TCP on Large BDP Networks

Erik Quanstrom
quanstro@coraid.com

ABSTRACT

Plan 9's TCP has had difficulty getting line-speed on large bandwidth-delay product (large BDP) networks. This paper discusses how the tcp implementation has been modified to achieve near line speed with the targeted network parameters of 100Mbps and 70ms latency. It is expected that other fast, but high-latency connections will similarly benefit. The changes made more completely implement NewReno congestion control (RFC 6582) and appropriate byte counting (ABC, RFC 3465).

Overview

Due to the system's needs, Plan 9's tcp has been optimized for either low-latency or low-bandwidth connections. However as networks continue to improve, and as the speed of light remains constant, high-latency, high-bandwidth links are becoming more important as well as more common. By using the usual optimizations such as "appropriate byte counting" (ABC, RFC 3465) and initial window enlargement (RFC 3390) to speed the slow startup phase of TCP, and NewReno (RFC 6582) to speed the congestion control phase near line-rate TCP was realized for the target network with 100Mbps bandwidth and 70ms latency.

ABC Byte Counting and Initial Window

To speed slow start two techniques were used, increasing the initial window (RFC 3390) and adjusting the conditions for increasing the window during slow-start using ABC (RFC 3465). RFC 3390 increases the initial window from one to two MSS-sized segments in typical cases

```
static void
initialwindow(Tcpctl *tcb)
{
	/* RFC 3390 initial window */
	if(tcb->mss < 1095)
		tcb->cwind = 4*tcb->mss;
	else if(tcb->mss < 2190)
		tcb->cwind = 4380;
	else
		tcb->cwind = 2*tcb->mss;
}
```

During slow start, TCP typically counts the number of acks received. Each ack increases the congestion window by one MSS. Instead, ABC byte counting counts the number of bytes acknowledged and increases the congestion window by the number of bytes acked, up to two MSS. This prevents a problem known as “ack division” where the receiver could send multiple acks per packet, artificially inflating the congestion window. With well-behaved clients, a typical delayed ack should acknowledge two MSS worth of data. ABC allows the congestion window to usually open one byte for each byte acknowledged. Additionally, if congestion or loss causes the TCP timer to expire, the “RTO” state is entered. This state expires when slow start completes. But while it persists, the window is opened at most one MSS per ack.

After slow start, ABC increases the congestion window one MSS for every congestion window worth of acknowledged data.

The previous code did implement ABC correctly during congestion avoidance, but did not correctly respond to RTO during slow start, and inverted the test for increasing the congestion window.

The current code is

```
enum {
    L       = 2,        /* aggressive; legal values ∈ [1.0, 2.0] */
};

static void
tcpabcincr(Tcpctl *tcb, uint acked)
{
    uint limit;

    tcb->abcbytes += acked;
    if(tcb->cwind < tcb->ssthresh){
        /* slow start */
        if(tcb->snd.rto)
            limit = 1*tcb->mss;
        else
            limit = L*tcb->mss;
        tcb->cwind += MIN(tcb->abcbytes, limit);
        tcb->abcbytes = 0;
    }
    else{
        tcb->snd.rto = 0;
        /* avoidance */
        if(tcb->abcbytes >= tcb->cwind){
            tcb->abcbytes -= tcb->cwind;
            tcb->cwind += tcb->mss;
        }
    }
}
```

Window Scaling

For bandwidth-delay products above 64KB, window scaling is required. The window scale is an option transmitted with SYN that must be accepted by the receiver. This scales the window by 2^{scale} and obviously increases the minimum window to 2^{scale}. Since it is unlikely to be advantageous to send very small packets, a scale may always be used. The code was always modified to request `Defadvscale` which is currently 4. This gives a maximum window of $2^4 \cdot 65535$ or just under 1MB. Since our BDP is $0.070s \cdot 100\text{Mbps} = 918\text{KB}$, this is enough.

Previous code requested a window-scale that depended on the local link speed. The local link speed is very indirectly related to the maximum bandwidth delay product, and it is not important to keep the window scale small, so this adjustment was removed.

Window Slamming

TCP implementations must not “slam” windows shut. Window slamming is defined as the the receiver decreasing its window without acking any data. This is often is described as moving the right edge of the receive window to the left. Due to the fact that reads of the TCP device do not typically end on packet boundaries, *qio*(9) needs to put he remainder back on the queue. Since the data may be copied to user-space buffers, this needs to be done outside of any locks, thus the apparent queue length may increase unexpectedly when leftover data is put back on the head of the queue. Since the advertised window is the amount of buffer space we are willing to commit minus the amount buffered, previously the TCP implementation could slam the window shut. To prevent this we track the window at the current receive point, and insist that it does not move right. The new connection variable is `rcv.wptr`.

```
static void
tcprcvwin(Conv *s)              /* Call with tcb locked */
{
    int w;
    Tcpctl *tcb;

    tcb = (Tcpctl*)s->ptcl;
    w = tcb->window - qlen(s->rq);
    if(w < 0)
        w = 0;
    /* RFC 1122 § 4.2.2.17 do not move right edge of window left */
    if(seq_lt(tcb->rcv.nxt + w, tcb->rcv.wptr))
        w = tcb->rcv.wptr - tcb->rcv.nxt;
    if(w>>tcb->rcv.scale == 0 || tcb->window > 4*tcb->mss && w < tcb->mss/4)
        tcb->rcv.blocked = 1;
    tcb->rcv.wnd = w;
    tcb->rcv.wptr = tcb->rcv.nxt + w;
}
```

Double Updates

With the original implementation of TCP, updates to the congestion window are made whenever the packet is first received. If the segment is received out-of-order, then it is used to update the congestion window a second time when finally accepted. To prevent updates from being applied twice, a flag was added to the TCP segment structure. If `update` is given a segment with the `update` flag set, then it is ignored. Otherwise the flag is set, and the segment is considered.

NewReno Congestion Control

The essence of TCP NewReno is to try to avoid waiting for the ack timer to expire by looking for a triple-duplicate ack (that is, 4 acks in a row with the same sequence number), which is taken as an indication of packet loss. When this happens, loss recovery mode is entered, until the current `snd.ptr` is acked, or RTO state is entered. This mechanism was already partially implemented. However many of the details have now been filled in and a few errors have been corrected.

Triple-duplicate acks (TDA) are now recognized even when they arrive with data or change the window. This is because it is possible that the reader on the TDA sender can send data even though there has been congestion in the other direction. In addition, the reader can have increased buffering available if its reader reads data during a congestion event.

```
/* newreno fast retransmit */
if(seg->ack == tcb->snd.una)
if(tcb->snd.una != tcb->snd.nxt)
if(++tcb->snd.dupacks == 3){
     ...
}
```

Newly implemented is “window inflation” during recovery. Inflation tries to account for segments that have left the network during loss recovery. If a triple-duplicate ack is reeived during loss recovery, the congestion window is temporarily increased by one MSS. Also when entering recovery mode, `ssthresh` is recalculated then the congestion window is inflated by 3 MSS. On exit the congestion window is deflated to the size of the lack ack, or the `ssthresh`, whichever is smaller.

Also newly implemented is “partial ack” during loss recovery. A partial ack is one that acks some data, but not up to the retransmit point. Partial acks *deflate* the congestion window by the amount acked. The minimum remaining congestion window is 1 MSS.

To prevent undo time spent in loss recovery, only the first partial ack may reset the TCP timer. This is known as the “impatient” variant of NewReno. This was selected because it has better worst-case performance according to the RFC.

Receive Changes

Two small changes were made to receive functionality. If a resequenced packet is accepted, an ack is forced. This requirement is from RFC 5681 §3.2. The rationale is that the sender needs to get the partial ack (or perhaps full recovery from the senders perspective) signal as soon as possible. This condition is detected when we pull a segment from the resequence queue, or accept a packet when there are resequenced segments. The resequence queue was expanded to be large enough to cover the entire advertise window, if only MSS sized packets are sent.

```
if(seg.seq != tcb->rcv.nxt)
if(length != 0 || (seg.flags & (SYN|FIN))) {
    update(s, &seg);
    if(addreseq(f, tcb, tpriv, &seg, bp, length) < 0)
        ...;
    tcb->flags |= FORCE;        /* force dup ack; RFC 5681 §3.2 */
    goto output;
}

if(tcb->nreseq > 0)
    tcb->flags |= FORCE;        /* filled hole in seq; RFC 5681 §3.2 */
```

Additionally, accepted frames are no longer counted. This functionality was moved to sending, and the accepted segment counter was removed.

Send Changes

For sending, we now force an ack if two or more MSS worth of data have been received since the last ack, or if we have opened the window by two or more MSS. The first change shouldn't make any real difference unless we are receiving a large number of tinygrams. But, it does make it more obvious that delayed acks are being properly implemented.

```
/* force ack every 2*mss */
if((tcb->flags & FORCE) == 0)
if(tcb->rcv.nxt - tcb->rcv.ackptr >= 2*tcb->mss){
    tpriv->stats[Delayack]++;
    tcb->flags |= FORCE;
}

/* force ack if window opening */
if((tcb->flags & FORCE) == 0){
    tcprcvwin(s);
    if((int)(tcb->rcv.wnd - tcb->rcv.wsnt) >= 2*tcb->mss){
        tpriv->stats[Wopenack]++;
        tcb->flags |= FORCE;
    }
}
```

With the help of the `retransmit` function, we also do not set the RTT timer during retransmissions.

Retransmission Timeout

During RTO, we no longer apply the congestion backoff for a secondary RTO while RTO is still active. Additionally, we allow TCP timers to be as short as 300ms. In particular, this allows for slightly quicker recovery from a loss storm. This functionality was tested accidentally due to loss storms triggered by a send queue which was too small.

Queue Management

Since the sender may need to retransmit any packet which has not yet been acknowledged, at a minimum we need to be able to retransmit the entire bandwidth-delay product. TCP keeps data available for retransmission in the Queue `Conv.wq`. Formerly, this queue was only QMAX = 65535. Now the limit is multiplied by the window scale.

```
qsetlimit(s->wq, QMAX<<tcb->qscale);
```

The resequence queue also needs to cover the entire tcp window. The largest allowed resequence queue is now

```
qmax = tcb->window / tcb->mss;        /* ~390 for qscale=3 */
```

This gets rather large rather quickly and is the source of some concern.

Abbreviated References

[1] RFC 1122, Requirements for Internet Hosts—Communication Layers

[2] RFC 3390, Increasing TCP's Initial Window

[3] RFC 3465, TCP Congestion Control with Appropriate Byte Counting (ABC)

[4] RFC 5681, TCP Congestion Control

[5] RFC 6582, The NewReno Modification to TCP's Fast Recovery Algorithm