UDT: A Transport Protocol for Data Intensive Applications

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC 2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt
The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Abstract

This document proposes UDT, or UDP based Data Transfer protocol, which can be used in high bandwidth-delay product (BDP) networks to utilize the abundant network resource efficiently. Potential users are people who work with distributed data intensive applications, such as scientific computing on computational grids.

The current widely used TCP version (TCP Reno with SACK option) does not work well in such environments due to its slow discovery and recovery to the available bandwidth as the network BDP increases, as well as due to its fairness problem when concurrent TCP flows have different RTTs.

UDT is an application layer solution by adding congestion control and reliability control to UDP. UDT combines rate control and window control and it uses bandwidth estimation techniques to dynamically configure the control parameters.
1. Introduction

As the network BDP increases, standard TCP becomes inefficient. This is because its AIMD (additive increase multiplicative decrease) algorithm reduces the TCP congestion window drastically but fails to recover it to the available bandwidth fast. Theoretical flow level analysis has shown that TCP becomes more vulnerable to packet loss as the BDP increases higher [LM97].

On the other hand, the unfairness of RTT bias inherent in TCP congestion control also becomes more serious than that on traditional Internet. Performance of distributed applications can be limited by the slowest data flow.

By now improvements on the standard TCP still fails to satisfy the efficiency and fairness (especially the RTT bias problem) objectives in high BDP environments. Such TCP modifications as presented in RFC 1423 (high-performance extensions), RFC 2018 (SACK), RFC 2582 (New Reno), RFC 2883 (D-SACK), and RFC 2988 (RTO calculation) do improve the efficiency more or less, but the fundamental problem of the TCP AIMD algorithm is unsolved.
Considering the background above, new end-to-end approach is needed to support the high performance bulk data transfer in high BDP networks. The congestion control algorithm to be presented in this document could have been deployed directly above IP to form a new transport layer protocol. However, by building it above UDP to form an application layer solution it is more easily to be deployed.

The UDT congestion control algorithm combines rate control and window control (namely flow control in UDT), where the former tunes the inter-packet time and the latter limits the number of unacknowledged packets. The parameters used in these two mechanisms are updated by bandwidth estimation technique, which is derived from receiver based packet pair [P99], in real time. Meanwhile, the rate control interval is constant so that RTT bias is removed.

This document defines UDTÂs protocol specification. The detailed performance analysis can be found in [GG03], and an example implementation can be found at [UDT].

2. Target Use Scenarios and Design Goals

UDT is expected to be used in the situations where small number of bulk sources share the abundant bandwidth. Typical use scenario is the computational grids [FKT01] built on wide area optical networks. A few institutes run their data intensive distributed application on the networks, such as remote access to scientific instruments, distributed data mining, and high resolution media streams.

The main objectives of UDT are efficiency, fairness, and robustness. Single or small number of UDT flows should utilize all the available bandwidth provided by the optical links. Meanwhile, all concurrent flows must share the bandwidth fairly, independent of the different bottleneck bandwidth, start time, or RTTs. The objective of robustness is raised from the fact that entering or leaving of a flow can cause significant bandwidth changes in such environments, since the total number of flows are small. Efficiency and robustness must be observed even the available bandwidth changes frequently.

UDT is NOT used to replace TCP on traditional 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 should not occupy more bandwidth than TCP does unless the TCP flows fail to utilize their fair share. 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.
3. Protocol Specification

3.1 Overview

UDT is a duplex transport protocol. Each UDT entity has two 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 packet and control packets, and sends out control packets according to the received packets as well.

Application data are sent from the sender of one UDT entity to the receiver of another UDT entity. The control packets for the data transfer in this direction are only sent in the reverse direction as feedback, from the receiver to the peer side's receiver.

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

Rate control updates the inter-packet time every constant interval, and flow control updates the flow window size each time an acknowledgement packet is received. Except for the timer used for packet scheduling, all other timers are maintained in the receiver by self-clocking.

3.2 Packet Structures

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

The data packet header structure is as following.

```
0                   1                   2                   3
                      |                         |
                      | Sequence Number          |
                      +----------------------------+
```

The data packet header starts with 0. Sequence number is the only content in the header of a UDT data packet. It uses the following 31 bits after the flag bit.

UDT packs application data in path MTU size (including all headers) as long as it is possible. A sequence number is assigned to each packet according to the increasing order that the packet is sent out. That is, UDT uses packet based sequencing but not byte based...
sequencing as TCP does. The actual packet size can be told by the UDP interface at the receiver.

UDT has no specific mechanism to find path MTU. Implementations should use existing schemes such as RFC 1191 (Path MTU Discovery) or leave it to applications to set it manually. On the other hand, applications should try their best to allow UDT to pack fixed MTU size data packet. Packet with size less than MTU are allowed but should be kept as few as possible.

If the flag bit of a UDT packet is 1, then it is a control packet and parsed according to the following structure.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1|type |        Reserved       |   Loss Length / ACK Seq. No.  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                 Control Information Field                     ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

There are 6 types of control packets in UDT and the type information is put in bit field 1 - 3 of the header. The contents of the following fields depend on the packet type.

TYPE 000: Protocol Connection Handshake
bits 16-31: Undefined
Control Info: Handshake information (MTU and maximum flow window size)

TYPE 001: Keep-alive
bits 16-31: Undefined
Control Info: None

TYPE 010: Acknowledgement (ACK)
bits 16-31: Sequence number of the ACK
Control Info:
1) The sequence number to which (but not include) all the previous packets have been received
2) RTT
3) Packets receiving rate (in number of packets per second)
4) Estimated link capacity (in number of packets per second)

TYPE 011: Negative Acknowledgement (NAK)
bits 16-31: Number of loss carried in the packet
Control Info: Compressed loss information

TYPE 100: Unused
TYPE 101: Unused
TYPE 110: Acknowledgement of Acknowledgement (ACK2)
bits 16-31: The ACK sequence number of the ACK that triggers this ACK2 packet.

Control Info: None

TYPE 111: Explained by bits 4 – 15, reserved for future use.

3.3 Timers

There are totally 5 timers used in UDT:

SND timer: Used in rate control to schedule the data packet sending. Its interval is updated by rate control scheme of UDT.

SYN timer: Used to trigger a rate control event. Its interval is a fixed and constant value.

ACK timer: Used to trigger an acknowledgement (ACK). Its interval is same as SYN interval.

NAK timer: Used to trigger a negative acknowledgement (NAK). Its interval is updated to the current RTT value each time the SYN timer is expired.

EXP timer: Used to trigger data packets retransmission and maintain connection status. Its interval (exp-int) is updated each time the timer is expired with the following formula:

\[ \text{exp-int} = (\text{exp-count} + 1) \times \text{RTT} + \text{SYN}. \]

\[ \text{exp-count} = \text{exp-count} + 1. \]

where exp-count is initialized to 1 and reset to 1 each time a packet is received. It is used to record the number of continuous time-out events.

The SYN interval is 0.01 seconds and it must be observed by all UDT implementations. This is an empirical value.

SND timer needs a high precision implementation, whereas the rest 4 timers are self clocked and they are queried each time a time bounded UDP receiving is returned.

3.4 The Sender's Algorithm

Data Sending:
1) If the sender's loss list is not empty, retransmit the first packet in the list and remove it from the list. The sender's loss list is a data structure to store the sequence numbers of the lost packets fed back by the receiver through NAK packets. Go to 5).
2) If there are not application data to be sent, sleep until new data comes.
3) If the number of unacknowledged packets does not exceeds the flow window size, pack a new data packet and send out. Otherwise, go to 1).
4) If the sequence number of the current packet is 16n, where n is an integer, go to 2).
5) If this is the first packet sent after last time when the sending rate is decreased, sleep SYN time.
6) Sleep the inter-packet time. Go to 1).

3.5 The Receiver’s Algorithm

Data Receiving:
1) Query the ACK, NAK, SYN, and EXP timers. Process the proper event if any of them expires.
2) Start time bounded UDP receiving. (The UDP time-out value should be set by implementation according to the balance of system responsiveness and overhead.) If no packet arrives, go to 1).
3) Reset the variable of exp-count to 1.
4) Check the flag bit of the packet header. If it is a control packet, process it according to its type and go to 1).
5) If the sequence number of the current data packet is \(16n + 1\), where \(n\) is an integer, record the time interval between this packet and the last data packets.
6) Record the packet arrival time.
7) a. If the sequence number of the current data packet is greater than the largest sequence number ever received plus 1, put all the sequence numbers between (but excluding) these two values into the receiver’s loss list and send them to the sender in an NAK packet.
   b. Otherwise, if the sequence number of the current data packet is less than the largest sequence number ever received, remove it from the receiver’s loss list.
8) If the size of the current data packet does not equal to MTU, update the receiver’s protocol buffer properly. Go to 1).

On ACK timer expired:
1) Find the sequence number prior to which all the packets have been received by the receiver (ACK number) according to the following rule: if the receiver’s loss list is empty, the ACK number is the largest sequence number ever received plus 1; otherwise it is the smallest sequence number in the loss list.
2) If (a) the ACK number equals to the largest ACK number ever acknowledged by ACK2, or (b) it is equal to the ACK number in the last ACK and the time interval between this two ACK packets is less than 2 RTTs, stop (do not send this ACK).
3) Assign this ACK a unique increasing ACK sequence number.
4) Calculate the packet arrival speed according to the following algorithm:
   Calculate the median value of the last 16 packet arrival intervals (AI), and the packet arrival speed is \(1/AI\) (number of packets per second).
5) Calculate the estimated link capacity according to the following algorithm:
Calculate the median value of the last 16 packet pair intervals (PI), and the link capacity is 1/PI (number of packets per second).

6) Pack the ACK sequence number, ACK number, RTT, packet arrival speed, and estimated link capacity into the ACK packet and send it out.

7) Record the ACK sequence number, ACK number and the departure time of this ACK in an ACK history window.

On NAK timer expired:
Search the receiver’s loss list, find out all those sequence numbers whose last feedback time is $k \times RTT$ before, where $k$ is initialized as 2 and increased by 1 each time the number is fed back. Compress and send these numbers back to the sender in an NAK packet.

On SYN timer expired:
Update the inter-packet time by rate control algorithm.

On EXP timer expired:
1) Put all the unacknowledged packets into the sender’s list.
2) Update the EXP timer interval.
3) If exp-count > 16, which means that the connection is broken, close the UDT connection and exit.
4) If the sender’s loss list is empty, send a keep-alive packet to the peer side.
5) Reset the EXP timer.

On ACK packet received:
1) Update the largest acknowledged sequence number.
2) Update RTT by: $RTT = (RTT \ast 7 + \text{rtt}) / 8$, where rtt is the value carried in the ACK.
3) Update estimated link capacity: $B = (B \ast 7 + b) / 8$, where b is the value carried in the ACK.
4) Update flow window size by flow control.
5) Send back an ACK2 with the same ACK sequence number in this ACK.
6) Reset EXP timer.

On NAK packet received:
1) Add all sequence numbers carried in the NAK into the sender’s loss list.
2) Update the inter-packet time by rate control.
3) Reset EXP timer.

On ACK2 packet received:
1) Locate the related ACK in the ACK history window according to the ACK sequence number in this ACK2.
2) Update the largest ACK number ever been acknowledged.
3) Calculate new rtt, record it according to the ACK2 arrival time and the ACK departure time, and update the RTT value as: \( \text{RTT} = \frac{\text{RTT} \times 7 + \text{rtt}}{8} \).

### 3.6 Rate Control Algorithm

Before the first NAK packet is received, keep inter-packet time as 0, i.e., rate control is not activated. This is the slow start phase for congestion control. After that, the following algorithm is activated.

On SYN timer expired:
1) Calculate the loss rate during the last SYN time according to the number of packets sent and the number of total lost packets fed back in NAK. If the loss rate is greater than 0.1\%, stop (inter-packet time is not updated).
2) The number of sent packets to be increased in the next SYN time (inc) is calculated as:
   
   \[
   \text{inc} = \begin{cases} 
   1/\text{MTU}; & (B \leq C) \\
   \max(10^{\text{ceil}(\log_{10}((B-C) \times \text{MTU} \times 8))} \times \beta/\text{MTU}, 1/\text{MTU}); & \text{else}
   \end{cases}
   \]

   where \( B \) is the estimated link capacity and \( C \) is the current sending speed. Both are counted as packets per second. \( \beta \) is a constant empirical value of 0.0000015. MTU is counted in bytes.
3) The inter-packet time (I) is updated as:
   \[
   I = \frac{I \times \text{SYN}}{I \times \text{inc} + \text{SYN}}.
   \]

These three parameters are used in rate decrease, and their initial values are in the parentheses: \( \text{dec} (1) \), \( \text{e} (4) \), \( \text{last-dec-seq} (-1) \).

On NAK packet received:
1) If the largest lost sequence number in the NAK is greater than last-dec-seq:
   1.1) Update inter-packet time \( I = I \times 1.125 \);
   1.2) Reset the variables \( \text{dec} = 1 \), \( \text{e} = 4 \);
   1.3) Record the current largest sent sequence number (last-dec-seq), which is -1 initially.
2) Otherwise, increase \( \text{dec} \) by 1. If \( \text{dec} \geq 2^\text{e} \), update \( I = I \times 1.125 \) and increase \( \text{e} \) by 1.

### 3.7 Flow Control Algorithm

The flow control window size (W) is 2 initially.

On ACK packet received:
1) If no NAK has been ever received, the flow window size is updated to the total number of acknowledged packets.
2) Otherwise, it is updated as:
   \[
   W = \frac{W \times 7 + \text{AS} \times (\text{SYN} + \text{RTT})}{8}.
   \]
where AS is the packet arrival speed carried in ACK.

3) Limit W to the maximum window size of UDT.

3.8 Connection Setup

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 set by applications expires.

The server, when receiving a handshake packet, compares the MTU and maximum window size with its own values and set its own values as the smaller ones. The result values are then sent back to the client by a handshake packet. The server is ready for sending/receiving data right after this step is finished. However, it must send back response packet as long as it receives any handshakes from the same client.

The client can start sending/receiving data once it gets response handshake packet from the server.

3.9 Loss List Compression Scheme

The loss information carried in an NAK packet is an array of 32-bit integers. If an integer in the array is a normal sequence number (1st bit is 0), it means that the packet with this sequence number is lost; if the 1st bit is 1, it means all the packets starting from (including) this number to (including) the next number in the array (whose 1st bit must be 0) are lost.

4. Efficiency and Fairness Characters

UDT can utilize almost full available bandwidth independent of link capacity, RTT, and background coexisting flows, given the link bit error rate (BER) of the current wired networks.

Obvious drop in UDT performance will appear as the link BER exceeds $10^{-5}$. Therefore, UDT is not suitable for wireless networks.

UDT strictly satisfies max-min fairness in single bottleneck network topologies. In multi-bottleneck scenarios, it guarantees that flows over smaller bottleneck links obtain at least half of its fair share according to max-min rule. RTT has little effect on the fairness.
When coexisting with concurrent bulk TCP flow, TCP can occupy more bandwidth than UDT does, except in one of the following 3 situations:

1) The network BDP is very large that TCP fails to utilize their fair share bandwidth. In this situation, UDT will occupy the bandwidth that TCP cannot utilize.

2) The link capacity is so small that UDT’s bandwidth estimation and delay increasing trend detection techniques does not work well. Simulation shows that this threshold link capacity is about 100 kbps.

3) In networks where DropTail queues are used along the paths, TCP’s bandwidth share decreases as the queue size increases. However, to reach about equal share with UDT, the queue size needed generally exceeds the practical queue sizes. When active queue management such as RED is used, TCP always take more bandwidth than UDT except when situation 1) or 2) is satisfied.

When short-time web like TCP flows coexist with small number of concurrent bulk UDT flows, the effect of UDT on TCP flows is very small.

More analysis can be found in [GG03].

Security Considerations

UDT has not its specific security mechanism, whereas it depends on the application to provide authentication and lower layer to provide security mechanisms, if necessary.

However, UDT implementations should check each arrived packets are from the expected source, since UDP is connectionless.

Normative References

None.

Informative References


[LM97] T. V. Lakshman and U. Madhow, ÂThe Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random LossÂ,


Acknowledgments

The authors thank Dr. Xinwei Hong for his cooperated work in the SABUL protocol, an early prototype of UDT protocol, which uses rate based UDP to transfer data and TCP to transfer control information. Dr. Marco Mazzucco and Dr. Sivakumar Harinath also contribute to the earlier idea of SABUL.

We appreciate those researchers, developers, and system administrators inside or outside LAC/NCDM for their helpful feedbacks on using UDT. We are grateful to StarLight, SARA, and Canarie for providing us the high speed networking test beds.

Author’s Addresses

Yunhong Gu
Laboratory for Advanced Computing
University of Illinois at Chicago
700 SEO, M/C 249, 851 S Morgan St
Chicago, IL 60607, USA
Phone: +1 (312) 996-0305
Email: yunhong@lac.uic.edu

Robert Grossman
Laboratory for Advanced Computing
University of Illinois at Chicago
727 SEO, M/C 249, 851 S Morgan St
Chicago, IL 60607, USA
Phone: +1 (312) 413-2176
Email: grossman@uic.edu