TCP Optimizations

We start with the question on what can be the maximum achievable throughput for a given TCP flow. If we maximise the throughput, we reduce the flow completion time and hence reducing the response time.

A simple back of the envelope calculation would be to divide the receive buffer size (rcvbuf) by the round trip time(rtt) which is the effective throughput of the flow

```
Throughput =< Receiverwindow/RTT implies that  RTT =< Receiverwindow/Throughput
```

The above should tell you that if the RTT is greater than 5 ms the throughput will be limited even with the maximum receiverwindow So for example on a 100ms link with a 32KB receive buffer caps the throughput of that flow at 2.56Mbps regardless of the available capacity. Typically a sender can only use 16 bits of windows size so in effect you can only send 65535 bytes without receiving ACK unless you use window scaling. Alternatively we can also calculate the required amount of bytes in the receiverwindow to utilize the connection to its fullest by multiplying bandwidth with rtt like:
(100 000 000 * 0,0072) / 8 = 90 000 bytes

Assuming arbitrarily large buffers on high speed networks and knowing the classic saw tooth pattern of congestion window oscillations we can easily calculate that the effective throughput of a TCP flow can never exceed

$$\min\left(\frac{W_{max}}{RTT}, \frac{1}{RTT\sqrt{\frac{4p}{3}} + T_0 \min\left(1, 3\sqrt{\frac{3p}{4}}\right) p(1+32p^2)}\right)$$

where p is the packet loss probability and RTT is the round trip time between the endpoints. The average PacketSize is usually the average MSS.The maximum segment size(MSS) of a TCP connection is the maximum amount of data the server may send in a TCP packet. With larger MSS,there is less per-packet overhead of administrative information. However, the average value of congestion window of the sender will decrease, given a larger MSS.This increases the probability that packet losses will lead to timeout, thereby reducing the throughput of the TCP connection.

The following are the assumptions made in the derivation of the formula:

- The flow is long lived to reach an equilibrium which means it applies to large file transfers.
- Most packet losses is handled by a fast retransmit rather than retransmission timer expiry.
- Congestion window is halved upon packet loss
- Neither network nor application is a bottleneck

1 Congestion Backoff

The TCP congestion window variable cwnd is the limiting factor of a TCP flow assuming no other problems. It should be recognized that the important thing is that one reduce cwnd by some multiplicative factor but the actual value used is less significant. There is nothing special about n = 2, aside from the fact that everyone else uses the same value. The value n = 2 was not the result of any analysis or empirical testing.Because it halves cwnd on packet loss, traditional TCP shows large variation in the transfer rate withtime.It is also very reactive to loss,as it is under the assumption that the flow itself caused the congestion event. In our implementation, if the cwnd backoff variable is set to n then the congestion window is reduced to of the old value. When n equals 2, this is the standard TCP behavior. We want a higher value of n, as it gives us greater smoothness and throughput at the price of lower reactivity. Sustained packet loss still drives cwnd down, but the reaction to a single loss by itself is less dramatic. When we are acting on behalf of multiple users and if we are multiplexing traffic for N users over one TCP flow, to be TCP-friendly there's a good argument to take conservative values.
The alternative would be to establish N separate TCP flows, one per user . We settle for a value of 8 as its proven to be more fair and stable than the conventional TCP

2 Congestion Floor

We set minimum value for the TCP congestion window and never fall below that value Even at slow start, and after a timeout, the congestion window will remain at least at this value When set to 30 or more we can ensure that most web objects get sent in a single flight of packets (one RTT) even after . slow start or packet loss. (30 x 1500 = 45K, larger than most objects sent by HTTP for a HTML document.)

3 Delayed Acknowledgements

The factor $\sqrt{2}$ comes in the above formula due to the fact that acknowledgements can be delayed. When a TCP receiver uses delayed acknowledgment, it usually sends one acknowledgment for every two data packets it receives. This strategy reduces the number of packets sent into the network. However, this also slows down the rate of growth of the congestion window of the sender and reduces the sender throughput. Moreover, for HTTP type request/response traffic there is no hope of piggybacking the ACK on the data anyway. In our situation we operate on high bandwidth links where conservation of a few bytes isnt an issue. We can set the appropriate profile sysctl like `quick_ack` to be 1 instead of 0 to enable this feature.

4 SYN/ACKOptimization

TCP starts a connection with a three-way handshaking, sending a SYN packet, waiting for a response(SYNACK packet) and then sending ACK for SYNACK. In case of SYN or SYNACK getting lost TCP stack waits 3 seconds (or longer for further back-offs) before retransmitting them. This delay can significantly degrade a performance, especially for short-lived connections. Since two SYNs or . SYNACKs have a higher probability to arrive at their destination, a reasonable solution to enhance the performance is send each SYN(SYNACK) twice with a short time interval syn_timeout between. This reduces the probability of failures at the cost of a little bit more traffic in the network.

5 RTO Optimization

Because many of the TCP timeouts seem unavoidable (full window loss, lost retransmit), it makes sense to try reducing the time spent waiting for a timeout. While it can visibly improve goodput, this solution should be viewed with caution because it also increases the probability of premature timeouts. So, estimating the right retransmission timeout (RTO) value is important for achieving a timely response to packet losses while avoiding premature timeouts. A premature timeout has two negative effects:

- it leads to a spurious retransmission;
- with every timeout, TCP enters the slow start mode even though no packets are lost. Since there is no congestion, TCP thus would underestimate the link capacity and throughput would suffer.

TCP has a conservative minimum RTO (RTOmin) value to guard against spurious retransmissions. Linux TCP stack uses an RTOmin value of 200ms. Unfortunately, this value may be in times greater than round-trip times for end-user connections (which are typically about 20-50 ms). To fix this situation the following approach may be accepted:

- we reduce RTOmin down to 20 ms
- the current RTO value is estimated as 3 x current smoothed RTT.

Numerous tests with wireless clients had shown that this strategy helps achieving a timely response to packet losses while keeping rather small risks of spurious retransmissions in case of rtt spikes.

6 Reordering Optimization

A packet reordering in Internet is a well-known phenomenon that the order of packets is inverted due to multi-path routing or parallelism at routers and communicating hosts. It can affect the performance for several reasons:

- Causes unnecessary retransmission: When the TCP receiver gets packets out of order, it sends duplicate ACKs to trigger fast retransmit algorithm at the sender. These ACKs makes the TCP sender infer a packet has been lost and retransmit it. If the temporary sequence number gap is caused by reordering, then the duplicate ACKs and the fast retransmission are unnecessary and a waste of bandwidth.
- Limits transmission speed: When fast retransmission is triggered by dupACKs, the TCP sender assumes it is an indication of network congestion. It reduces its congestion window to limit the transmission speed, which needs to grow larger from a "slow start" again. If reordering happens frequently, the congestion window is at a small size and can hardly grow larger. As a result, TCP has to transmit packets at a limited speed and can not efficiently utilize the bandwidth.

Results of measurements demonstrate the high prevalence of packet reordering to packet losses across high speed backbone networks with a degree of reordering up to 90. Investigations of real IP flows show also that the most of reordered packets arrive at the receiver with time lags less then 3-5 ms. If to take into account this fact the following strategy can be accepted as a mean blocking an impact of this phenomenon to the performance:

- If the first dupACK is detected the TCP stack is blocked from any actions on this event for a certain time
- If the actual packet reordering took place this timeout is enough for self-recovery.
- If the packet loss took place a "standard" fast retransmit algorithm starts.

7 Redundant Packets

The only way to avoid a "classic" RTO retransmit and to start ether "slow start" or "fast retransmit" mechanisms in the case of loss of a last sent packet(or a bunch of last sent packets) is to resend it, if we did not receive its ACK for a time of a bit more then a single RTT. Two packets have a higher probability to arrive at their destination.