Internet Engineering Task Force
Internet Draft
Intended status: Informational
Expires: October 12, 2010

Yunhong Gu
University of Illinois at Chicago

April 12, 2010

UDT: UDP-based Data Transfer Protocol
draft-gg-udt-03.txt

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

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.

This Internet-Draft will expire on October 15, 2010.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Abstract

This document describes UDT, or the UDP based Data Transfer protocol. UDT is designed to be an alternative data transfer protocol for the situations when TCP does not work well. One of the most common cases, and also the original motivation of UDT, is to overcome TCP's

inefficiency in high bandwidth-delay product (BDP) networks. Another important target use scenario is to allow networking researchers, students, and application developers to easily implement and deploy new data transfer algorithms and protocols. Furthermore, UDT can also be used to better support firewall traversing.

UDT is completely built on top of UDP. However, UDT is connection oriented, unicast, and duplex. It supports both reliable data streaming and partial reliable messaging. The congestion control module is an open framework that can be used to implement and/or deploy different control algorithms. UDT also has a native/default control algorithm based on AIMD rate control.

Table of Contents

1. Introduction....4
2. Packet Structures....5
3. UDP Multiplexer....8
4. Timers....8
5. Connection Setup and shutdown....9
5.1 Client/Server Connection Setup....10
5.2 Rendezvous Connection Setup....10
5.3 Shutdown....11
6. Data Sending and Receiving....11
6.1 The Sender's Algorithm....11
6.2 The Receiver's Algorithm....12
6.3 Flow Control....15
6.4 Loss Information Compression Scheme....15
7. Configurable Congestion Control (CCC)....15
7.1 CCC Interface....15
7.2 UDT's Native Control Algorithm....16
Security Considerations....18
Normative References....18
Informative References....18
Author's Addresses....19

1. Introduction

The Transmission Control Protocol (TCP) [RFC5681] has been very successful and greatly contributes to the popularity of today's Internet. Today TCP still contributes the majority of the traffic on the Internet.

However, TCP is not perfect and it is not designed for every specific applications. In the last several years, with the rapid advance of optical networks and rich Internet applications, TCP has been found inefficient as the network bandwidth-delay product (BDP) increases. 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].

To overcome the TCP's inefficiency problem over high speed wide area networks is the original motivation of UDT. Although there are new TCP variants deployed today (for example, BiC TCP [XHR04] on Linux and Compound TCP [TS06] on Windows), certain problems still exist. For example, none of the new TCP variants address RTT unfairness, the situation that connections with shorter RTT consume more bandwidth.

Moreover, as the Internet continues to evolve, new challenges and requirements to the transport protocol will always emerge. Researchers need a platform to rapidly develop and test new algorithms and protocols. Network researchers and students can use UDT to easily implement their ideas on transport protocols, in particular congestion control algorithms, and conduct experiments over real networks.

Finally, there are other situations when UDT can be found more helpful than TCP. For example, UDP-based protocol is usually easier for punching NAT firewalls. For another example, TCP's congestion control and reliability control is not desirable in certain applications of VOIP, wireless communication, etc. Application developers can use (with or without modification) UDT to suit their requirements.

Due to all those reasons and motivations described above, we believe that it is necessary to design a well defined and developed UDP-based data transfer protocol.

As its name suggest, UDT is built solely on the top of UDP [RFC768]. Both data and control packets are transferred using UDP. UDT is connection-oriented in order to easily maintain congestion control,

reliability, and security. It is a unicast protocol while multicast is not considered here. Finally, data can be transferred over UDT in

duplex.

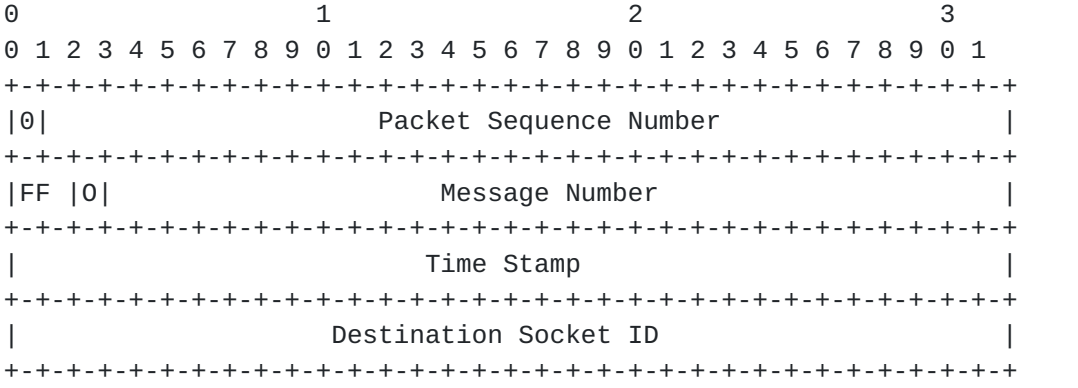
UDT supports both reliable data streaming and partial reliable messaging. The data streaming semantics is similar to that of TCP, while the messaging semantics can be regarded as a subset of SCTP [RFC4960].

This document defines UDT's protocol specification. The detailed description and performance analysis can be found in [GG07], and a fully functional reference implementation can be found at [UDT].

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.



The data packet header starts with 0. Packet sequence number uses the following 31 bits after the flag bit. UDT uses packet based sequencing, i.e., the sequence number is increased by 1 for each sent data packet in the order of packet sending. Sequence number is wrapped after it is increased to the maximum number (2^31 - 1).

The next 32-bit field in the header is for the messaging. The first two bits "FF" flags the position of the packet is a message. "10" is the first packet, "01" is the last one, "11" is the only packet, and "00" is any packets in the middle. The third bit "O" means if the message should be delivered in order (1) or not (0). A message to be delivered in order requires that all previous messages must be either delivered or dropped. The rest 29 bits is the message number, similar to packet sequence number (but independent). A UDT message may contain multiple UDT packets.

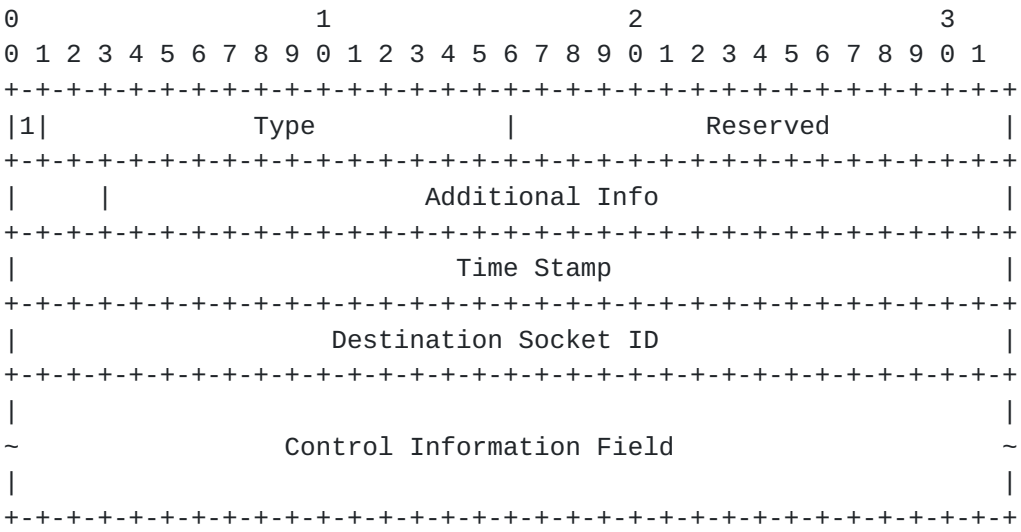
Following are the 32-bit time stamp when the packet is sent and the

destination socket ID. The time stamp is a relative value starting

from the time when the connection is set up. The time stamp information is not required by UDT or its native control algorithm. It is included only in case that a user defined control algorithm may require the information (See Section 6).

The Destination ID is used for UDP multiplexer. Multiple UDT socket can be bound on the same UDP port and this UDT socket ID is used to differentiate the UDT connections.

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



There are 8 types of control packets in UDT and the type information is put in bit field 1 - 15 of the header. The contents of the following fields depend on the packet type. The first 128 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 32 - 63 ("Additional Info") in the control packet header. The ACK sequence number ranges from 0 to (2^31 - 1).

TYPE 0x0: Protocol Connection Handshake
Additional Info: Undefined
Control Info:
1) 32 bits: UDT version
2) 32 bits: Socket Type (STREAM or DGRAM)
3) 32 bits: initial packet sequence number
4) 32 bits: maximum packet size (including UDP/IP headers)
5) 32 bits: maximum flow window size

6) 32 bits: connection type (regular or rendezvous)

7) 32 bits: socket ID
8) 32 bits: SYN cookie
9) 128 bits: the IP address of the peer's UDP socket

TYPE 0x1: Keep-alive
Additional Info: Undefined
Control Info: None

TYPE 0x2: Acknowledgement (ACK)
Additional Info: ACK sequence number
Control Info:
1) 32 bits: The packet sequence number to which all the previous packets have been received (excluding)
[The following fields are optional]
2) 32 bits: RTT (in microseconds)
3) 32 bits: RTT variance
4) 32 bits: Available buffer size (in bytes)
5) 32 bits: Packets receiving rate (in number of packets per second)
6) 32 bits: Estimated link capacity (in number of packets per second)

TYPE 0x3: Negative Acknowledgement (NAK)
Additional Info: Undefined
Control Info:
1) 32 bits integer array of compressed loss information (see section 3.9).

TYPE 0x4: Unused

TYPE 0x5: Shutdown
Additional Info: Undefined
Control Info: None

TYPE 0x6: Acknowledgement of Acknowledgement (ACK2)
Additional Info: ACK sequence number
Control Info: None

TYPE 0x7: Message Drop Request:
Additional Info: Message ID
Control Info:
1) 32 bits: First sequence number in the message
2) 32 bits: Last sequence number in the message

TYPE 0x7FFF: Explained by bits 16 - 31, reserved for user defined Control Packet

Finally, Time Stamp and Destination Socket ID also exist in the control packets.

3. UDP Multiplexer

A UDP multiplexer is used to handle concurrent UDT connections sharing the same UDP port. The multiplexer dispatch incoming UDT packets to the corresponding UDT sockets according to the destination socket ID in the packet header.

One multiplexer is used for all UDT connections bound to the same UDP port. That is, UDT sockets on different UDP port will be handled by different multiplexers.

A multiplexer maintains two queues. The sending queue includes the sockets with at least one packet scheduled for sending. The UDT sockets in the sending queue are ordered by the next packet sending time. A high performance timer is maintained by the sending queue and when it is time for the first socket in the queue to send its packet, the packet will be sent and the socket will be removed. If there are more packets for that socket to be sent, the socket will be re-inserted to the queue.

The receiving queue reads incoming packets and dispatches them to the corresponding sockets. If the destination ID is 0, the packet will be sent to the listening socket (if there is any), or to a socket that is in rendezvous connection phase. (See Section 5.)

Similar to the sending queue, the receiving queue also maintains a list of sockets waiting for incoming packets. The receiving queue scans the list to check if any timer expires for each socket every SYN (SYN = 0.01 second, defined in Section 4).

4. Timers

UDT uses four timers to trigger different periodical events. Each event has its own period and they are all independent. They use the system time as origins and should process wrapping if the system time wraps.

For a certain periodical event E in UDT, suppose the time variable is ET and its period is p. If E is set or reset at system time t0 (ET = t0), then at any time t1, (t1 - ET >= p) is the condition to check if E should be triggered.

The four timers are ACK, NAK, EXP and SND. SND is used in the sender only for rate-based packet sending (see Section 6.1), whereas the other three are used in the receiver only.

ACK is used to trigger an acknowledgement (ACK). Its period is set by the congestion control module. However, UDT will send an ACK no

longer than every 0.01 second, even though the congestion control does not need timer-based ACK. Here, 0.01 second is defined as the SYN time, or synchronization time, and it affects many of the other timers used in UDT.

NAK is used to trigger a negative acknowledgement (NAK). Its period is dynamically updated to 4 * RTT_+ RTTVar + SYN, where RTTVar is the variance of RTT samples.

EXP is used to trigger data packets retransmission and maintain connection status. Its period is dynamically updated to N * (4 * RTT + RTTVar + SYN), where N is the number of continuous timeouts. To avoid unnecessary timeout, a minimum threshold (e.g., 0.5 second) should be used in the implementation.

The recommended granularity of their periods is microseconds. However, accurate time keeping is not necessary, except for SND.

In the rest of this document, a name of a time variable will be used to represent the associated event, the variable itself, or the value of its period, depending on the context. For example, ACK can mean either the ACK event or the value of ACK period.

5. Connection Setup and shutdown

UDT supports two different connection setup methods, the traditional client/server mode and the rendezvous mode. In the latter mode, both UDT sockets connect to each other at (approximately) the same time.

The UDT client (in rendezvous mode, both peer are clients) sends a handshake request (type 0 control packet) to the server or the peer side. The handshake packet has the following information (suppose UDT socket A sends this handshake to B):

1) UDT version: this value is for compatibility purpose. The current version is 4.
2) Socket Type: STREAM (0) or DGRAM (1).
3) Initial Sequence Number: It is the sequence number for the first data packet that A will send out. This should be a random value.
4) Packet Size: the maximum size of a data packet (including all headers). This is usually the value of MTU.
5) Maximum Flow Window Size: This value may not be necessary; however, it is needed in the current reference implementation.
6) Connection Type. This information is used to differential the connection setup modes and request/response.
7) Socket ID. The client UDT socket ID.
8) Cookie. This is a cookie value used to avoid SYN flooding attack [RFC4987].

9) Peer IP address: B's IP address.

5.1 Client/Server Connection Setup

One UDT entity starts first as the server (listener). The server accepts and processes incoming connection request, and creates new UDT socket for each new connection.

A client that wants to connect to the server will send a handshake packet first. The client should keep on sending the handshake packet every constant interval until it receives a response handshake from the server or a timeout timer expires.

When the server first receives the connection request from a client, it generates a cookie value according to the client address and a secret key and sends it back to the client. The client must then send back the same cookie to the server.

The server, when receiving a handshake packet and the correct cookie, compares the packet size and maximum window size with its own values and set its own values as the smaller ones. The result values are also sent back to the client by a response handshake packet, together with the server's version and initial sequence number. 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 further handshakes from the same client.

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

The connection type from the client should be set to 1 and the response from the server should be set to -1.

The client should also check if the response is from the server that the original request was sent to.

5.2 Rendezvous Connection Setup

In this mode, both clients send a connect request to each other at the same time. The initial connection type is set to 0. Once a peer receives a connection request, it sends back a response. If the connection type is 0, then the response sends back -1; if the connection type is -1, then the response sends back -2; No response will be sent for -2 request.

The rendezvous peer does the same check on the handshake messages (version, packet size, window size, etc.) as described in Section 5.1. In addition, the peer only process the connection request from

the address it has sent a connection request to. Finally, rendezvous connection should be rejected by a regular UDT server (listener).

A peer initializes the connection when it receives -1 response.

The rendezvous connection setup is useful when both peers are behind firewalls. It can also provide better security and usability when a listening server is not desirable.

5.3 Shutdown

If one of the connected UDT entities is being closed, it will send a shutdown message to the peer side. The peer side, after received 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 after 16 continuous EXP timeout (see section 3.5). However, the total timeout value should be between a minimum threshold and a maximum threshold. In our reference implementation, we use 3 seconds and 30 seconds, respectively.

6. Data Sending and Receiving

Each UDT entity has two logical parts: the sender and the receiver. The sender sends (and retransmits) application data according to the flow control and congestion control. The receiver receives both data packets and control packets, and sends out control packets according to the received packets and the timers.

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

UDT always tries to pack application data into fixed size packets (the maximum packet size negotiated during connection setup), unless there is not enough data to be sent.

We explained the rationale of some of the UDT data sending/receiving schemes in [GHG04b].

6.1 The Sender's Algorithm

Data Structures and Variables:

1) Sender's Loss List: The sender's loss list is used to store the sequence numbers of the lost packets fed back by the receiver through NAK packets or inserted in a timeout event. 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) In messaging mode, if the packets has been the loss list for a time more than the application specified TTL (time-to-live), send a message drop request and remove all related packets from the loss list. Go to 1).
3) Wait until there is application data to be sent.
4)
   a. If the number of unacknowledged packets exceeds the flow/congestion window size, wait until an ACK comes. Go to 1).
   b. Pack a new data packet and send it out.
5) If the sequence number of the current packet is 16n, where n is an integer, go to 2).
6) Wait (SND - t) time, where SND is the inter-packet interval updated by congestion control and t is the total time used by step 1 to step 5. Go to 1).

6.2
The Receiver's Algorithm

Data Structures and Variables:

1) Receiver's Loss List: It is a list of tuples whose values include: the sequence numbers of detected lost data packets, the latest feedback time of each tuple, and a parameter k that is the number of times each one has been fed back in NAK. 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. The most recent value will overwrite the oldest one if no more free space in the array.
3) 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 minus 1.
6) ExpCount: A variable to record number of continuous EXP time-out events.

Data Receiving Algorithm:
1) Query the system time to check if ACK, NAK, or EXP timer has expired. If there is any, process the event (as described below in this section) and reset the associated time variables. For ACK, also check the ACK packet interval.
2) Start time bounded UDP receiving. If no packet arrives, go to 1).
1) Reset the ExpCount to 1. If there is no unacknowledged data packet, or if this is an ACK or NAK control packet, reset the EXP

timer.

3) Check the flag bit of the packet header. If it is a control

packet, process it according to its type and go to 1).
4) 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 the Packet Pair Window.
5) Record the packet arrival time in PKT History Window.
6)
   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.
7) Update LRSN. Go to 1).

ACK Event Processing:
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 LRSN + 1; otherwise it is the smallest sequence number in the receiver's 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. Pack the ACK packet with RTT, RTT Variance, and flow window size (available receiver buffer size). If this ACK is not triggered by ACK timers, send out this ACK and stop.
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*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 the estimated link capacity according to the following algorithm:
   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).
6) Pack the 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 the ACK History Window.

NAK Event Processing:
Search the receiver's loss list, find out all those sequence numbers

whose last feedback time is k*RTT before, where k is initialized as 2

and increased by 1 each time the number is fed back. Compress (according to section 6.4) and send these numbers back to the sender in an NAK packet.

EXP Event Processing:
1) Put all the unacknowledged packets into the sender's loss list.
2) If (ExpCount > 16) and at least 3 seconds has elapsed since that last time when ExpCount is reset to 1, or, 3 minutes has elapsed, close the UDT connection and exit.
3) If the sender's loss list is empty, send a keep-alive packet to the peer side.
4) Increase ExpCount by 1.

On ACK packet received:
1) Update the largest acknowledged sequence number.
2) Send back an ACK2 with the same ACK sequence number in this ACK.
3) Update RTT and RTTVar.
4) Update both ACK and NAK period to 4 * RTT + RTTVar + SYN.
5) Update flow window size.
6) If this is a Light ACK, stop.
7) Update packet arrival rate: A = (A * 7 + a) / 8, where a is the value carried in the ACK.
8) Update estimated link capacity: B = (B * 7 + b) / 8, where b is the value carried in the ACK.
9) Update sender's buffer (by releasing the buffer that has been acknowledged).
10) Update sender's loss list (by removing all those that has been acknowledged).

On NAK packet received:
1) Add all sequence numbers carried in the NAK into the sender's loss list.
2) Update the SND period by rate control (see section 3.6).
3) Reset the EXP time variable.

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 value as: RTT = (RTT * 7 + rtt) / 8.
4) Update RTTVar by: RTTVar = (RTTVar * 3 + abs(RTT - rtt)) / 4.
5) Update both ACK and NAK period to 4 * RTT + RTTVar + SYN.

On message drop request received:
1) Tag all packets belong to the message in the receiver buffer so that they will not be read.

2) Remove all corresponding packets in the receiver's loss list.

On Keep-alive packet received:
Do nothing.

On Handshake/Shutdown packet received:
See Section 5.

6.3 Flow Control

The flow control window size is 16 initially.

On ACK packet received:
The flow window size is updated to the receiver's available buffer size.

6.4 Loss Information 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.

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.

7. Configurable Congestion Control (CCC)

The congestion control in UDT is an open framework so that user-defined control algorithm can be easily implemented and switched. Particularly, the native control algorithm is also implemented by this framework.

The user-defined algorithm may redefine several control routines to read and adjust several UDT parameters. The routines will be called when certain event occurs. For example, when an ACK is received, the control algorithm may increase the congestion window size.

7.1 CCC Interface

UDT allow users to access two congestion control parameters: the congestion window size and the inter-packet sending interval. Users

may adjust these two parameters to realize window-based control, rate-based control, or a hybrid approach.

In addition, the following parameters should also be exposed.

1) RTT
2) Maximum Segment/Packet Size
3) Estimated Bandwidth
4) The latest packet sequence number that has been sent so far
5) Packet arriving rate at the receiver side

A UDT implementation may expose additional parameters as well. This information can be used in user-defined congestion control algorithms to adjust the packet sending rate.

The following control events can be redefined via CCC (e.g., by a callback function).

1) init: when the UDT socket is connected.
2) close: when the UDT socket is closed.
3) onACK: when ACK is received.
4) onLOSS: when NACK is received.
5) onTimeout: when timeout occurs.
6) onPktSent: when a data packet is sent.
7) onPktRecv: when a data packet is received.

Users can also adjust the following parameters in the user-defined control algorithms.

1) ACK interval: An ACK may be sent every fixed number of packets. User may define this interval. If this value is -1, then it means no ACK will be sent based on packet interval.
2) ACK Timer: An ACK will also be sent every fixed time interval. This is mandatory in UDT. The maximum and default ACK time interval is SYN.
3) RTO: UDT uses 4 * RTT + RTTVar to compute RTO. Users may redefine this.

Detailed description and discussion of UDT/CCC can be found in [GG05].

7.2 UDT's Native Control Algorithm

UDT has a native and default control algorithm, which will be used if no user-defined algorithm is implemented and configured. The native UDT algorithm should be implemented using CCC.

UDT's native algorithm is a hybrid congestion control algorithm, hence it adjusts both the congestion window size and the inter-packet interval. The native algorithm uses timer-based ACK and the ACK interval is SYN.

The initial congestion window size is 16 packets and the initial

inter-packet interval is 0. The algorithm start with Slow Start phase

until the first ACK or NAK arrives.

On ACK packet received:
1) If the current status is in the slow start phase, set the congestion window size to the product of packet arrival rate and (RTT + SYN). Slow Start ends. Stop.
2) Set the congestion window size (CWND) to: CWND = A * (RTT + SYN) + 16.
3) The number of sent packets to be increased in the next SYN period (inc) is calculated as:

```
if (B <= C)
   inc = 1/PS;
else
   inc = max(10^(ceil(log10((B-C)*PS*8))) * Beta/PS, 1/PS);
```

   where B is the estimated link capacity and C is the current sending speed. All are counted as packets per second. PS is the fixed size of UDT packet counted in bytes. Beta is a constant value of 0.0000015.
4) The SND period is updated as:

```
SND = (SND * SYN) / (SND * inc + SYN).
```

These four parameters are used in rate decrease, and their initial values are in the parentheses: AvgNAKNum (1), NAKCount (1), DecCount(1), LastDecSeq (initial sequence number - 1).

We define a congestion period as the period between two NAKs in which the first biggest lost packet sequence number is greater than the LastDecSeq, which is the biggest sequence number when last time the packet sending rate is decreased.

AvgNAKNum is the average number of NAKs in a congestion period. NAKCount is the current number of NAKs in the current period.

On NAK packet received:
1) If it is in slow start phase, set inter-packet interval to 1/recvrate. Slow start ends. Stop.
2) If this NAK starts a new congestion period, increase inter-packet interval (snd) to snd = snd * 1.125; Update AvgNAKNum, reset NAKCount to 1, and compute DecRandom to a random (average distribution) number between 1 and AvgNAKNum. Update LastDecSeq. Stop.
3) If DecCount <= 5, and NAKCount == DecCount * DecRandom:
   a. Update SND period: SND = SND * 1.125;
   b. Increase DecCount by 1;
   c. Record the current largest sent sequence number (LastDecSeq).

The native UDT control algorithm is designed for bulk data transfer over high BDP networks. [GHG04a]

Security Considerations

UDT's security mechanism is similar to that of TCP. Most of TCP's approach to counter security attack should also be implemented in UDT.

IANA Considerations

This document has no actions for IANA.

Normative References

[RFC768] J. Postel, User Datagram Protocol, Aug. 1980.

Informative References

[RFC4987] W. Eddy, TCP SYN Flooding Attacks and Common Mitigations.

[GG07] Yunhong Gu and Robert L. Grossman, UDT: UDP-based Data Transfer for High-Speed Wide Area Networks, Computer Networks (Elsevier). Volume 51, Issue 7. May 2007.

[GG05] Yunhong Gu and Robert L. Grossman, Supporting Configurable Congestion Control in Data Transport Services, SC 2005, Nov 12 - 18, Seattle, WA, USA.

[GHG04b] Yunhong Gu, Xinwei Hong, and Robert L. Grossman, Experiences in Design and Implementation of a High Performance Transport Protocol, SC 2004, Nov 6 - 12, Pittsburgh, PA, USA.

[GHG04a] Yunhong Gu, Xinwei Hong, and Robert L. Grossman, An Analysis of AIMD Algorithms with Decreasing Increases, First Workshop on Networks for Grid Applications (Gridnets 2004), Oct. 29, San Jose, CA, USA.

[LM97] T. V. Lakshman and U. Madhow, The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss, IEEE/ACM Trans. on Networking, vol. 5 no 3, July 1997, pp. 336-350.

[RFC5681] Allman, M., Paxson, V. and E. Blanton, TCP Congestion Control, September 2009.

[RFC4960] R. Stewart, Ed. Stream Control Transmission Protocol. September 2007.

[TS06] K. Tan, Jingmin Song, Qian Zhang, Murari Sridharan, A Compound TCP Approach for High-speed and Long Distance Networks, in IEEE Infocom, April 2006, Barcelona, Spain.

[UDT] UDT: UDP-based Data Transfer, URL http://udt.sf.net.

[XHR04] Lisong Xu, Khaled Harfoush, and Injong Rhee, Binary Increase Congestion Control for Fast Long-Distance Networks, INFOCOM 2004.

Author's Addresses

Yunhong Gu
National Center for Data Mining
University of Illinois at Chicago
713 SEO, M/C 249, 851 S Morgan St
Chicago, IL 60607, USA
Phone: +1 (312) 413-9576
Email: yunhong@lac.uic.edu