UDT: A Transport Protocol for Data Intensive Applications

Abstract

This document proposes a new UDP based Data Transfer protocol, also known as UDT. UDT can be used in high bandwidth-delay product (BDP) networks to utilize the abundant network resource efficiently and fairly. It is expected to be used in distributed data intensive applications over high-speed wide area networks, such as scientific computing on computational grids.

The current most widely used TCP version (TCP NewReno/SACK) does not work well in such environments due to its slow discovery and recovery to the available bandwidth as the network BDP increases. In addition, it exhibits "unfairness" when concurrent TCP flows have different round trip time (RTT).

UDT is an application layer solution by introducing new congestion control and reliability control. The protocol is defined upon UDP. The congestion control combines both rate-based and window-based methods and uses bandwidth estimation techniques to dynamically configure the control parameters.
Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Table of Contents

1. Introduction...................................................2
2. Target Use Scenarios and Design Goals..........................3
3. Protocol Specification............................................4
   3.1 Overview...................................................4
   3.2 Packet Structures.........................................4
   3.3 Timing......................................................6
   3.4 The Sender’s Algorithm...................................7
   3.5 The Receiver’s Algorithm................................8
   3.6 Rate Control Algorithm..................................11
   3.7 Flow Control Algorithm................................12
   3.8 Connection Setup and shutdown..........................12
   3.9 Loss Information Compression Scheme....................13
4. Efficiency and Fairness Characters............................14
   Security Considerations.......................................14
   Normative References..........................................15
   Informative References.......................................15
   Acknowledgments................................................15
   Author’s Addresses.............................................16

1. Introduction

As the network bandwidth-delay product (BDP) increases, conventional TCP implementations become 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 quickly. Theoretical flow level analysis has shown that TCP becomes more vulnerable to packet loss as the BDP increases higher [LM97].

Additionally, the unfairness of RTT bias inherent in TCP congestion control also becomes more serious in distributed data intensive applications. Concurrent TCP flows with different RTT will tend to unfairly share the available bandwidth. Although small BDP networks will share bandwidth relatively fairly using conventional TCP implementations, networks with large BDP tend to suffer severe unfairness issues under conventional TCP. This RTT bias severely limits the effectiveness of distributed computations based on high-speed wide-area networks as, for example, Grid-based computations in
an Internet2-based context.

As of today, improvements on the standard TCP still fail 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. HS TCP (RFC 3649) achieved high utilization of bandwidth in high BDP networks by radically changing the TCP congestion control algorithm, but the fairness problem still remains.

Considering the background above, new transport protocol is needed to support the high performance bulk data transfer in high BDP networks. We propose a new application level transport protocol, named UDT, or UDP based Data Transfer Protocol, together with a new congestion control algorithm.

This document presents the two orthogonal parts of the UDT protocol and the UDT congestion control algorithm. The application level protocol, layering over UDP, can use other congestion control algorithms; whereas the algorithm defined in this document can be implemented in other protocols, such as TCP.

An example implementation of the protocol can be found at [UDT]. Detailed performance analysis of the congestion control algorithm can be found in [GHG04].

2. Target Use Scenarios and Design Goals

UDT is expected to be used in the situations where a small number of bulk sources share abundant bandwidth. Typical use scenario is the computational grids [FKT01] built on wide area optical networks. A few institutes run their data intensive distributed applications 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 stability. Single, or small number of, UDT flows should utilize all the available bandwidth provided by the high-speed links, even if the bandwidth changes drastically. Meanwhile, all concurrent flows must share the bandwidth fairly, independent of the different bottleneck bandwidth, start time, or RTT. Stability requires that the packet sending rate should always converge to the available bandwidth very quickly, and congestion collapse must be avoided.

UDT is NOT used to replace TCP in the Internet where the bottleneck
bandwidth is relatively small and there are large amounts of multiplexed short life flows.

However, UDT aims to be TCP friendly - when coexisting with TCP, the bandwidth that UDT allocates should not exceed its fair share according to the max-min rule. (Note that the max-min rule allows UDT to allocate the available bandwidth that TCP fails to utilize in high BDP lossy links.) We feel that TCP friendliness is critical since TCP will always be in use in these high BDP networks, and UDT applications may sometimes run in the Internet.

3. Protocol Specification

3.1 Overview

UDT is duplex. 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 packets and control packets, and sends out control packets according to the received packets as well. Both the sender and the receiver share one UDP port for packet sending/receiving.

The receiver is also responsible for triggering and processing all control events, including congestion control and reliability control, and their related mechanisms such as RTT estimation, bandwidth estimation, acknowledging and retransmission.

UDT always tries to pack application data into fixed size packets, unless there is not enough data to be sent. Similar to TCP, this fixed packet size is named MSS (maximum segment size). Since UDT is supposed to be used to transfer bulk data stream, we assume that there is only a very small portion of irregular sized packets in a UDT session. MSS can be set up by applications (see Section 3.8) and the optimal value is the path MTU (including all packet headers).

The UDT congestion control algorithm combines rate control and window control (flow control), where the former tunes the packet-sending period and the latter limits the number of unacknowledged packets. The parameters used in rate control are updated by bandwidth estimation technique, which is derived from the receiver based packet pair method [P99], in real time. Meanwhile, the rate control period is constant in order to eliminate RTT bias. The flow control parameters depend on the data arrival speed at the peer side, in addition to the receiver’s free buffer size.

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 flag bit of a data packet is 0, whereas it is 1 for a control packet.

The data packet structure is shown as below.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|0|                     Packet Sequence Number                  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
~                   Application Data Payload                    ~
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Packet sequence number is the only content in the header of a UDT data packet. It is an unsigned integer using the following 31 bits after the flag bit. UDT uses packet based sequencing, i.e., the sequence number is increased by 1 for each non-retransmitted data packet. The sequence number is wrapped after it is increased to the maximum number (2^31 - 1). Following this 32-bit packet header is the payload for application data.

The control packet has 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       |          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 first 32 bits must exist in the packet header. The control Information Field contains zero (i.e., it does not exist; in this case it is noted as "none" in the specification) or multiple 32-bit unsigned integers, depending on the packet type.

UDT uses ACK sub-sequencing. Each ACK/ACK2 packet has a unique 16-bit sequence number, which is independent of the data packet sequence number. It 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.
TYPE 000: Protocol Connection Handshake
Control Info:
1) 32 bits: UDT version
2) 32 bits: initial sequence number
3) 32 bits: MSS (bytes)
4) 32 bits: maximum flow window size (bytes)

TYPE 001: Keep-alive
Control Info: None

TYPE 010: Acknowledgement (ACK)
bits 16-31: ACK sequence number
Control Info:
1) 32 bits: The packet sequence number to which (excluding) all the previous packets have been received
2) 32 bits: RTT (microseconds)
3) 32 bits: RTT variance, or RTTVar (microseconds)
4) 32 bits: Flow window size (number of packets)
5) 32 bits: Estimated link capacity (number of packets per second)

TYPE 011: Negative Acknowledgement (NAK)
Control Info:
32-bit integer array of compressed loss information (see Section 3.9).

TYPE 100: Reserved.
This type of control message is reserved for congestion warning that can be sent from the receiver to the sender. A congestion warning can be triggered by ECN, or a measurement of increasing trend in the packet delay.

TYPE 101: Shutdown
Control Info: None

TYPE 110: Acknowledgement of Acknowledgement (ACK2)
bits 16-31: ACK sequence number
Control Info: None

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

Note that for both data and control packets, the actual packet sizes can be ascertained from UDP header [UDP]. The packet size information can be used to derive the size of the payload in a data packet and the size of control information field in an NAK control packet.

3.3 Timing

UDT uses four timers in the receiver to trigger different periodical events, including rate control, acknowledgement, loss report (negative acknowledgement), and retransmission / connection maintenance.

Timers in UDT use the system time as origins and can process wrapping if the system time wraps. The UDT receiver actively queries the system time to check whether a timer is expired or not. For a certain
timer \( T \) in UDT with a period of \( TP \), suppose variable \( t \) is used to record the most recent time when \( T \) is set or reset. If \( T \) is set or reset at system time \( t_0 \) (\( t = t_0 \)), then at any time \( t_1 \), \( (t_1 - t >= TP) \) is the condition to check if \( T \) is expired.

The four timers are 1) RC timer, 2) ACK timer, 3) NAK timer, and 4) EXP timer. Their periods (or timeout values) are \( RCTP \), \( ATP \), \( NTP \), and \( ETP \), respectively.

RC timer is used to trigger periodical rate control. ACK timer is used to trigger periodical selective acknowledgements (ACK packets). Both \( RCTP \) and \( ATP \) are constant values: \( RCTP = ATP = 0.01 \) seconds.

NAK is used to trigger negative acknowledgements (NAK packets). Retransmission timer is used to trigger a data packet’s retransmission and maintain connection status. Their periods depend on the estimated RTT value. The \( ETP \) value also depends on the number of continuous EXP timeouts. The recommended initial value of RTT in UDT is 0.1 seconds and the values of \( NTP \) and \( ETP \) are initialized as: \( NTP = 3 \times RTT \), \( ETP = 3 \times RTT + ATP \).

The system time is queried after each time bounded UDP receiving (there will be additional necessary data processing time if a UDP packet is received) to check if any of the four timers is expired. The recommended granularity of their periods is microseconds. The timeout value of UDP receiving is a choice of implementation, depending on the tradeoff between the system burden caused by the looped queries on system time and the accuracy of event periods.

The rate control event updates the packet-sending period, \( STP \), which is used to schedule data packet sending in the UDT sender. Suppose a packet is to be sent at time \( t_0 \), then the next packet sending is scheduled to \( (t_0 + STP) \). In the other word, if the previous packet sending takes \( t’ \) time, the sender will wait for \( (STP - t’) \) for the next sending (if \( STP - t’ < 0 \), no waiting is needed). This waiting interval needs a high precision implementation. It is recommended that CPU clock cycle granularity be used for this timer.

3.4 The Sender’s Algorithm

Data Structures and Variables:
1) SND PKT History Window: A circular array that records the departure time of each data packet.
2) Sender’s Loss List: The sender’s loss list is a linked list used to store the sequence numbers of the lost packets fed back by the receiver in NAK packets. The numbers are stored in increasing order.

Data Sending Algorithm:
1) If the sender’s loss list is not empty, retransmit the first packet in the list and remove it from the list. Go to 5).
2) Wait until there is application data to be sent.
3) 
   a. If the number of unacknowledged packets exceeds the flow window size, go to 1).
   b. Pack a new data packet and send it out.
4) If the sequence number of the current packet is 16n, where n is an integer, go to 2).
5) Record the packet sending time in SND PKT History Window.
6) If this is the first packet sent after the last time when the sending rate is decreased, wait SYN time.
7) Wait (STP - t) time, where t is the total time used by step 1 to step 4. Go to 1).

3.5 The Receiver’s Algorithm

Data Structures and Variables:

1) Receiver’s Loss List: It is a linked list of tuple whose values include: the sequence number of a lost data packet, the latest feedback time of the lost packet, and the number of times the packet has been fed back. Values are stored in the increasing order of packet sequence numbers.
2) ACK History Window: A circular array of each sent ACK and the time it is sent out. By "circular" it means that the most recent value will overwrite the oldest one if there is no more free space in the array.
3) RCV PKT History Window: A circular array that records the arrival time of each data packet.
4) Packet Pair Window: A circular array that records the time interval between each probing packet pair.
5) LRSN: A variable to record the largest received data packet sequence number. LRSN is initialized to the initial sequence number (to be sent) minus 1.
6) exp-count: A variable to record number of continuous EXP timeout events and its initial value is 1.

Data Receiving Algorithm:
1) Query the system time to check if RC, ACK, NAK, or EXP timer is expired. If there is any, process the event accordingly (as described below in this section) and reset the expired timers.
2) Start time bounded UDP receiving. If no packet arrives, go to 1).
3) Set exp-count to 1 and update ETP: ETP = RTT + 4 * RTTVar + ATP.
4) If all sent data packets have been acknowledged, reset the EXP time variable.
5) Check the flag bit of the packet header. If it is a control packet, process it according to its type and go to 1).
6) 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 packet in Packet Pair Window.

7) Record the packet arrival time in PKT History Window.

8) a. If the sequence number of the current data packet is greater than \( LRSN + 1 \), 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. If the sequence number is less than \( LRSN \), remove it from the receiver’s loss list.

9) Update \( LRSN \). Go to 1).

On RC timer expired:
Update STP by rate control algorithm (see Section 3.6).

On ACK timer expired:
1) Find the sequence number prior to (excluding) 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 \( LRSN + 1 \); otherwise it is the smallest sequence number in the receiver’s loss list.

2) If (a) the ACK number is not greater than 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 \((RTT + 4\times RTTVar)\), stop (do not send this ACK).

3) Assign this ACK a unique increasing ACK sequence number. It is recommended that the ACK sequence number should be increased by 1 for each sent ACK in the sending order (start with 0) and wrapped after reaching the maximum value.

4) Calculate the packet arrival speed according to the following algorithm:
   - Calculate the median value of the last 16 packet arrival intervals (AI) using the values stored in PKT History Window.
   - In these 16 values, remove those either greater than \( AI\times 8 \) or less than \( AI/8 \). If more than 8 values are left, calculate the average of the left values \( AI' \), and the packet arrival speed is \( 1/AI' \) (number of packets per second); Otherwise, return 0.

5) Calculate flow control window for the peer side (W) according to Section 3.7. Then calculate effective flow window size as: \( \max(\min(W, \text{available receiver buffer size}), 2) \).

6) Calculate the estimated link capacity according to the following algorithm:
   - If the flow control quick start phase (see Section 3.7) is still ongoing, return 0. Otherwise,
     Calculate the median value of the last 16 packet pair intervals (PI) using the values in Packet Pair Window, and the link capacity is \( 1/PI \) (number of packets per second).

7) Pack the ACK sequence number, ACK number, RTT, RTT variance, effective flow window size, and estimated link capacity into the
ACK packet and send it out.
8) Record the ACK sequence number, ACK number and the departure time of this ACK in the ACK History Window.

On NAK timer expired:
1) Search the receiver’s loss list, find out all those sequence numbers whose last feedback time is \((k \times (RTT + 4 \times RTTVar))\) before, where \(k\) is the number of feedback times of this packet plus 1. If there is no loss to be fed back, stop.
2) Compress the sequence numbers obtained in step 1 (according to Section 3.9) and send them back to the sender in an NAK packet.
3) Stop flow control quick start phase if it not.

On EXP timer expired:
4) If the sender’s loss list is not empty, stop.
5) Put all the unacknowledged packets into the sender’s loss list.
6) If \((\text{exp-count} > 16)\) and the total time since last time a packet from the peer side is received has exceeded 3 seconds, or this time is greater than 3 minutes, which is supposed to be an indication that the connection is broken, close the UDT connection.
7) If there is no data packet that is not acknowledged, send a keep-alive packet to the peer side; otherwise, put all the sequence numbers of the unacknowledged packets into the sender’s loss list.
8) Update exp-count: \(\text{exp-count} = \text{exp-count} + 1\).
9) Update ETP: \(\text{ETP} = \text{exp-count} \times (RTT + 4 \times RTTVar) + \text{ATP}\).

On ACK packet received:
1) Update the largest acknowledged sequence number.
2) Update RTT and RTTVar by:
   \[
   \text{RTT} = \text{rtt} \\
   \text{RTTVar} = \text{rv}
   \]
   where \(\text{rtt}\) and \(\text{rv}\) are the RTT and RTTVar values carried in the ACK.
3) Update NTP and ETP:
   \[
   \text{NTP} = \text{RTT} + 4 \times \text{RTTVar} \\
   \text{ETP} = \text{exp-count} \times (\text{RTT} + 4 \times \text{RTTVar}) + \text{ATP}
   \]
4) Update estimated link capacity: \(B = (B \times 7 + b) / 8\), where \(b\) is the value carried in the ACK.
5) Send back flow window size to the value carried in this ACK.
6) Send back an ACK2 with the same ACK sequence number in this ACK.
7) Reset the EXP timer.

On NAK packet received:
1) Add all sequence numbers carried in the NAK into the sender’s loss list.
2) Update STP by rate control (see Section 3.6).
3) Reset the 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 according to the ACK2 arrival time and the ACK departure time, and update the RTT and RTTVar values as:
   \[ RTTVar = \frac{RTTVar \times 3 + \text{abs}(\text{rtt} - RTT)}{4} \]
   \[ RTT = \frac{RTT \times 7 + \text{rtt}}{8}. \]
   The initial values of RTT and RTTVar are 0.1 seconds and 0.05 seconds, respectively.
4) Update NTP and ETP: \[ \text{NTP} = \text{RTT}, \quad \text{ETP} = (\exp\text{-count} + 1) \times \text{RTT} + \text{ATP}. \]

On Keep-alive packet received:
Do nothing.

On Handshake/Shutdown packet received:
See Section 3.8.

3.6 Rate Control Algorithm

Rate Control Quick Start:
The STP is initially set to the minimum time precision (e.g., 1 CPU clock cycle or 1 microsecond). This is the quick start phase of UDT and it stops as soon as an ACK is received the estimated bandwidth carried by it is greater than zero. The packet sending period is then set as \(1/W\), where \(W\) is the flow window size carried by this ACK.

The quick start occurs only at the beginning of a UDT connection, and it never executed again in the UDT connection life. After quick start, the following algorithms start to work.

On RC timer expired:
1) If there is no ACK received during last RCTP, stop.
2) Calculate the loss rate during the last RCTP 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.
3) The number of sent packets to be increased in the next RCTP (inc) is calculated as:
   \[
   \text{inc} = \begin{cases} 
   1/\text{MSS} & \text{if } B \leq C \\
   \max(10^{\lceil \log_{10}(\text{B-C}) \cdot \text{MSS} \cdot 8 \rceil}) \times \text{Beta}/\text{MSS}, & \text{else}
   \end{cases}
   \]
   where \(B\) is the estimated link capacity and \(C\) is the current sending speed. Both are counted as packets per second. MSS is counted in bytes. Beta is a constant value of 0.0000015.
4) Update STP: \[ \text{STP} = \frac{\text{STP} \times \text{RCTP}}{\text{STP} \times \text{inc} + \text{RCTP}}. \]
5) Compute the real data sending period (rsp) from SND PKT History Window, If \((\text{STP} < 0.5 \times \text{rsp})\), set STP to \((0.5 \times \text{rsp})\).
6) If \((\text{STP} < 1.0)\), set STP to 1.0.
On NAK packet received:

Data Structures and Variables:
1) LSD: The largest sequence number that has been sent when last rate decrease occurred.
2) NumNAK: Number of NAKs since last time the LSD is updated.
3) AvgNAK: The moving average number of NAKs between two events when the largest sequence number is greater than LSD.
4) DR: A random number that satisfies the uniform distribution on [1, AvgNAK].

The rate control algorithm on NAK received:
1) If the largest lost sequence number in the NAK is greater than LSD:
   Increase the STP by 1/8: STP = STP * (1 + 1/8).
   Update AvgNAK: AvgNAK = (AvgNAK * 7 + NumNAK) / 8.
   Update DR.
   Reset NumNAK = 0;
   Record LSD.
2) Otherwise, increase NumNAK by 1, and if NumNAK % DR = 0:
   Increase the STP by 1/8: STP = STP * (1 + 1/8).
   Record LSD.

3.7 Flow Control Algorithm

The flow control window size (W) is 16 initially.

On ACK timer expired:
1) Flow Control Quick Start: If no NAK has been ever generated, or, W has not reached or exceeded 16 packets and AS > 0, the flow window size is updated to the total number of acknowledged packets.
2) Otherwise, if (AS > 0), where AS is the packet arrival speed, W is updated as:
   \[ W = \text{ceil}(W * 0.875 + AS * (RTT + ATP) * 0.125) \]
3) Limit W to the maximum flow window size of the peer side.

3.8 Connection Setup and shutdown

One UDT entity starts first as the server, and its peer side (the client) will send a handshake packet when it wishes to connect. The client should keep on sending the handshake packet every constant interval, (implementations should decide this interval according to the balance between response time and system overhead), until it receives a response handshake from the server or timeout occurs.

The handshake packet has the following information:
1) UDT version: this value is for compatibility purpose. The current version is 2.
2) Initial Sequence Number: this is the initial data packet sequence number that the UDT entity that sends this handshake will use to send out data packets. This must be a random value between 1 and \(2^{31} - 1\). In addition, it is recommended that the value should not repeat within a reasonable time history window.

3) MSS: the size of a data packet (measured by IP payload).

4) Maximum Flow Window Size: This is the maximum flow window size the UDT entity that receives this handshake is allowed to have. The window size is generally limited by the sizes of data structures in the receiver.

The server, when receiving a handshake packet, compares the MSS value with its own value and set its own values as the smaller one. The resulting 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 version value is used to check the compatibility of the two peers. The initial sequence number and maximum flow window size are used to initialize the parameters of the UDT entity that receives this handshake packet.

The server is ready for sending/receiving data right after this step is completed. However, it must send back response packets as long as it receives any further handshakes from the same client.

The client can start sending/receiving data once it gets a response handshake packet from the server, sets its own MSS as the value carried in the handshake packet, and initialize its parameters related to initial sequence number and maximum flow window size. Further response handshake messages, if received any, 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 according to the timeout mechanism (see Section 3.5).

3.9 Loss Information Compression Scheme

The loss information carried in an NAK packet is an array of 32-bit integers. If a number 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.
For example, the following information carried in an NAK:
  0x00000002, 0x80000006, 0x0000000B, 0x0000000E
means packets with sequence number 2, 6, 7, 8, 9, 10, 11, and 14 are lost.

4. Efficiency and Fairness Characters

UDT can utilize close to the full available bandwidth independent of link capacity, RTT, and background coexisting flows, given the link bit error rate (BER) of the current wired networks. There is a constant time needed for UDT to increase from 0bits/s to 90 percent of the available bandwidth when there is no loss, which is 7.5 seconds. 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 their fair share according to max-min rule. RTT has little impact 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 techniques do not work optimally. Simulation shows that this threshold link capacity is about 100 kb/s.
3) In networks where FIFO queues are used along the paths, if the queue size is greater than BDP, TCP’s bandwidth share decreases as the queue size increases. However, to reach about equal sharing with UDT, the queue size needed generally exceeds the amount that a practical router/switch will provide.

When short (timewise) web-like TCP flows coexist with a small number of concurrent UDT flows, the effect of UDT on TCP flows is very small.

More analysis can be found in [GHG03].

Security Considerations

UDT has not its specific security mechanism, whereas it depends on the application to provide authentication and the lower layer to provide security mechanisms, if necessary.
However, UDT implementations should check that all arrived packets are from the expected source, since UDP is connectionless. This is inherently accomplished because in the socket API a "connected" UDP socket only accepts packets from a certain source the UDP socket is "connected" to.

Normative References


Informative References


Acknowledgments

The authors thank Dr. Xinwei Hong for his cooperated work in the SABUL protocol, an early prototype of UDT protocol. 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 infrastructures.

We are also grateful for the feedback and comments from the following individuals: Guy Almes, Cees de Laat, Freek Dijkstra, Rajkumar Kettimuthu, and Johan Blom.
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