet is received. Similar to *onPktSent*, the entire packet information can be accessed by users through the function parameters.

onPktSent and *onPktReceived* are the two most powerful event handlers, because they allow users to check every single data packet. For example, *onPktReceived* can be redefined to compute the loss rate in TFRC. Due to the same reason, these two callbacks can also allow users to trace the microscopic behavior of a protocol.

processCustomMsg: This method is used for UDT to process user-defined control messages.

5.1.2.2 Protocol Configuration

To accommodate certain control algorithms, some of the protocol behavior has to be customized. For example, a control algorithm may be sensitive to the way that data packets are acknowledged. UDT/CCC provides necessary protocol configuration APIs for these purposes.

It allows users to define how to acknowledge received packets at the receiver side. The functions of *setACKTimer* and *setACKInterval* determine how often an acknowledgment is sent, in elapsed time and the number of arrived packets, respectively.

The method of *sendCustomMsg* sends out a user-defined control packet to the peer side of a UDT connection, where it is processed by callback functions *processCustomMsg*.

Finally, UDT/CCC also allows users to modify the values of RTT and RTO. A new congestion control class can choose to use either the RTT value provided by UDT, or its own calculated value. Similarly, the RTO value can also be redefined.

There are other features of the UDT protocol that are either not related to congestion control or are helpful to most control algorithms. These features, such as selective acknowledgment (SACK) [69] and robust reordering (RR) [100], cannot be configured by CCC users, although some of the features can be configured through UDT interfaces.

5.1.2.3 Packet Extension

It is necessary to allow user-defined control packets for a configurable protocol stack.

Because our UDT/CCC library is mainly focused on congestion control algorithms, we only give limited customization ability to the control packets. Data packet processing contributes to a large portion of CPU utilization and customized data packets may hurt the performance.

Users can define their own control packets using the Type 2 information in the UDT control packet header (Figure 2-3). The detailed control information carried by these packets varies depending on the packet types. At the receiver side, users need to override *processCustomMsg* to tell UDT/CCC how to process these new types of packets.

Note that UDT's packet-based sequencing with the packet size information provided by UDP is equivalent to TCP's byte-based sequencing and can also support data streaming.

5.1.2.4 Performance Monitoring

Protocol performance information supports the decisions and diagnosis of a control algorithm. For example, certain algorithms need some history information to tune the future packet-sending rate. Meanwhile, when testing new algorithms, performance statistics and internal protocol parameters are needed.

The performance monitor provides information including the duration time since the connection was started, RTT, sending rate, receiving rate, loss rate, packet sending period, congestion window size, flow window size, number of ACKs, and number of NAKs. UDT records these traces whenever the values are changed.

These performance traces can be read in three categories (when applicable): the aggregate values since the connection started, the local values since the last time the trace is queried, and the instant values when the query is made.

5.1.3 Inside The UDT Protocol

In this section, we will introduce how the control event handlers are processed inside the original UDT sending and receiving algorithms described in Section 2.2.6. The detailed implementation will be introduced in the next section.

5.1.3.1 The Sending Algorithm

Figure 5-2 describes the abstract UDT/CCC sending algorithm. In this algorithm, a sender's loss list is a data structure that records the lost data packets when informed of them by loss reports from the receiver or by sender side timeouts. ACK and NAK are the abbreviations of acknowledgment and loss report (negative acknowledgment), respectively.

-
- 1) If there is no application data to send, sleep until it is activated by the application.
 - 2) Packet sending:
 - a) If the sender's loss list is not empty and the number of unacknowledged packets does not exceed the congestion window size, remove the first lost sequence number from the list and pack the corresponding packet.
 - b) Otherwise, if the number of unacknowledged packets does not exceed the congestion and flow window sizes, pack a new packet.
 - c) Otherwise, wait here until an ACK or NAK is received, or timeout occurs. Go to Step 1.
 - 3) **onPktSent()**.
 - 4) Send the packed packet out.
 - 5) Wait until the next packet sending time. Go to Step 1.
-

Figure 5-2: UDT/CCC sending algorithm.

Step 2.a and 2.b are the window/flow control, which limits the number of unacknowledged packets. Retransmissions (2.a) are sent first if there are any lost packets. Regular packet sending (2.b) is limited to either the congestion window size or the flow window size, whichever one is smallest.

Step 2.c implements self-clocking. In a pure window-based control protocol, the packet sending is blocked here until an ACK comes. Note that the timeout in this step is used to break the deadlock when there is no feedback. It is different from the retransmission timeout in the *onTimeout* method.

Step 5 is the rate control, which suspends the data sending until the next sending time. In a pure window-based control protocol, the inter-packet time is set to 0, and thus this step is always skipped.

Both rate and window based approaches can be applied here. There are two specific situations. The first one is window-based control with pacing. In this situation an inter-packet time is calculated each time the congestion window size is updated, and Step 5 is used for the packet sending to sleep for this inter-packet time. The second situation is more complicated rate-based control with self-clocking. In the second situation, Step 5 is skipped, whereas Step 2.c is used for the packet sending to sleep for a certain time. Each time an ACK or a NAK is received, the sender compares the time passed since the last time a packet was sent and the expected packet-sending period, and none, one, or more packets (a burst) may be sent. A virtual congestion window may be used to control packet sending. This second method is like the one used in TFRC [30].

5.1.3.2 The Receiving Algorithm

Figure 5-3 describes the abstract UDT/CCC receiving algorithm. In this algorithm, the receiver's loss list is a data structure to store the sequence numbers of the lost packets. EXP is the abbreviation for timeout (expiration).

-
- 1) Query the timers
 - a) If ACK timer is expired and there are new packets to acknowledge, send back an ACK report; otherwise, if the user-defined ACK interval is reached, send back a lightweight ACK report.
 - b) If NAK timer is expired and the receiver's loss list is not empty, send back a NAK report;
 - c) If EXP timer is expired and there are sent but unacknowledged packets, execute **onTimeOut()**, and put the sequence numbers of these packets into the sender's loss list;
 - d) Reset the expired timers.
 - 2) Start time bounded UDP receiving. If nothing is received before the UDP timer expires, go to Step 1.
 - 3) If there is no unacknowledged packet, reset the EXP timer.
 - 4) If the received packet is a control packet, process it, and reset EXP timer if it is an ACK or NAK; According to the packet type, one of the following callback functions may be executed:
onACK(); **onLoss()**; **processCustomMsg()**;
Go to Step 1.
 - 5) Process the data packet.
 - 6) Check packet loss. If there are packet losses, insert the sequence numbers of the lost packets into the receiver's loss list and generate a loss report (NAK).
 - 7) **onPktReceived()**; Go to Step 1.
-

Figure 5-3: UDT/CCC receiving algorithm.

Step 1.c process the timeout event, similar to the TCP timeout event.

The control packet (including customized control packet) will be processed in Step 4. Note that UDT has its own ACK and NAK processing mechanism in addition to the user-defined event handler for data reliability purposes.

Finally, if a data packet is received, a user-defined event on packet receiving will be called in Step 7.

5.2 Implementation

Implementation efficiency is critical to UDT/CCC for two major reasons: 1) the primary goal of the library is to support the emerging high performance applications; 2) we do not want to limit the performance of the control algorithm by a poor implementation.

We have described the optimizations of the UDT library in Chapter 2. In this section, we only give a very brief summary of these optimizations, plus new problems arising from the introduction of CCC.

The first optimization category is focused on the avoidance of memory copy. Ideally, application data should be exchanged directly with the UDP channel. This optimization needs support from the API and protocol buffer management modules.

The second optimization category is designed to reduce the frequency of control events and the overhead of event handling. The majority of control events come from ACK/NAK triggering and processing. Other events, such as API call and timeout, are much less frequent. The UDT library has introduced an optimized loss processing algorithm to handle NAK processing.

The UDT protocol uses timer-based acknowledgment, so ACKs only comprise a small portion of packets. However, when introducing CCC, a new algorithm may require much more frequent ACKs. To handle this problem, UDT/CCC still only modifies the protocol buffers at certain time intervals; for all other ACKs (namely light ACKs), only the necessary sequence number related data are updated and no ACK2 will be sent for these light ACKs.

Finally, since high-end workstations usually have multiple processors, UDT use a multi-threading implementation. The sender, the receiver, and the listener are concurrent threads, but they are only started when necessary (lazy start). It is also necessary to provide fine granularity of the threading implementation.

5.3 Expressiveness

To evaluate the expressiveness of UDT/CCC, we implement a set of representative control algorithms using the library. Any algorithms belonging to a similar set can be implemented in a similar way. Meanwhile, we show that the implementation is simple and easy to learn.

In this section, we describe in detail how to implement control algorithms of rate based UDP, TCP variants, including both loss-based and delay-based algorithms, and group transport protocols as well.

UDT/CCC uses an object-oriented design. It provides a base C++ class (CCC) that contains all the functions and event handlers described in Section 5.1.2. A new control algorithm can inherit from this class and redefine certain control event handlers.

The implementation of any control algorithm is to update at least one of the two control parameters: the congestion window size (*m_dCWndSize*) and the packet-sending period (*m_dPacketPeriod*), both of which are CCC class member variables.

5.3.1 Rate-based UDP

A rate-based reliable UDP library (CUDPBlast) is often used to transfer bulk data over private links. To implement this control mechanism, CUDPBlast initializes the congestion window with a very large value so that the window size will not limit the packet sending. The rest is to provide a method to assign a data transfer rate to a specific CUDPBlast instance. A piece of pseudo code is shown below:

```
class CUDPBlast: public CCC
{
public:
    CUDPBlast() {m_dCWndSize = 83333.0;}

    void setRate(int mbps)
    {
        m_dPktSndPeriod = (SMSS * 8.0) / mbps;
    }
}
```

By using *setsockopt* an application can assign CUDPBlast to a UDT socket and by using *getsockopt* the application can obtain a pointer to the instance of CUDPBlast being used by the UDT socket. The application can then call the *setRate* method of this instance to set or modify a fixed sending rate at any time.

5.3.2 Standard TCP (TCP NewReno)

As a more complex example, we further show how to use the UDT/CCC library to implement the standard TCP congestion control algorithm (CTCP). Because a large portion of newly proposed congestion control algorithms are TCP-based, this CTCP class can be further inherited and redefined to implement more TCP variants, which we will describe in the next two subsections.

TCP is a pure window-based control protocol. Therefore, during initialization, the inter-packet time is set to zero. In addition, TCP needs data packets to be acknowledged frequently, usually every one or two packets⁶. This is also configured in the initialization.

TCP does not need explicit loss notification, but uses three duplicate ACKs to indicate packet loss. Therefore, for congestion control, CTCP only redefined two event handlers: *onACK* and *onTimeout*. In *onACK*, CTCP detects duplicate ACKs and takes proper actions. Here is the pseudo code of the fast retransmit and fast recovery algorithm in RFC 2581:

```
virtual void onACK(const int& ack)
{
    if (three duplicate ACK detected)
    {
        // ssthresh = max{flight_size / 2, 3}
        // cwnd = ssthresh + 3 * SMSS
    }
    else if (further duplicate ACK detected)
    {
        // cwnd = cwnd + SMSS
    }
    else if (end fast recovery)
    {
        // cwnd = ssthresh
    }
    else
    {
        // cwnd = cwnd + 1/cwnd
    }
}
```

The CTCP implementation can provide more TCP event handlers such as *DupACKAction* and *ACKAction*, which will further reduce the work of implementing new TCP variants.

Note that here we are only implementing TCP's congestion control algorithm, but NOT the whole TCP protocol. The UDT/CCC library does not implement exactly the same protocol mechanisms as in the TCP specification but it does provide similar functionality. For example, TCP uses byte-based sequencing whereas UDT uses packet-based sequencing, but this should not prevent CTCP from simulating TCP's congestion avoidance behavior. Certain TCP mechanisms that can affect congestion control, such as SACK and RR, have their equivalents implemented in the UDT library. We believe these mechanisms will benefit most congestion control algorithms.

⁶ Although TCP uses accumulative acknowledgments, a TCP implementation usually acknowledges at the boundary of a data segment. This is equivalent to acknowledging a UDT data packet in CTCP.

5.3.3 *New TCP Algorithms (Loss-based)*

New TCP variants that use loss-based approaches usually redefine the increase and decrease formulas of the congestion window size. Implementations of these protocols can simply inherit from CTCP and redefine proper TCP event handlers.

For example, to implement Scalable TCP, we can simply derive a new class from CTCP, and override the actions of increasing (by 0.1 instead of $1/cwnd$) and decreasing (by $1/8$ instead of $1/2$) the congestion window size.

Similarly, we have also implemented HighSpeed TCP (CHS), BiC TCP (CBiC), and TCP Westwood (CWestwood).

5.3.4 *New TCP Algorithms (Delay-based)*

Delay-based algorithms usually need accurate timing information for each packet. For efficiency, UDT does not calculate RTT for each data packet because it is unnecessary for most control algorithms. However, this can be done by overriding *onPktSent* and *onACK* event handlers, where the time of packet sending and the arrival of its acknowledgment can be recorded. For algorithms preferring one-way delay (OWD) information, each UDT packets contains the sending time in its packet header, and a new algorithm can override *onPktReceived* to calculate OWD.

Using the strategy described above, we implement the TCP Vegas (CVegas) control algorithm. CVegas uses its own data structure to record packet departure timestamps and ACK arrival timestamps, and then calculates accurate RTT values. With simple modifications to the control formulas, we further implement FAST TCP (CFAST).

5.3.5 *Group Transport Control*

While we have demonstrated that UDT/CCC can be used to implement end-to-end unicast congestion control algorithms, we now show that it can also be used to implement group-based control mechanisms, such as CM and GTP.

To support this feature, the new algorithm class simply needs to implement a central manager to control a group of connections. The control parameters are calculated by the central manager and then fed back to the control class instance of each individual connection.

We implemented GTP (CGTP) as an example of group-based control mechanisms. The GTP protocol controls a group of flows with the same destination. CGTP tunes the packet-sending rate at the receiver side periodically and feeds back the parameters using UDT/CCC's *sendCustomMsg* method.

5.3.6 *Summary*

We have implemented nine example algorithms using UDT/CCC, including rate-based reliable UDP, TCP and its variants, and group-based protocols. We demonstrated that our UDT/CCC library can support a large variety of congestion control algorithms, which are supported by only 8 event handlers, 4 protocol control functions, and 1 performance monitoring function.

The concise UDT/CCC API is easy to learn. In fact, it takes a small piece of code to implement most of the algorithms described above. Table 5-1 lists the lines of code (LOC) of implementations of TCP algorithms using UDT/CCC, as well as the LOC of those native implementations (Linux kernel patches). The LOC value is estimated by the number of semicolons in the corresponding C/C++ code segment.

To give more insight into the difference between LOCs in UDT/CCC based implementations and native implementations, we use the FAST TCP case as an example. The 31 lines of CFAST only implement the FAST congestion avoidance algorithm, whereas much of its code, especially the timing part, is inherited from CVegas. In contrast, of the 367 lines of FAST TCP patch, 142 of them are used to implement the FAST protocol (new files), 81 lines are used to modify the Linux TCP files, 86 lines are used to do monitoring and statistics, and 58 lines are used to do burst control and pacing.

As a reference point, the UDT library has 3134 lines of effective code (i.e., excluding comments, blank lines, etc.), SABUL has 2670 lines of code, and the RBUDP library has approximately 2330 lines of code. While these numbers are not enough to reflect the complexity of implementing a transport protocol, the much smaller number of LOC values of UDT/CCC based implementation can indicate the simplicity of using UDT/CCC.

Table 5-1. Lines of code (LOC) of implementations of TCP algorithms.

This table lists LOC of different TCP algorithms implemented using UDT/CCC and their respective Linux kernel patches (native implementations). The LOC of Linux patches include both added lines and removed lines.

Protocol	UDT/CCC	Native	
		Added	Removed
TCP	28	-	
Scalable TCP	11	192	29
HighSpeed TCP	8	27	1
BiC TCP	38	248	30
TCP Westwood	27	145	2
TCP Vegas	37 + 36 ⁷	132	6
FAST TCP	31	365	2

To give more insight into the difference between LOCs in UDT/CCC based implementations and native implementations, we use the FAST TCP case as an example. The 31 lines of CFAST only implement the FAST congestion avoidance algorithm, whereas much of its codes, especially the timing part, are inherited from CVegas. In contrast, of the 367 lines of FAST TCP patch, 142 of them are used to implement the FAST protocol (new files), 81 lines are used to modify the Linux TCP files, 86 lines are used to do monitoring and statistics, and 58 lines are used to do burst control and pacing.

⁷ CVegas reuses a timing class implemented by UDT, which contains 36 lines of code.

The class inheritance relationship of these UDT/CCC implemented algorithms can be found in Figure 5-4. Code reuse by class inheritance also contributes to the small LOC values of those TCP-based algorithms.

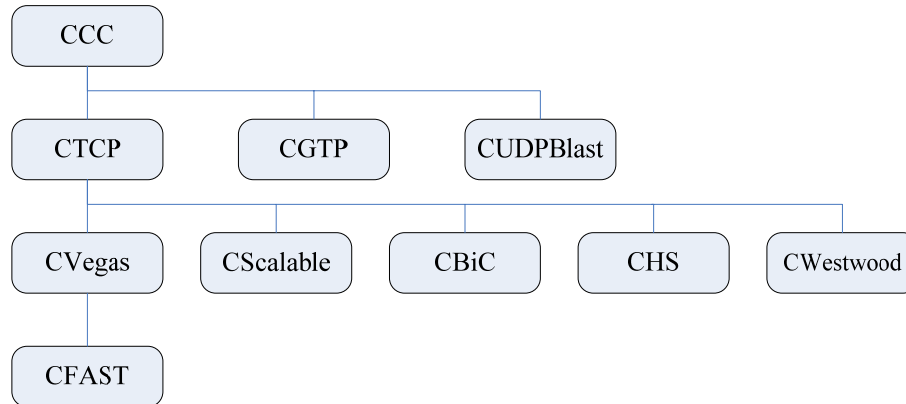


Figure 5-4: UDT/CCC based protocols.

This figure shows the class inheritance relationship among the control algorithms we implemented. Note that this is only for the purpose of code reuse, and it does NOT imply any other relationship among these algorithms.

5.4 Performance

It is a fundamental goal for UDT/CCC to simulate the performance of a control algorithm's native implementation or realize the algorithm's theoretical performance. In this sub-section, we will inspect two properties of UDT/CCC: 1) the similarity between UDT/CCC-based implementations and the native implementations, and 2) the CPU overhead brought in by UDT/CCC.

5.4.1 Similarity

In most cases, congestion/flow control algorithms are the most significant factor that determines a protocol's performance-related behavior (throughput, fairness, and stability). Less significant factors include other protocol control mechanisms, such as RTT calculation, timeout calculation, acknowledgment interval, etc.

We have made most of these control mechanisms configurable through the CCC interface and the UDT protocol control interface. In this subsection we will show that a UDT/CCC based implementation demonstrates similar performance to a native implementation.

Since TCP is probably the most representative control protocol, we compared an application level TCP implementation using our UDT/CCC library (CTCP) against the standard TCP implementation provided by Linux kernel 2.4.18.

The experiment was performed between two Linux boxes between Chicago and Amsterdam. The link is 1 Gb/s with 110ms RTT and was reserved for our experiment only in order to eliminate cross traffic noises. Each Linux box has dual Xeon 2.4GHz

processors and was installed with Linux kernel 2.4.18. We started multiple TCP and CTCP flows in separate runs, each of which was kept running for at least 60 minutes. The total TCP buffer size was set to at least the size of BDP (bandwidth delay product). Both TCP and CTCP experiments used the same testing program (except the connections were TCP and CTCP, respectively) with the same configuration (buffer size, etc.).

We recorded the aggregate throughput (value between 0 and 1000 Mbps), fairness index (value between 0 and 1), and stability index (equal to or greater than 0) in Table 5-2. The definitions of the fairness index and stability index can be found in Chapter 4. The fairness index represents how fairly the bandwidth is shared by concurrent flows and larger values are better. The stability index describes the oscillations of the flows and smaller values mean less oscillation. These three measurements summarize most of the performance characteristics of a congestion control algorithm.

Table 5-2: Performance characteristics of TCP and CTCP with various parallel flows.

The table lists the aggregate throughput (in Mb/s), fairness index, and stability index of concurrent TCP and CTCP flows. Each row records an independent run with a different number of parallel flows.

Flow #	Throughput		Fairness		Stability	
	TCP	CTCP	TCP	CTCP	TCP	CTCP
1	112	122	1	1	0.517	0.415
2	191	208	0.997	0.999	0.476	0.426
4	322	323	0.949	0.999	0.484	0.492
8	378	422	0.971	0.999	0.633	0.550
16	672	642	0.958	0.985	0.502	0.482
32	877	799	0.988	0.997	0.491	0.470
64	921	716	0.994	0.996	0.569	0.529

From Table 5-2, we find that TCP and CTCP have pretty similar throughput for small numbers of parallel flows. However, as the number of parallelism increases, CTCP stops increasing its throughput first and thus has a significantly smaller throughput than TCP when there are 64 parallel flows⁸. Further analysis indicates that the reason for this is that CTCP costs more CPU than kernel implemented TCP and with 64 flows the CPU time has been used up. To verify this assertion, we started another experiment using machines with dual AMD 64-bit Opteron processors and this time CTCP reaches more than 900Mbps at 64 parallel flows. The CPU usage problem will be further analyzed in Section 5.4.2.

In spite of the CPU utilization limitation, both of the implementations have similar performance on fairness and stability. They both realize good fairness with near-one fairness indexes, as the AIMD algorithm indicates. The stability indexes are around 0.5 for all runs.

⁸ TCP throughput will also start to decrease as the number of parallel flows increases [12].

In addition to the experiments above, we have also tested several reliable UDP-based protocols such as UDP Blast (CUDPBlast) to examine if the UDT/CCC based implementation conforms to the protocol's theoretical performance. We also examined the performance of UDT/CCC in a real streaming merge application, in which the receiver (where data is merged) requests an explicit sending rate to the data sources. This service is provided by a specific control mechanism implemented using UDT/CCC. The results of these experiments were positive and expected performance was reached.

5.4.2 CPU Usage Overhead

While we have addressed the expressiveness and similarity issues, there is one last major concern in using the UDT/CCC library: how much is the overhead brought in by UDT/CCC?

Before we go into the details, we show the CPU usages (percentage of CPU time used by TCP and CTCP) of the experiments in Table 5-2, Section 5.4.1. The result is listed in Table 3. Because there are two processors in each of our testing machines, the percentage varies between 0% and 200%.

Table 5-3: CPU usage of TCP and CTCP with various parallel flows.

The table lists the CPU usage percentage of both the sender and receiver sides. Each row records the data for the two runs of TCP and CTCP, respectively. The maximum CPU usage percentage is 200%.

Flow #	Sender		Receiver	
	TCP	CTCP	TCP	CTCP
1	9.5	16.7	10.1	13.0
2	18.6	33.9	19.8	24.4
4	31.2	58.2	20.9	26.5
8	35.0	71.3	38.6	44.0
16	62.5	122.8	69.6	73.1
32	88.1	198.0	90.5	105.7
64	93.2	198.0	91.9	94.2

Table 5-3 shows that at the receiver side, CTCP has very good CPU usage compared to the Linux TCP implementation. However, at the sender side CTCP has a much higher CPU usage; this is why CTCP stops increasing its throughput after 32 parallel flows in Table 5-2.

The major performance overhead added by an application level implementation comes from additional memory copy between application buffer and protocol buffer and additional context switches by packet processing and threading.

UDT has a best-effort method to reduce the additional memory copy and in the best case, additional memory copy will be completely eliminated.

However, UDT can do little for context switches caused by packet processing. The number of packets (for both data and control information) is decided by how much data applications need to transfer, whereas the number of control packets also depends on the specific protocol. For example, most reliable UDP protocols (SABUL, RBUDP, etc) only feed back an

acknowledgment at the end of a large data block, whereas TCP needs to acknowledge arrived packets more frequently, at about every 1 or 2 packets.

Our more detailed experiments discovered two major overheads added by CTCP compared to TCP. One is from acknowledging, including the subsequent overheads of context switches, buffer and sequence number updating, and thread synchronizations. The other is from application level threading.

In the following experiments, we increased the acknowledgment interval of our CTCP implementation, and, accordingly, the increases per ACK at the sender side. We run the experiments with the same setup as Table 5-2 in Section 5.4.1. The CPU usages are recorded in Table 5-4.

In Table 5-4 we use MHz/Mbps to describe CPU utilization. Because we cannot force TCP or CTCP to output a predictable throughput, we need to consider the data throughput when comparing CPU utilizations. The measurement of MHz/Mbps equals $CPU\ percentage * CPU\ frequency\ (MHz) / throughput\ (Mbps)$. Note that both CPU percentage and MHz/Mbps are NOT generic measurements. That is, these values are only comparable against those values obtained on the same system, or at least systems with the same configuration.

Table 5-4: CPU utilization of CTCP against number of parallel flows and ACK intervals.

This table lists the CPU utilization (MHz/Mbps) of CTCP with different numbers of parallel flows and different numbers of ACK intervals. Note that the first column (ACK interval = 2) is the result from Table 3 and it is listed here for comparison

Flow #	ACK Intervals						
	2	4	8	16	32	64	128
1	3.28	3.15	3.20	3.43	2.57	2.59	2.07
2	3.91	3.77	3.95	3.59	3.52	3.35	3.51
4	4.32	4.36	1.45	3.08	3.54	3.44	3.27
8	4.05	4.87	4.32	3.84	3.91	3.63	3.63
16	4.59	5.07	5.60	4.41	4.41	4.17	3.12
32	5.41	5.31	5.27	4.99	5.15	4.53	4.01
64	6.63	6.58	6.15	5.89	5.35	5.08	4.51

Table 5-4 shows that as the number acknowledgments decreases, the CPU usage drops sharply. For the same reason, less frequent acknowledging is recommended when designing an application level protocol for the purpose of efficiency.

Meanwhile, the MHz/Mbps measurement increases as the number of flows increases, but this is insignificant for TCP experiments. The situation, however, is caused by the user level threads used by CTCP. CTCP needs to start at least one thread for each connection, whereas TCP realizes the multiplexing in the kernel and does not have this overhead. This situation is worse for the sender side because each CTCP sender has to start two threads (UDT sender and receiver) and to deal with the thread synchronization caused by packet acknowledging.

In fact, through profiling analysis we found that UDP IO, threading synchronization, and control event handling contributes more than 90% of CPU utilization of CTCP. The fourth most significant CPU consumer is timing, which takes about 5% of overall CPU utilization on stamping each packet and triggering timer related events.

Although the CPU overhead of UDT/CCC may limit its usage in certain scenarios, we argue that the library is still useful in many other situations. First, in high performance computing, there are usually only a small number of flows sharing the high bandwidth. In this case, the threading overhead is low. Second, the overhead can be overcome by more powerful processors and more machines. Third, the overhead of UDT/CCC only exists when compared to kernel space implementations; it is insignificant when compared to other user space implementations. Finally, as we have shown in this section, control mechanisms with less frequent acknowledging would result in much less overhead.

We are, however, quite aware of the importance of CPU efficiency. Our current work is focused on code optimization.

We have also performed the same experiments on similarity and performance on other systems with different operating systems (Linux, BSD, OS X, Windows XP), hardware (Intel, AMD, and PowerPC processors), and networks. Although specific systems have more or less impact on the results, all the experiments conform to the similarity and performance trends we obtained from the experiments described in this section.

5.5 Related Work

There are few user level protocol stacks that provide a programming interface for user-defined congestion control algorithms as UDT/CCC does.

The Globus XIO [103] library has somewhat similar objectives, but the approach is quite different. XIO implements a set of primitive protocol components and APIs for fast creation or prototyping new protocols, which helps simplify the lower level simplification such as timing and message passing. In contrast, UDT/CCC allows users to focus only on the congestion control algorithm, and thus usually results in a much smaller program.

Less similar user level libraries include several user level TCP implementations [26, 66, 78, 91]. One particular implementation is the Alpine [26] library. Alpine is an attempt to move the entire kernel protocol stack into the user space, and provides (almost) transparent application interfaces at the same time. None of these libraries provide programmable interfaces.

In kernel space, the most similar work to UDT/CCC is probably the icTCP [40] library. It exposes key TCP parameters and provides controls to these parameters to allow new TCP algorithms deployed in user space. Despite the different nature of kernel and user space implementations, icTCP limits the update on TCP controls only, whereas UDT/CCC supports a broader set of protocols. Other work that uses a similar approach to icTCP includes Web100/Net100 [68] and CM [5].

Another work, STP [74], has more radical changes but also has more powerful expression ability. The STP's approach is to provide a set of protocol implementation APIs in a sandbox. Meanwhile, STP itself is a protocol that supports run time code

upgrading; thus, new protocols or algorithms can be deployed implicitly. To address the security problem arising from untrusted code, STP involves a complex security mechanism.

Yet another more complex library is the *x*-kernel [42]. *x*-kernel is an OS kernel designed to support data transport protocol implementations. The support mechanism of *x*-kernel is a modular based system and it is more decomposed than STP. Besides the support of protocol implementation, *x*-kernel has many optimizations inside the OS kernel for data communications.

Other modularized approaches include Horus [80], CTP [93] and its high performance successor [95].

While some of these in-kernel libraries may have performance and transparency advantages, their goals of fast deployment of new protocols/algorithms are compromised by the difficulty of getting themselves deployed. For example, *x*-kernel has been proposed for more than a decade and it still remains a research tool. In contrast, UDT/CCC library provides a very practical solution for the time being.

In addition, kernel space approaches need to protect their host systems and the network from security problems and they have to limit users' privileges to control the protocol behavior. For example, both STP and icTCP prevent new algorithms from utilizing more bandwidth than standard TCP. Such limitations are not feasible to the new control algorithms for high-speed networks such as Scalable, HighSpeed, BiC, and FAST. The security problem is much less serious for UDT/CCC because it is at user space and it is only installed as needed (in contrast, those libraries such as icTCP and STP will be accessible to every user if they are accepted by OS vendors).

Finally, there is another category of related work that attempts to provide a protocol language for easier, faster, or more readable protocol implementation. Such work includes Prolac [55] and FoxNet [12].

5.6 Concluding Remarks

The maturity of high-speed wide area networks encouraged the emergence of numerous new applications, and new control mechanisms supporting these applications as well. It has often been the case that implementing these new control algorithms in the kernel is not practical or proper. On the one hand, the wide deployment of new protocols or algorithms usually suffers a long time lag. On the other hand, OS vendors may only choose a very small number of protocols to implement in the kernel.

We have presented a user level transport protocol stack named UDT/CCC, which allows user defined congestion control algorithms to be easily implemented. Our UDT/CCC library enables easy implementations of a large variety of control algorithms while these implementations can still match the performance characteristics of those native implementations.

However, UDT/CCC is not meant to replace kernel protocol stacks, nor is it proposed as a means to implement any new protocols. Instead, it provides a practical alternative when a kernel space approach is difficult to implement, evaluate, and deploy. These use scenarios include the implementation of a new application or network specific congestion mechanism and the evaluation of new congestion control algorithms.

Finally, we are aware of the CPU utilization limitations of the current UDT/CCC implementation. On the one hand, there are unavoidable efficiency side effects for application level protocol implementations because of the additional memory copy and context switches. On the other hand, we will continue to optimize the implementation to minimize these negative impacts.

6. CONCLUSION

This dissertation addresses the solution to bridge high-speed wide area optical networks and distributed data intensive applications. Today's emerging high performance applications requires large volumetric data to be transferred at Gb/s speed among geographically distributed machines. However, although the maturity of optical network technology can provide this ability, the current Internet data transport protocols simply fail to effectively utilize the network resources in this situation.

We have presented UDT, a high performance data transport protocol to meet these requirements. We divided the problem into two orthogonal parts: 1) protocol design and implementation, and 2) Internet congestion control. We designed the UDT protocol particularly for fast bulk data transfer. The goal is to introduce minimum overheads to the network and the end host. We also designed the UDT congestion control algorithm such that the UDT data traffic can utilize the bandwidth efficiently and fairly. The UDT algorithm integrates a specific class of AIMD algorithm (DAIMD), bandwidth estimation technique, and several packet loss processing schemes. We have done extensive simulation and experimental studies to examine the performance.

This final chapter begins with a summary of the contributions of this dissertation in section 6.1. Section 6.2 lists some limitations as well as the guidelines for future research work. In section 6.3 we give final remarks on the whole work of UDT.

6.1 Contributions

Below we highlight the contributions of this dissertation:

6.1.1 *A High Performance Data Transport Protocol and Associated Implementation*

UDT provides a timely and practical solution to the problem of transferring bulk data in high-speed wide area networks. The UDT protocol is efficient and fair for bulk data transfer in high BDP network environments.

Our work systematically investigated the design and implementation issues of a high performance data transport protocol at the application level. The UDT project identified the overhead arising from acknowledgments, loss processing, threading, and memory copy, and proposed appropriate solutions. Although they were often neglected, protocol design and implementation have a significant impact on the efficiency.

In addition, UDT is easily deployable. This is important as there are only four versions of TCP that have been widely deployed in the past three decades because of the long time lag of standardization, implementation, and deployment of kernel space protocols. Although there were numerous TCP variants proposed at the same time as UDT was developed, these protocols are not expected to be deployed in the near future.

Finally, as a result of our research and development work, we developed an open source UDT library for both research and industrial use. The UDT library is in productivity phase and has been used in many real applications.

6.1.2 *An Efficient and Fair Congestion Control Algorithm*

We summarized a class of AIMD-based control algorithms named DAIMD, whose increase parameter is decreasing as the sending rate increases. We showed that DAIMD is fair and stable, and can be efficient as well given proper parameter tuning. UDT's control algorithm is a specific case of DAIMD.

UDT's congestion control algorithm addresses both efficiency and fairness. The algorithm takes approximately a constant time to converge to 90% of available bandwidth. UDT flows are fair to each other, even if they have different RTTs. While UDT is highly efficient, it is not necessarily aggressive. It is friendly to concurrent TCP flows. Bandwidth estimation techniques are used in the congestion control mechanism such that there is no need for manual tuning of the control parameters. In addition, the UDT algorithm also solves the loss synchronization problem using a random decreasing method. Finally, UDT can also handle limited non-congestion packet losses.

UDT is one of the first transport protocols that attempt to use bandwidth estimation techniques in determining control parameters. The rationale that the increase of the sending rate should be proportional to the available bandwidth is obvious. In fact, XCP uses such bandwidth utilization information obtained from the intermediate routers. Since end-to-end protocols cannot get explicit information from routers, estimation technique is a natural choice.

6.1.3 *A Configurable Transport Protocol Framework*

Composable UDT, or UDT/CCC, offers more to application development and network research by allowing configurable congestion control algorithms. This feature enables easy development of application or network specific control mechanisms, as well as easy evaluation of new control algorithms. Our UDT/CCC library enables easy implementations of a large variety of control algorithms while these implementations can still match the performance characteristics of those native implementations.

UDT/CCC is currently one of the few configurable protocol frameworks available at user space. Because it is usually difficult to implement and test new control protocols in kernel space, the user space framework is very desirable.

6.2 **Limitations and Future Research Direction**

6.2.1 *Bandwidth Estimation in the Context of Transport Protocol*

End-to-end transport protocols such as TCP usually suffer from the lack of explicit network usage information. XCP has demonstrated that such information can significantly promote the efficiency of congestion control protocols. However, a different approach is to estimate this information at the end hosts, such as the bandwidth estimation techniques used in UDT.

UDT's approach is effective in data intensive applications where there are a small number of concurrent flows, but it is not suitable in a high concurrency environment⁹, because the bandwidth estimation technique currently used in UDT may overestimate the available bandwidth in high concurrency environments.

Future work can further investigate bandwidth estimation techniques in the context of transport protocols. A bandwidth estimation scheme within transport protocols may have more advantages than regular bandwidth estimation tools. For example, it may be allowed to take longer time to obtain more accurate information, and it can also utilize certain history information by its host protocol.

6.2.2 *Implementation Optimization*

Profiling analysis to the UDT implementation shows that there is still space to improve the CPU usage. Efficient implementation is especially important to user space protocol stacks because they cost more CPU time on memory copies and context switches. We believe that further investigation of the efficient implementation of UDT will also benefit many other transport protocol implementations.

6.2.3 *Configurability*

The current Composable UDT is limited for congestion control mechanisms. A more advanced version could have enabled many more extensions to network protocols, similar to what NS-2 does, but working on real networks. A particularly interesting extension is the data reliability and timeliness control, which often has different requirements by different applications. For example, multimedia applications may require unreliable messaging service. Furthermore, more complex extensions can introduce gateway software for protocols involving gateway algorithms and for overlay networks.

6.3 **Final Remarks**

Scalability has been one of the major research problems of the Internet community ever since the emergence of the World Wide Web (WWW). The insufficient number of IP addresses may be the most commonly known scalability problem. However, in many high-speed networks researchers have also found that as a network's bandwidth-delay product increases TCP, the major Internet data transport protocol, does not scale well either.

As an effective, timely, and practical solution to this BDP scalability problem, we designed and implemented the UDT protocol that can utilize the abundant optical bandwidth efficiently and fairly in distributed data intensive applications.

⁹ Here we mean to enhance UDT with new features (so that it may be used in some traditional environments low bandwidth high concurrency environments also). The current UDT was not designed for such environments.

UDT's approach is highly scalable. Given that there is enough CPU power, UDT can support up to unlimited bandwidth within terrestrial areas. The timer-based selective acknowledgment generates a constant number of ACKs no matter how fast the data transfer rate is. The congestion control algorithm and the bandwidth estimation technique allow UDT to increase to 90% of the available bandwidth no matter how large it is. Finally, the constant rate control interval removes the impact of RTT.

We have done extensive simulations and experimental studies to verify UDT's performance characteristics. UDT can utilize high bandwidth very efficiently and fairly. The intra-protocol fairness is maintained even between flows with different RTTs. This is very important for many distributed applications.

To benefit a broader set of network developers and researchers, we have expanded our UDT protocol and associated implementation to accommodate various congestion control algorithms.

In the short term, UDT is a practical solution to the data transfer problem in the emerging distributed data intensive applications. In the long term, because of the long time lag in deployment of in-kernel protocols but the fast speed with which new applications are emerging, UDT will still be a very useful tool in both application development and network research.

CITED LITERATURE

- [1] A. Aggarwal, S. Savage, and T. Anderson: Understanding the performance of TCP pacing. *IEEE Infocom '00*, Tel Aviv, Israel, Mar. 26-30, 2000.
- [2] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, S. Tuecke: GridFTP protocol specification. *GGF GridFTP Working Group*. Document, September 2002.
- [3] M. Allman and V. Paxson: On estimating end-to-end network path properties. *ACM SIGCOMM '99*, Cambridge, MA, Aug. 30 - Sep. 3, 1999.
- [4] M. Allman, V. Paxson, and W. Stevens: TCP congestion control. *IETF*, RFC 2581, April 1999.
- [5] David G. Andersen, Deepak Bansal, Dorothy Curtis, Srinivasan Seshan, and Hari Balakrishnan: System support for bandwidth management and content adaptation in Internet applications. *Proc. 4th USENIX Conference on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.
- [6] Cosimo Anglano, Massimo Canonico: A comparative evaluation of high-performance file transfer systems for data-intensive grid applications. *WETICE 2004*, pp. 283-288.
- [7] 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.
- [8] A. Banerjee, 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.
- [9] Amitabha Banerjee, Wu-chun Feng, Biswanath Mukherjee, and Dipak Ghosal: Routing and scheduling large file transfers over lambda grids. *3rd International Workshop on Protocols for Fast Long-Distance Networks (PFLDNet 2005)*, Feb., 2005
- [10] Deepak Bansal and Hari Balakrishnan: Binomial congestion control algorithm. *Proc. IEEE INFOCOM Conf.*, Anchorage, AK, April 2001.
- [11] Sumitha Bhandarkar, Saurabh Jain, and A. L. Narasimha Reddy: Improving TCP performance in high bandwidth high RTT links using layered congestion control. *Proc. PFLDNet 2005 Workshop*, February 2005.
- [12] Edoardo Biagioni: A structured TCP in standard ML. *Proc. the ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications*, pages 36-45, London, England, 1994.
- [13] Robert Braden, Aaron Falk, Ted Faber, Aman Kapoor, and Yuri Pryadkin: Studies of XCP deployment issues. *Proc. PFLDNet 2005 Workshop*, February 2005.
- [14] 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.
- [15] T. Braun and C. Diot: Protocol implementation using integrated layer processing. *ACM SIGCOMM '95*, Cambridge, MA, Aug. 28 - Sep. 1, 1995.
- [16] D. Cheriton: VMTP: A transport protocol for the next generation of communication systems. *ACM SIGCOMM '87*, Stowe, VT, Aug. 1987.
- [17] A. Chien, T. Faber, A. Falk, J. Bannister, R. Grossman, J. Leigh: Transport protocols for high performance: Whither TCP?, *Communications of the ACM*, Volume 46, Issue 11, November, 2003, pages 42-49.
- [18] 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.

- [19] J. Chu: Zero-copy TCP in Solaris. Proc. *USENIX Annual Conference '96*, San Diego, CA, Jan. 1996.
- [20] D. Clark, M. Lambert, and L. Zhang: NETBLT: A high throughput transport protocol. *ACM SIGCOMM '87*, Stowe, VT, Aug. 1987.
- [21] D. Clark and D. Tennenhouse: Architectural considerations for a new generation of protocols. *ACM SIGCOMM '90*, Philadelphia, PA, Sep. 24-27, 1990.
- [22] Tom DeFanti, Cees de Laat, Joe Mambretti, Kees Neggers, Bill St. Arnaud: TransLight: a global-scale Lambda Grid for e-science. *Communications of the ACM*, Volume 46, Issue 11, (November 2003), Pages: 34 - 41.
- [23] Phillip M. Dickens: FOBS: A lightweight communication protocol for grid computing. *Euro-Par 2003*: 938-946.
- [24] C. Dovrolis, P. Ramanathan, D. Moore: What do packet dispersion techniques measure? Proc. *IEEE Infocom*, April 2001.
- [25] A. Edwards and S. Muir: Experiences implementing a high-performance TCP In user-space. Proc. *ACM SIGCOMM 1995*, Cambridge, MA, pages 196 - 205.
- [26] David Ely, Stefan Savage, and David Wetherall: Alpine: A user-level infrastructure for network protocol development. Proc. *3rd USENIX Symposium on Internet Technologies and Systems (USITS 2001)*, pages 171-183, March 2001.
- [27] W. Feng and P. Tinnakornsrisuphap: The failure of TCP in high-performance computational grids. *SC '00*, Dallas, TX, Nov. 4 - 10, 2000.
- [28] S. Figueira, S. Naiksatam, H. Cohen, D. Cutrell, D. Gutierrez, D. B. Hoang, T. Lavian, J. Mambretti, S. Merrill, F. Travostino: DWDM-RAM: Enabling grid services with dynamic optical networks. *GAN '04 (Workshop on Grids and Networks)*, Chicago, IL, 19-22 April 2004.
- [29] S. Floyd: HighSpeed TCP for large congestion windows. *IETF, RFC 3649, Experimental Standard*, Dec. 2003.
- [30] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer: Equation-based congestion control for unicast applications. Proc. *ACM SIGCOMM 2000*, Stockholm, Aug. 2000.
- [31] S. Floyd, H. Henderson: The NewReno modification to TCP's fast recovery algorithm, *RFC 2582, IETF*, 1999.
- [32] S. Floyd and V. Jacobson: On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 3:115-156, 1992.
- [33] 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.
- [34] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky: An extension to the selective acknowledgment (SACK) option for TCP. *IETF RFC 2883, Proposed Standard*, July 2000.
- [35] 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.
- [36] S. Gorinsky and H. Vin: Extended analysis of binary adjustment algorithms. *Technical Report TR2002-39, Department of Computer Sciences, The University of Texas at Austin*, August 2002.
- [37] Yunhong Gu and R. L. Grossman: SABUL: A transport protocol for grid computing. *Journal of Grid Computing*, 2003, Volume 1, Issue 4, pp. 377-386.
- [38] Yunhong Gu and R. Grossman. UDT: A transport protocol for data intensive applications. *Internet Draft, work in progress*.

- [39] Yunhong Gu, Xinwei Hong, and Robert Grossman: Experiences in design and implementation of a high performance transport protocol. *SC 2004*, Nov 6 - 12, Pittsburgh, PA, USA.
- [40] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau: Deploying safe user-level network services with icTCP. *OSDI 2004*.
- [41] E. He, J. Leigh, O. Yu, T. A. DeFanti: Reliable Blast UDP: Predictable high performance bulk data transfer. *IEEE Cluster Computing 2002*, Chicago, IL 09/01/2002.
- [42] N. C. Hutchinson and L. L. Peterson: The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1): 64-76, Jan. 1991.
- [43] Van Jacobson, Michael J. Karels: Congestion avoidance and control. *Proc. the Sigcomm '88*, Stanford, CA, August, 1988.
- [44] R. Jain: The art of computer systems performance analysis: Techniques for experimental design, measurement, simulation, and modeling. *Wiley- Interscience*, New York, NY, April 1991.
- [45] R. Jain: A Delay-based approach for congestion avoidance in interconnected heterogeneous computer networks. *ACM SIGCOMM '89*, Austin, TX, Sep. 19-22, 1989.
- [46] M. Jain and C. Dovrolis. Pathload: A measurement tool for end-to-end available bandwidth. *Passive and Active Measurements (PAM) 2002 workshop*, pp 14-25, Fort Collins, CO.
- [47] N. Jain, M. Schawrtz, T. Bashkow: Transport protocol processing at GBPS rates. *SIGCOMM '90*, Philadelphia, PA, Sep. 24-27, 1990.
- [48] C. Jin, D. X. Wei, and S. H. Low: FAST TCP: motivation, architecture, algorithms, performance. *IEEE Infocom '04*, Hongkong, China, Mar. 2004.
- [49] 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.
- [50] D. Katabi, M. Hardley, and C. Rohrs: Internet congestion control for future high bandwidth-delay product environments. *ACM SIGCOMM '02*, Pittsburgh, PA, Aug. 19 - 23, 2002.
- [51] Frank Kelly: Fairness and stability of end-to-end congestion control. *European Journal of Control*, 9 (2003) 159-176.
- [52] F. P. Kelly, A.K. Maulloo, and D.K.H. Tan: Rate control in communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49 (1998), 237-252.
- [53] T. Kelly: Scalable TCP: Improving performance in highspeed wide area networks. *ACM Computer Communication Review*, Apr. 2003.
- [54] Eddie Kohler, Mark Handley, Sally Floyd, Jitendra Padhye: Datagram congestion control protocol (DCCP), <http://www.icir.org/kohler/dcp/>. Jan. 2005.
- [55] E. Kohler, M. F. Kaashoek, and D. R. Montgomery: A readable TCP in the Prolac protocol language. *Proc. SIGCOMM '99*, pages 3-13, Cambridge, Massachusetts, Aug. 1999.
- [56] Naveen Krishnaprasad, Venkatram Vishwanath, Shalini Venkataraman, Arun G.Rao, Luc Renambot, Jason Leigh, Andrew E.Johnson: JuxtaView - A tool for interactive visualization of large imagery on scalable tiled displays. *Proc. IEEE Cluster 2004*, San Diego, CA, September 20-23, 2004.
- [57] K. Kumazoe, Y. Hori, M. Tsuru, and Y. Oie: Transport protocol for fast long distance networks: Comparison of their performances in JGN. *SAINTE '04*, Tokyo, Japan, 26 - 30 Jan. 2004.

- [58] Kazumi Kumazoe, Katsushi Kouyama, Yoshiaki Hori, Masato Tsuru, Yuji Oie: Transport protocol for fast long-distance networks : Evaluation of penetration and robustness on JGNII. *The 3rd International Workshop on Protocols for Fast Long-Distance Networks, (PFLDnet 05)*, Lyon, France, Feb. 2005.
- [59] A. Kuzmanovic and E. W. Knightly: TCP-LP: A distributed algorithm for low priority data transfer. *IEEE Infocom '03*, San Francisco, CA, Mar. 30 - Apr. 3, 2003.
- [60] Kevin Lai, Mary Baker: Measuring link bandwidths using a deterministic model of packet delay. *SIGCOMM 2000*, pp. 283-294.
- [61] D. Leith: Linux TCP implementation issues in high-speed networks. *Technical report, Hamilton Institute, Ireland.* <http://www.hamilton.ie/net/LinuxHighSpeed.pdf>.
- [62] S. Leue and P. Oechslin: On parallelizing and optimizing the implementation of communication protocols. *IEEE/ACM Transactions on Networking* 4(1): 55-70 (1996).
- [63] D. Loguinov and H. Radha: End-to-end rate-based congestion control: Convergence properties and scalability analysis. *IEEE/ACM Transactions on Networking*, vol. 11, no. 4, August 2003.
- [64] Steven H. Low. A duality model of TCP and queue management algorithms. *IEEE/ACM Transactions on Networking* (TON), Volume 11, Issue 4, (August 2003), Pages: 525 - 536.
- [65] Steven H. Low, Larry L. Peterson, Limin Wang: Understanding TCP Vegas: a duality model. *ACM SIGMETRICS/Performance 2001*: 226-235
- [66] Kieran Mansley: Engineering a user-level TCP for the CLAN network. SIGCOMM 2003 workshop on Network-I/O convergence: experience, lessons, and implications.
- [67] J. Martin, A. Nilsson, and I. Rhee: Delay-based congestion avoidance for TCP. *ACM/IEEE Transactions on Networks*, June 2003.
- [68] M. Mathis, J. Heffner, and R. Reddy: Web100: Extended TCP instrumentation for research, education and diagnosis, *ACM Computer Communications Review*, Vol 33, Num 3, July 2003.
- [69] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow: TCP selective acknowledgment options. *IETF RFC 2018*, April 1996.
- [70] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott: The Macroscopic behavior of the congestion avoidance algorithm. *Computer Communications Review*, volume 27, number 3, July 1997.
- [71] 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*, Baltimore, MD, Nov. 16 - 22, 2002.
- [72] T. J. Ott, J. H. B. Kemperman, and M. Mathis: The stationary behavior of ideal TCP congestion avoidance. *Proc. IEEE INFOCOM '99*, New York, 1999.
- [73] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose: Modeling TCP throughput: a simple model and its empirical validation. *ACM SIGCOMM '98*, Vancouver, BC, Canada, Sep. 2 - 4, 1998.
- [74] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack: Upgrading transport protocols using untrusted mobile code. *Proc. the 19th ACM Symposium on Operating System Principles*, October 19-22, 2003.
- [75] V. Paxson: End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, Vol.7, No.3, pp. 277-292, June 1999.
- [76] V. Paxson and M. Allman: Computing TCP's retransmission timer. *IETF RFC 2988*, Nov. 2000.
- [77] J. Postel: User datagram protocol. *RFC 768, IETF*, 1980.

- [78] Prashant Pradhan, Srikanth Kandula, Wen Xu, Anees Shaikh, Erich Nahum: Daytona: A user-level TCP stack. <http://nms.lcs.mit.edu/~kandula/data/daytona.pdf>.
- [79] Ravi Prasad, Manish Jain and Constantinos Dovrolis: Effects of interrupt coalescence on network measurements. *PAM2004*, Antibes Juan-les-Pins, France, April 19-20, 2004.
- [80] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr: A framework for protocol composition in Horus. *Proc. the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 80-89, Ottawa, Ontario, Canada, 2-23 Aug. 1995.
- [81] 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.
- [82] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson: RTP: A transport protocol for real-time applications. *IETF, RFC 1889*.
- [83] R. N. Shorten, D. J. Leith: H-TCP: TCP for high-speed and long-distance networks. *Proc. PFLDNet 2004*, Argonne, IL, 2004.
- [84] H. Sivakumar, S. Bailey, R. L. Grossman. Pockets: The case for application-level network striping for data intensive applications using high speed wide area networks. *SC '00*, Dallas, TX, Nov. 2000.
- [85] H. Sivakumar, R. L. Grossman, M. Mazzucco, Y. Pan, Q. Zhang: Simple available bandwidth utilization library for high-speed wide area networks. *Journal of Supercomputing*, 2004.
- [86] R. Srikant: The mathematics of Internet congestion control. *Birkhauser*, 2004.
- [87] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson: Stream control transmission protocol. *IETF RFC 2960*, Oct. 2000.
- [88] W. Stevens: TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms, *RFC 2001, IETF*, 1997.
- [89] T. Strayer, B. Dempsey, and A. Weaver: XTP – the Xpress Transfer Protocol. *Addison-Wesley Publishing Company*, 1992.
- [90] A. Szalay, J. Gray, A. Thakar, P. Kuntz, T. Malik, J. Raddick, C. Stoughton, J. Vandenberg: The SDSS SkyServer - Public access to the Sloan digital sky server data. *ACM SIGMOD 2002*.
- [91] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska: Implementing network protocols at user level, *IEEE/ACM Transactions on Networking*, 1(5): 554--565, October 1993.
- [92] M. Veeraraghavan, X. Zheng, H. Lee, M. Gardner, W. Feng: CHEETAH: Circuit-switched high-speed end-to-end transport architecture. *Proc. of Opticomm 2003*, Oct. 13-17, 2003. Dallas, TX.
- [93] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting: A configurable and extensible transport protocol. *IEEE Infocom 2001*, April 22-26, 2001. Anchorage, Alaska, April 2001.
- [94] Ryan Wu and Andrew Chien: GTP: Group transport protocol for lambda-grids. *Proc. the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)*, Chicago, Illinois, April 2004.
- [95] Xinran Wu, Andrew A. Chien, Matti A. Hiltunen, Richard D. Schlichting, and Subhabrata Sen: High performance configurable transport protocol for grid computing. *Proc. the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*.
- [96] Qishi Wu, Nageswara S. V. Rao: Protocol for high-speed data transport over dedicated channels. *Third International Workshop on Protocols for Long-Distance Networks (PFLDnet 2005)*, Lyon, France, Feb. 2005.

- [97] C. Xiong, Leigh, J., He, E., Vishwanath, V., Murata, T., Renambot, L., DeFanti, T.: LambdaStream - a data transport protocol for streaming network-intensive applications over photonic networks. *Third International Workshop on Protocols for Long-Distance Networks (PFLDnet 2005)*, Lyon, France, Feb. 2005.
- [98] L. Xu, K. Harfoush, and I. Rhee: Binary increase congestion control for fast long-distance networks. *IEEE Infocom '04*, Hongkong, China, Mar. 2004.
- [99] Lixia Zhang, Scott Shenker, David D. Clark: Observations on the dynamics of a congestion control algorithm: The Effects of two-way traffic. *ACK SIGCOMM 1991*, pp. 133-147.
- [100] M. Zhang, B. Karp, S. Floyd, and L. Peterson: RR-TCP: A reordering-robust TCP with DSACK. *Proc. the Eleventh IEEE International Conference on Networking Protocols (ICNP 2003)*, Atlanta, GA, November 2003.
- [101] Y. Zhang, E. Yan, and S. K. Dao: A measurement of TCP over long-delay network. *The 6th International Conference on Telecommunication Systems, Modeling and Analysis*, Nashville, TN, March 1998.
- [102] X. Zheng, A. P. Mudambi, and M. Veeraraghavan: FRTP: Fixed rate transport protocol -- A modified version of SABUL for end-to-end circuits. *Pathnets2004 on Broadnet2004*, Sept. 2004, San Jose, CA.
- [103] Globus XIO: <http://www-unix.globus.org/toolkit/docs/3.2/xio/index.html>. Retrieved on Apr. 3, 2005.
- [104] Intel 82093AA I/O advanced programmable interrupt controller (I/O APIC), <http://www.intel.com/design/chipsets/datashts/290566.htm>.
- [105] Intel VTune Performance Analyzer, <http://www.intel.com/software/products/vtune>.
- [106] NS-2, <http://www.isi.edu/nsnam/ns/>.
- [107] Teraflow Testbed, <http://www.teraflowtestbed.net>.
- [108] UDT source release. <http://udt.sourceforge.net>.

VITA

- NAME: Yunhong Gu
- EDUCATION: Ph.D., University of Illinois at Chicago (UIC), Chicago, Illinois, 2005
M.Eng., Beijing University of Aeronautics and Astronautics (BUAA), Beijing, China, 2001
B.Eng., Hangzhou Institute of Electronics Engineering, Hangzhou (HIEE), Hangzhou, China, 1998
- HONORS: Dean's Scholar Award & Fellowship, UIC, Aug. 2005 – Dec. 2005
Guanghua Scholarship, BUAA, Oct. 1999
Excellent Graduate (*Cum Laude*) of HIEE, June 1998
Excellent Graduate of Zhejiang Province, China, May 1998
Distinguished Student Scholarship, HIEE, Mar. 1998
Top Prize in Chinese Undergraduate Mathematical Contest in Modeling, Dec. 1997
- PROFESSIONAL EXPERIENCES: National Center for Data Mining / Laboratory for Advanced Computing, UIC, Chicago, Illinois
Research Assistant, Aug. 2001 – Dec. 2005
V2 Technologies, Inc, Beijing, China
R&D Engineer, Apr. 2001 – Aug. 2001
Digital Media Laboratory, BUAA, Beijing, China
Research Assistant, Sep. 1998 – Apr. 2001
- REFEREED PUBLICATIONS: [1] Yunhong Gu and Robert L. Grossman: Supporting Configurable Congestion Control in Data Transport Services. *SC 05*, Nov 12 – 18, 2005, Seattle, WA, USA.
[2] Yunhong Gu and Robert L. Grossman: Optimizing UDP-Based Protocol Implementations. *Proceedings of the Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2005)*, Feb 3 – 4, 2005, Lyon, France.
[3] Yunhong Gu, Xinwei Hong, and Robert Grossman: Experiences in Design and Implementation of a High Performance Transport Protocol. *SC 04*, Nov 6 - 12, Pittsburgh, PA, USA. Best student paper nominee.
[4] Yunhong Gu, Xinwei Hong, and Robert Grossman: An Analysis of AIMD Algorithms with Decreasing Increases. *Gridnets 2004*, First Workshop on Networks for Grid Applications, Oct. 29, San Jose, CA, USA.
[5] Robert L. Grossman, Yunhong Gu, Dave Hanley, Xinwei Hong, and Parthasarathy Krishnaswamy: Experimental Studies of Data Transport and Data Access of Earth Science Data over Networks with High Bandwidth Delay Products. *Computer Networks*, Volume 46, Issue 3, Oct. 2004, pp. 411-421.
[6] Yunhong Gu and Robert L. Grossman: SABUL: A Transport Protocol for Grid Computing. *Journal of Grid Computing*, 2003, Volume 1, Issue 4, pp. 377-386.

- [7] Robert L. Grossman, Yunhong Gu, Xinwei Hong, Antony Antony, Johan Blom, Freek Dijkstra, and Cees de Laat: Teraflows over Gigabit WANs with UDT. *Journal of Future Computer Systems*, Elsevier Press, 2005.
- [8] Robert L. Grossman, Yunhong Gu, Chetan Gupta, David Hanley, Xinwei Hong, and Parthasarathy Krishnaswamy. Open DMIX: High Performance Web Services for Distributed Data Mining. *7th International Workshop on High Performance and Distributed Mining*, in association with the Fourth International SIAM Conference on Data Mining.
- [9] R. L. Grossman, Y. Gu, D. Hanley, X. Hong, and G. Rao: Open DMIX - Data Integration and Exploration Services for Data Grids, Data Web and Knowledge Grid Applications. *Proceedings of the First International Workshop on Knowledge Grid and Grid Intelligence (KGGI 2003)*, W. K. Cheung and Y. Ye, editors, pages 16-28.
- [10] R. L. Grossman, Y. Gu, D. Hanley, X. Hong, D. Lillethun, J. Levera, J. Mambretti, M. Mazzucco, and J. Weinberger: Global Access to Large Distributed Data Sets using Photonic Data Services. *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, Los Alamitos, California, pages 62-66.
- [11] Asvin Ananthanarayan, Rajiv Balachandran, Yunhong Gu, Robert Grossman, Xinwei Hong, Jorge Levera, Marco Mazzucco: Data Webs for Earth Science Data. *Parallel Computing*, Volume 29, 2003, pages 1363-1379.
- [12] Robert L. Grossman, Yunhong Gu, Dave Hanley, Xinwei Hong, Dave Lillethun, Jorge Levera, Joe Mambretti, Marco Mazzucco, and Jeremy Weinberger: Experimental Studies Using Photonic Data Services at IGrid 2002. *Journal of Future Computer Systems*, 2003, Volume 19, Number 6, pages 945-955.
- [13] J. Mambretti, J. Weinberger, J. Chen, E. Bacon, F. Yeh, D. Lillethun, R. Grossman, Y. Gu, M. Mazzucco: The Photonic TeraStream: Enabling Next Generation Applications Through Intelligent Optical Networking at iGrid 2002. *Journal of Future Computer Systems*, Elsevier Press, Volume 19, Number 6, pages 897-908.
- NON-REFEREED PUBLICATIONS:
- [14] Stanislav Shalunov, Lawrence D. Dunn, Yunhong Gu, Steven Low, Injong Rhee, Steven Senger, Bartek Wydrowski, and Lisong Xu: Design Space for a Bulk Transport Tool. *Technical Report, Internet2 Transport Group*, 2005.
- [15] Yunhong Gu and Robert L. Grossman, UDT: A Transport Protocol for Data Intensive Applications. Internet Draft. Work in Progress.
- [16] Yunhong Gu and Robert L. Grossman: UDT: An Application Level Transport Protocol for Grid Computing. *PFLDNet 2004, the Second International Workshop on Protocols for Fast Long-Distance Networks*, Feb. 13 - 14, Chicago, IL, USA. Extended Abstract.
- [17] R. L. Grossman, Y. Gu, D. Hanley, X. Hong, D. Lillethun, J. Levera, J. Mambretti, M. Mazzucco, and J. Weinberger: Photonic Data Services: Integrating Path, Network and Data Services to Support Next Generation Data Mining Applications. *Data Mining: Next Generation Challenges and Future Directions*, H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, editors, AAAI Press, 2004.
- [18] Geoffrey Fox, Harshwardhan Gadgil, Shrideep Pallickara, Marlon Pierce, Robert L. Grossman, Yunhong Gu, David Hanley, Xinwei Hong: High Performance Data Streaming in Service Architecture. *CGL Technical Report, Indiana University*, July 2004.
- [19] Mathieu Goutelle (editor), Yunhong Gu, Eric He (editor), Sanjay Hegde, Rajkumar Kettimuthu, Jason Leigh, Pascale Vicat-Blanc/Primet, Michael Welzl (editor), and Chaoyue Xiong: A Survey of Transport Protocols other than Standard TCP. *Global Grid Forum Data Transport Research Group Document*. February 2004.