BEP:	29
Title:	uTorrent transport protocol
Version:	023256c7581a4bed356e47caf8632be2834211bd
Last-Modified:	Thu Jan 12 12:29:12 2017 -0800
Author:	Arvid Norberg <arvid@bittorrent.com>
Status:	Accepted
Type:	Standards Track
Content-Type:	text/x-rst
Created:	22-Jun-2009
Post-History:	20-Oct-2012 (arvid@bittorrent.com), update the loss factor from 0.78 to 0.5. Removed 'bitfield' extension header. Fixed typo in uTP header (type and version were swapped)

uTorrent Transport Protocol

credits

The uTorrent transport protocol was designed by Ludvig Strigeus, Greg Hazel, Stanislav Shalunov, Arvid Norberg and Bram Cohen.

rationale

The motivation for uTP is for BitTorrent clients to not disrupt internet connections, while still utilizing the unused bandwidth fully.

The problem is that DSL and cable modems typically have a send buffer disproportional to their max send rate, which can hold several seconds worth of packets. BitTorrent traffic is typically background transfers, and should have lower priority than checking email, phone calls and browsing the web, but when using regular TCP connections BitTorrent quickly fills up the send buffer, adding multiple seconds delay to all interactive traffic.

The fact that BitTorrent uses multiple TCP connections gives it an unfair advantage when competing with other services for bandwidth, which exaggerates the effect of BitTorrent filling the upload pipe. The reason for this is because TCP distributes the available bandwidth evenly across connections, and the more connections one application uses, the larger share of the bandwidth it gets.

The traditional solution to this problem is to cap the upload rate of the BitTorrent client to 80% of the up-link capacity. 80% leaves some head room for interactive traffic.

The main drawbacks with this solution are:

1. The user needs to configure his/her BitTorrent client, it won't work out-of-the-box.
2. The user needs to know his/her internet connection's upload capacity. This capacity may change, especially on laptops that may connect to a large number of different networks.
3. The headroom of 20% is arbitrary and wastes bandwidth. Whenever there is no interactive traffic competing with BitTorrent, the extra 20% are wasted. Whenever there is competing interactive traffic, it cannot use more than 20% of the capacity.

uTP solves this problem by using the modem queue size as a controller for its send rate. When the queue grows too large, it throttles back.

This lets it utilize the full upload capacity when there is no competition for it, and it lets it throttle back to virtually nothing when there is a lot of interactive traffic.

overview

This document assumes some knowledge of how TCP and window based congestion control works.

uTP is a transport protocol layered on top of UDP. As such, it must (and has the ability to) implement its own congestion control.

The main difference compared to TCP is the delay based congestion control. See the congestion control section.

Like TCP, uTP uses window based congestion control. Each socket has a `max_window` which determines the maximum number of bytes the socket may have *in-flight* at any given time. Any packet that has been sent, but not yet acked, is considered to be in-flight.

The number of bytes in-flight is `cur_window`.

A socket may only send a packet if `cur_window` + `packet_size` is less than or equal to min(`max_window`, `wnd_size`). The packet size may vary, see the packet sizes section.

`wnd_size` is the advertised window from the other end. It sets an upper limit on the number of packets in-flight.

An implementation MAY violate the above rule if the `max_window` is smaller than the packet size, and it paces the packets so that the average `cur_window` is less than or equal to `max_window`.

Each socket keeps a state for the last delay measurement from the other endpoint (`reply_micro`). Whenever a packet is received, this state is updated by subtracting `timestamp_microseconds` from the hosts current time, in microseconds (see header format).

Every time a packet is sent, the sockets `reply_micro` value is put in the `timestamp_difference_microseconds` field of the packet header.

Unlike TCP, sequence numbers and ACKs in uTP refers to packets, not bytes. This means uTP cannot *repackage* data when resending it.

Each socket keeps a state of the next sequence number to use when sending a packet, `seq_nr`. It also keeps a state of the sequence number that was last received, `ack_nr`. The oldest unacked packet is `seq_nr` - `cur_window`.

header format

version 1 header:

```
0       4       8               16              24              32
+-------+-------+---------------+---------------+---------------+
| type  | ver   | extension     | connection_id                 |
+-------+-------+---------------+---------------+---------------+
| timestamp_microseconds                                        |
+---------------+---------------+---------------+---------------+
| timestamp_difference_microseconds                             |
+---------------+---------------+---------------+---------------+
| wnd_size                                                      |
+---------------+---------------+---------------+---------------+
| seq_nr                        | ack_nr                        |
+---------------+---------------+---------------+---------------+
```

All fields are in network byte order (big endian).

version

This is the protocol version. The current version is 1.

connection_id

This is a random, unique, number identifying all the packets that belong to the same connection. Each socket has one connection ID for sending packets and a different connection ID for receiving packets. The endpoint initiating the connection decides which ID to use, and the return path has the same ID + 1.

timestamp_microseconds

This is the 'microseconds' parts of the timestamp of when this packet was sent. This is set using gettimeofday() on posix and QueryPerformanceTimer() on windows. The higher resolution this timestamp has, the better. The closer to the actual transmit time it is set, the better.

timestamp_difference_microseconds

This is the difference between the local time and the timestamp in the last received packet, at the time the last packet was received. This is the latest one-way delay measurement of the link from the remote peer to the local machine.

When a socket is newly opened and doesn't have any delay samples yet, this must be set to 0.

wnd_size

Advertised receive window. This is 32 bits wide and specified in bytes.

The window size is the number of bytes currently in-flight, i.e. sent but not acked. The advertised receive window lets the other end cap the window size if it cannot receive any faster, if its receive buffer is filling up.

When sending packets, this should be set to the number of bytes left in the socket's receive buffer.

extension

The type of the first extension in a linked list of extension headers. 0 means no extension.

There is currently one extension:

1. Selective acks

Extensions are linked, just like TCP options. If the extension field is non-zero, immediately following the uTP header are two bytes:

```
0               8               16
+---------------+---------------+
| extension     | len           |
+---------------+---------------+
```

where `extension` specifies the type of the next extension in the linked list, 0 terminates the list. And `len` specifies the number of bytes of this extension. Unknown extensions can be skipped by simply advancing `len bytes`.

SELECTIVE ACK

Selective ACK is an extension that can selectively ACK packets non-sequentially. Its payload is a bitmask of at least 32 bits, in multiples of 32 bits. Each bit represents one packet in the send window. Bits that are outside of the send window are ignored. A set bit specifies that packet has been received, a cleared bit specifies that the packet has not been received. The header looks like this:

```
0               8               16
+---------------+---------------+---------------+---------------+
| extension     | len           | bitmask
+---------------+---------------+---------------+---------------+
                                |
+---------------+---------------+
```

Note that the len field of extensions refer to bytes, which in this extension must be at least 4, and in multiples of 4.

The selective ACK is only sent when at least one sequence number was skipped in the received stream. The first bit in the mask therefore represents ack_nr + 2. ack_nr + 1 is assumed to have been dropped or be missing when this packet was sent. A set bit represents a packet that has been received, a cleared bit represents a packet that has not yet been received.

The bitmask has reverse byte order. The first byte represents packets [ack_nr + 2, ack_nr + 2 + 7] in reverse order. The least significant bit in the byte represents ack_nr + 2, the most significant bit in the byte represents ack_nr + 2 + 7. The next byte in the mask represents [ack_nr + 2 + 8, ack_nr + 2 + 15] in reverse order, and so on. The bitmask is not limited to 32 bits but can be of any size.

Here is the layout of a bitmask representing the first 32 packet acks represented in a selective ACK bitfield:

```
0               8               16
+---------------+---------------+---------------+---------------+
| 9 8 ...   3 2 | 17   ...   10 | 25   ...   18 | 33   ...   26 |
+---------------+---------------+---------------+---------------+
```

The number in the diagram maps the bit in the bitmask to the offset to add to `ack_nr` in order to calculate the sequence number that the bit is ACKing.

type

The type field describes the type of packet.

It can be one of:

ST_DATA = 0
: regular data packet. Socket is in connected state and has data to send. An ST_DATA packet always has a data payload.

ST_FIN = 1
: Finalize the connection. This is the last packet. It closes the connection, similar to TCP FIN flag. This connection will never have a sequence number greater than the sequence number in this packet. The socket records this sequence number as `eof_pkt`. This lets the socket wait for packets that might still be missing and arrive out of order even after receiving the ST_FIN packet.

ST_STATE = 2
: State packet. Used to transmit an ACK with no data. Packets that don't include any payload do not increase the `seq_nr`.

ST_RESET = 3

Terminate connection forcefully. Similar to TCP RST flag. The remote host does not have any state for this connection. It is stale and should be terminated.

ST_SYN = 4

Connect SYN. Similar to TCP SYN flag, this packet initiates a connection. The sequence number is initialized to 1. The connection ID is initialized to a random number. The syn packet is special, all subsequent packets sent on this connection (except for re-sends of the ST_SYN) are sent with the connection ID + 1. The connection ID is what the other end is expected to use in its responses.

When receiving an ST_SYN, the new socket should be initialized with the ID in the packet header. The send ID for the socket should be initialized to the ID + 1. The sequence number for the return channel is initialized to a random number. The other end expects an ST_STATE packet (only an ACK) in response.

seq_nr

This is the sequence number of this packet. As opposed to TCP, uTP sequence numbers are not referring to bytes, but packets. The sequence number tells the other end in which order packets should be served back to the application layer.

ack_nr

This is the sequence number the sender of the packet last received in the other direction.

connection setup

Here is a diagram illustrating the exchanges and states to initiate a connection. The c.* refers to a state in the socket itself, pkt.* refers to a field in the packet header.

```
initiating endpoint                           accepting endpoint

          | c.state = CS_SYN_SENT                         |
          | c.seq_nr = 1                                  |
          | c.conn_id_recv = rand()                       |
          | c.conn_id_send = c.conn_id_recv + 1           |
          |                                               |
          |                                               |
          | ST_SYN                                        |
          |   seq_nr=c.seq_nr++                           |
          |   ack_nr=*                                    |
          |   conn_id=c.rcv_conn_id                       |
          | >-------------------------------------------> |
          |             c.receive_conn_id = pkt.conn_id+1 |
          |             c.send_conn_id = pkt.conn_id      |
          |             c.seq_nr = rand()                 |
          |             c.ack_nr = pkt.seq_nr             |
          |             c.state = CS_SYN_RECV             |
          |                                               |
          |                                               |
          |                                               |
          |                                               |
          |                     ST_STATE                  |
          |                       seq_nr=c.seq_nr++       |
          |                       ack_nr=c.ack_nr         |
          |                       conn_id=c.send_conn_id  |
          | <------------------------------------------<  |
          | c.state = CS_CONNECTED                        |
          | c.ack_nr = pkt.seq_nr                         |
          |                                               |
          |                                               |
          |                                               |
          | ST_DATA                                       |
          |   seq_nr=c.seq_nr++                           |
          |   ack_nr=c.ack_nr                             |
          |   conn_id=c.conn_id_send                     |
          | >----------------------------------------->  |
          |                        c.ack_nr = pkt.seq_nr |
          |                        c.state = CS_CONNECTED |
          |                                               |
          |                                               | connection established
     .. ..|.. .. .. .. .. .. .. .. .. .. .. .. .. .. .. ..|.. ..
          |                                               |
          |                     ST_DATA                   |
          |                       seq_nr=c.seq_nr++       |
          |                       ack_nr=c.ack_nr         |
          |                       conn_id=c.send_conn_id  |
          | <-------------------------------------------< |
          | c.ack_nr = pkt.seq_nr                         |
          |                                               |
          |                                               |
          V                                               V
```

Connections are identified by their `conn_id` header. If the connection ID of a new connection collides with an existing connection, the connection attempt will fails, since the ST_SYN packet will be unexpected in the existing stream, and ignored.

packet loss

If the packet with sequence number (`seq_nr` - `cur_window`) has not been acked (this is the oldest packet in the send buffer, and the next one expected to be acked), but 3 or more packets have been acked past it (through Selective ACK), the packet is assumed to have been lost. Similarly, when receiving 3 duplicate acks, `ack_nr` + 1 is assumed to have been lost (if a packet with that sequence number has been sent).

This is applied to selective acks as well. Each packet that is acked in the selective ack message counts as one duplicate ack, which, if it 3 or more, should trigger a re-send of packets that had at least 3 packets acked after them.

When a packet is lost, the `max_window` is multiplied by 0.5 to mimic TCP.

timeouts

Every packet that is ACKed, either by falling in the range (last_ack_nr, ack_nr] or by explicitly being acked by a Selective ACK message, should be used to update an `rtt` (round trip time) and `rtt_var` (rtt variance) measurement. last_ack_nr here is the last ack_nr received on the socket before the current packet, and ack_nr is the field in the currently received packet.

The `rtt` and `rtt_var` is only updated for packets that were sent only once. This avoids problems with figuring out which packet was acked, the first or the second one.

`rtt` and `rtt_var` are calculated by the following formula, every time a packet is ACKed:

```
delta = rtt - packet_rtt
rtt_var += (abs(delta) - rtt_var) / 4;
rtt += (packet_rtt - rtt) / 8;
```

The default timeout for packets associated with the socket is also updated every time `rtt` and `rtt_var` is updated. It is set to:

```
timeout = max(rtt + rtt_var * 4, 500);
```

Where timeout is specified in milliseconds. i.e. the minimum timeout for a packet is 1/2 second.

Every time a socket sends or receives a packet, it updates its timeout counter. If no packet has arrived within `timeout` number of milliseconds from the last timeout counter reset, the socket triggers a timeout. It will set its `packet_size` and `max_window` to the smallest packet size (150 bytes). This allows it to send one more packet, and this is how the socket gets started again if the window size goes down to zero.

The initial timeout is set to 1000 milliseconds, and later updated according to the formula above. For every packet consecutive subsequent packet that times out, the timeout is doubled.

packet sizes

In order to have as little impact as possible on slow congested links, uTP adjusts its packet size down to as small as 150 bytes per packet. Using packets that small has the benefit of not clogging a slow up-link, with long serialization delay. The cost of using packets that small is that the overhead from the packet headers become significant. At high rates, large packet sizes are used, at slow rates, small packet sizes are used.

congestion control

The overall goal of the uTP congestion control is to use one way buffer delay as the main congestion measurement, as well as packet loss, like TCP. The point is to avoid running with full send buffers whenever data is being sent. This is specifically a problem for DSL/Cable modems, where the send buffer in the modem often has room for multiple seconds worth of data. The ideal buffer utilization for uTP (or any background traffic protocol) is to run at 0 bytes buffer utilization. i.e. any other traffic can at any time send without being obstructed by background traffic clogging up the send buffer. In practice, the uTP target delay is set to 100 ms. Each socket aims to never see more than 100 ms delay on the send link. If it does, it will throttle back.

This effectively makes uTP yield to any TCP traffic.

This is achieved by including a high resolution timestamp in every packet that's sent over uTP, and the receiving end calculates the difference between its own high resolution timer and the timestamp in the packet it received. This difference is then fed back to the original sender of the packet (timestamp_difference_microseconds). This value is not meaningful as an absolute value. The clocks in the machines are most likely not synchronized, especially not down to microsecond resolution, and the time the packet is in transit is also included in the difference of these timestamps. However, the value is useful in comparison to previous values.

Each socket keeps a sliding minimum of the lowest value for the last two minutes. This value is called *base_delay*, and is used as a baseline, the minimum delay between the hosts. When subtracting the base_delay from the timestamp difference in each packet you get a measurement of the current buffering delay on the socket. This measurement is called *our_delay*. It has a lot of noise it it, but is used as the driver to determine whether to increase or decrease the send window (which controls the send rate).

The *CCONTROL_TARGET* is the buffering delay that the uTP accepts on the up-link. Currently the delay target is set to 100 ms. *off_target* is how far the actual measured delay is from the target delay (calculated from CCONTROL_TARGET - our_delay).

The window size in the socket structure specifies the number of bytes we may have in flight (not acked) in total, on the connection. The send rate is directly correlated to this window size. The more bytes in flight, the faster send rate. In the code, the window size is called `max_window`. Its size is controlled, roughly, by the following expression:

```
delay_factor = off_target / CCONTROL_TARGET;
window_factor = outstanding_packet / max_window;
scaled_gain = MAX_CWND_INCREASE_PACKETS_PER_RTT * delay_factor * window_factor;
```

Where the first factor scales the *off_target* to units of target delays.

The scaled_gain is then added to the max_window:

```
max_window += scaled_gain;
```

This will make the window smaller if off_target is greater than 0 and grow the window if off target is less than 0.

If max_window becomes less than 0, it is set to 0. A window size of zero means that the socket may not send any packets. In this state, the socket will trigger a timeout and force the window size to one packet size, and send one packet. See the section on timeouts for more information.