

## The original Van Jacobson memo

To: Craig Partridge <cr...@aland.bbn.com>  
Cc: David Clark <d...@lcs.mit.edu>  
Subject: Re: query about TCP header on tcp-ip  
In-Reply-To: Your message of Tue, 07 Sep 93 09:48:00 PDT.  
Date: Tue, 07 Sep 93 22:30:14 PDT  
From: Van Jacobson <v...@ee.lbl.gov>

Craig,

As you probably remember from the "High Speed TCP" CNRI meeting, my kernel looks nothing at all like any version of BSD. *Mbufs* no longer exist, for example, and ``netipl'` and all the protocol processing that used to be done at `netipl` interrupt level are gone. TCP receive packet processing in the new kernel really is about 30 instructions on a RISC (33 on a sparc but three of those are compiler braindamage). Attached is the C code & the associated sparc assembler.

A brief recap of the architecture: Packets go in *'pbufs'* which are, in general, the property of a particular device. There is exactly one, contiguous, packet per *pbuf* (none of that *mbuf* chain stupidity). On the packet input interrupt, the device driver upcalls through the protocol stack (no more of that queue packet on the `netipl` software interrupt bs). The upcalls percolate up the stack until the packet is completely serviced (e.g., NFS requests that can be satisfied from in-memory data & data structures) or they reach a routine that knows the rest of the processing must be done in some process's context in which case the packet is laid on some appropriate queue and the process is unblocked. In the case of TCP, the upcalls almost always go two levels: IP finds the datagram is for this host & it upcalls a TCP demuxer which hashes the ports + SYN to find a PCB, lays the packet on the tail of the PCB's queue and wakes up any process sleeping on the PCB. The IP processing is about 25 instructions & the demuxer is about 10.

As Dave noted, the two processing paths that need the most tuning are the data packet send & receive (since at most every other packet is acked, there will be at least twice as many data packets as ack packets). In the new system, the receiving process calls `'tcp_usrrecv'` (the protocol specific part of the `'recv'` syscall) or is already blocked there waiting for new data. So the following code is embedded in a loop at the start of `tcp_usrrecv` that spins taking packets off the `pcb` queue until there's no room for the next packet in the user buffer or the queue is empty. The TCP protocol processing is done as we remove packets from the queue & copy their data to user space (and since we're in process context, it's possible to do a checksum-and-copy).

Throughout this code, 'tp' points to the pcb and 'ti' points to the TCP header of the first packet on the queue (the ip header was stripped as part of interrupt level ip processing). The header info (excluding the ports which are implicit in the pcb) are sucked out of the packet into registers [this is to minimize cache thrashing and possibly to take advantage of 64 bit or longer loads]. Then the header checksum is computed (tp->ph\_sum is the precomputed pseudo-header checksum + src & dst ports).

```
int tcp_usrrecv(struct uio* uio, struct socket* so)
{
    struct tcpcb *tp = (struct tcpcb *)so->so_pcb;
    register struct pbuf* pb;

    while ((pb = tp->tp_inq) != 0) {
        register int len = pb->len;
        struct tcphdr *ti = (struct tcphdr *)pb->dat;

        u_long seq = ((u_long*)ti)[1];
        u_long ack = ((u_long*)ti)[2];
        u_long flg = ((u_long*)ti)[3];
        u_long sum = ((u_long*)ti)[4];
        u_long cksum = tp->ph_sum;

        /* NB - ocadd is an inline gcc assembler function */
        cksum = ocadd(ocadd(ocadd(ocadd(cksum, seq), ack), flg), sum);
```

Next is the header prediction check which is probably the most opaque part of the code. tp->pred\_flags contains snd\_wnd (the window we expect in incoming packets) in the bottom 16 bits and 0x4x10 in the top 16 bits. The 'x' is normally 0 but will be set non-zero if header prediction shouldn't be done (e.g., if not in established state, if retransmitting, if hole in seq space, etc.). So, the first term of the 'if' checks four different things simultaneously:

- that the window is what we expect
- that there are no tcp options
- that the packet has ACK set & doesn't have SYN, FIN, RST or URG set
- that the connection is in the right state

and the 2nd term of the if checks that the packet is in sequence:

```
#define FMASK (((0xf000 | TH_SYN|TH_FIN|TH_RST|TH_URG|TH_ACK) << 16)
| 0xffff)

    if ((flg & FMASK) == tp->pred_flags && seq == tp->rcv_nxt) {
```

The next few lines are pretty obvious -- we subtract the header length from the total length and if it's less than zero the packet was malformed, if it's zero we must have a pure ack packet & we do the ack stuff otherwise if the ack field didn't move we have a pure data packet which we copy to the user's buffer, checksumming as we go, then update the pcb state if everything checks:

```
len -= 20;
if (len <= 0) {
    if (len < 0) {
        /* packet malformed */
    } else {
        /* do pure ack things */
    }
} else if (ack == tp->snd_una) {
    cksum = in_uio((u_char*)ti + 20, len, uio, cksum);
    if (cksum != 0) {
        /* packet or user buffer errors */
    }
    seq += len;
    tp->rcv_nxt = seq;
    if ((int)(seq - tp->rcv_acked) >= 0) {
        /* send ack */
    } else {
        /* free pbuf */
    }
}
continue;
}
}
/* header prediction failed -- take long path */
```

That's it. On the normal receive data path we execute 16 lines of C which turn into 33 instructions on a sparc (it would be 30 if I could trick gcc into generating double word loads for the header & my carefully aligned pcb fields). I think you could get it down around 25 on a cray or big-endian alpha since the loads, checksum calc and most of the compares can be done on 64 bit quantities (e.g., you can combine the seq & ack tests into one).

Attached is the sparc assembler for the above receive path. Hope this explains Dave's '30 instruction' assertion. Feel free to forward this to tcp-ip or anyone that might be interested.

- Van