Flow Control in the Linux Network Stack

Michael Smith, Steve Bishop
Computer Laboratory, University of Cambridge, England
`http://www.cl.cam.ac.uk/~pes20/Netsem/`

February 5, 2005

1 Introduction

This document describes in some detail the algorithms used for the tuning of buffers and window advertisements in the Linux kernel[1]. A brief discussion of memory structures and destination caches is also given, with relevance to these mechanisms.

We are not the designers; this has all been determined by experiment and by examination of the sources, and is likely incomplete and/or inaccurate. We would be very grateful for any corrections or comments.

2 Overview of Buffer Memory Allocation

The Linux kernel has a number of different memory allocation mechanisms, tuned for different purposes. The two that concern us are both allocations from kernel memory. The first is the allocation of a contiguous memory block using `kmalloc`. This uses a binary buddy system, and is useful for allocation of relatively small, variable amounts of memory. The second is memory allocation via the SLAB cache. If the amount of memory for an object is constant and frequently required, such a cache may be used to provide a memory pool for these objects. This eliminates the wasteful (up to 50%) nature of the former. SLAB caches are used for such objects as inodes and buffer heads – a cache is created using `kmem_cache_create`, and an object is allocated using `kmem_cache_alloc`.

The data structure used for the send and receive buffers of a socket is essentially a linked list of segments. Each packet received from the network is placed in its own buffer[2] (a `struct sk_buff`, in `include/linux/skbuff.h`). The control block of this buffer is allocated from the `skbuff_head_cache` (a SLAB cache). Of the fields in this structure, the most important are the pointers:

• `head`	Start of the data buffer.
• `data`	Start of the data for the current protocol layer (i.e. excluding headers).
• `tail`	End of the data for the current protocol layer.
• `end`	End of the data buffer.

[1]Specifically, we are using version 2.4.21 of the Linux kernel.
[2]This is done by the function `net/core/skbuff.c::alloc_skb`.

The `head` field points to the region of memory in which the raw packet data resides (allocated by `kmalloc`), immediately after which (at the `end` pointer) is a `struct skb_shared_info`, which among other things contains a pointer to a list of IP fragments.

For its receive and send queues, each socket then holds two pointers to lists of `struct sk_buff`[3]; `sk->receive_queue` and `sk->write_queue`.

3 `sysctl` Variables

The TCP implementation has three `sysctl` variable arrays that bound the amount of memory available for send and receive buffers. `sysctl_tcp_mem` bounds the total amount of memory (number of pages) used by TCP for the entire host, `sysctl_tcp_rmem` the amount (in bytes) for the receive buffer of a single socket, and `sysctl_tcp_wmem` the amount (in bytes) for a socket's send buffer. Each of these is an array of three elements, giving the minimum, memory pressure point, and maximum values for the size of these buffers. If we have fewer than `sysctl_tcp_mem[0]` pages allocated to TCP buffers, we are not under memory pressure. However, once we allocate more than `sysctl_tcp_mem[1]` pages, we remain under pressure (setting the `tcp_memory_pressure` flag), until we drop below `sysctl_tcp_mem[0]` again.

The initial values are set in `net/ipv4/tcp.c::tcp_init`, where $order = \log_2 \lfloor \frac{num\_physpages}{2^x} \rfloor$, with $x = 11$ if we have less than 128k pages, otherwise $x = 9$. The values are initialised to:

$order = 0$	`[0]`	`[1]`	`[2]`
`sysctl_tcp_mem`	768×2^{order}	1024×2^{order}	1536×2^{order}
`sysctl_tcp_rmem`	$PAGE\_SIZE$	43689	87378
`sysctl_tcp_wmem`	4096	16384	65536

$1 \leq order < 3$	`[0]`	`[1]`	`[2]`
`sysctl_tcp_mem`	$1536 \times 2^{order} - 1024$	$1536 \times 2^{order} - 512$	1536×2^{order}
`sysctl_tcp_rmem`	$PAGE\_SIZE$	43689	87378
`sysctl_tcp_wmem`	4096	16384	65536

$order \geq 3$	`[0]`	`[1]`	`[2]`
`sysctl_tcp_mem`	$1536 \times 2^{order} - 1024$	$1536 \times 2^{order} - 512$	1536×2^{order}
`sysctl_tcp_rmem`	4096	87380	174760
`sysctl_tcp_wmem`	4096	16384	131072

For a page size of 4KB, which is fairly usual, the values of *order* correspond to a physical memory of:

order	Physical Memory / MB
0	8
1	16
2	32
3	64

For the machines that we are running on our test network, each has 29340KB of physical memory, with 4KB pages, so this translates to an *order* of 2.

[3]Actually, each is a pointer to a `struct_sk_buff_head`, to which the `struct sk_buff` at the head of the queue is cast.

4 Buffer Tuning

Each socket stores two values to indicate the size of its buffers; `sk->rcvbuf` and `sk->sndbuf`. In addition, the fields `sk->rmem_alloc` and `sk->wmem_alloc` store the number of bytes committed (i.e. actually in use) from each queue. Note that the buffer sizes are maxima, and do not correspond to the amount of memory being allocated; memory is only actually allocated when it is needed to store a segment.

When the socket is first created, the function `net/ipv4/tcp_ipv4.c::tcp_v4_init_sock` initialises the receive and send buffer sizes to `sysctl_tcp_rmem[1]` and `sysctl_tcp_wmem[1]` respectively. The initial values are then tuned once the connection enters the established state.

The exception to this is during system startup, when the function `net/ipv4/inet_init` creates the TCP control socket. This has the special function of sending an `RST` on receiving a packet for a non-existent socket. In this case, the send and receive buffers are initialised to `sysctl_rmem_default` and `sysctl_wmem_default` by `net/core/sock.c::sock_init_data`. These in turn are set from `SK_RMEM_MAX` and `SK_WMEM_MAX`; both of which are defined to be 65535 in `include/linux/skbuff.h`[4].

On becoming established, `net/ipv4/tcp_input.c::tcp_init_buffer_space` calls the functions `tcp_fixup_rcvbuf` and `tcp_fixup_sndbuf`. The former sets the receive buffer size to be large enough to hold at least four MSS sized segments[5], so long as this is less than the maximum size imposed by `sysctl_tcp_rmem[2]`. The latter is similar, setting the send buffer to at least three MSS sized segments, bounded by the maximum send buffer size.

5 Maximum Segment Sizes

Knowledge of the maximum segment size (MSS) for a connection is essential to the operation of the algorithms described here. The Linux kernel, in fact, uses four distinct MSS values for each socket, a summary of which is given below:

- `tp->advmss` – The MSS advertised by the host. This is initialised in the function `net/ipv4/tcp_output.c::tcp_advertise_mss`, from the routing table's destination cache (`dst->advmss`). Given that the cached entry is calculated from the MTU (maximum transfer unit) of the next hop, this will have a value of 1460 over Ethernet.

- `tp->ack.rcv_mss` – A lower-bound estimate of the peer's MSS. This is initiated in `net/ipv4/tcp_input.c::tcp_initialize_rcv_mss`, and updated whenever a segment is received by `tcp_measure_rcv_mss`.

- `tp->mss_cache` – The current effective sending MSS, which is calculated in the function `net/ipv4/tcp_output.c::tcp_sync_mss`. When the socket is created, it is initialised to 536 by `net/ipv4/tcp_ipv4.c::tcp_v4_init_sock`. Note that these are the only functions that alter the value of `tp->mss_cache`.

- `tp->mss_clamp` – An upper-bound value of the MSS of the connection. This is negotiated at `connect()`, such that it is the minimum of the MSS values advertised by the two hosts. We will never see a segment larger than this.

[4]`net/core/sock.c` sets both `sysctls` to 32767 if the system has less than 4096 physical pages. Note that these are set by the sockets layer and are not specific to TCP.

[5]We also reserve some multiple of 128 in the overhead of storing an MSS sized segment, such that the reserved overhead is at least one third of the advertised MSS.

The most interesting of these values, for our purposes, is `tp->ack.rcv_mss`. Its initial value is set to the minimum of `tp->mss_cache` and `tp->advmss`, which results is 536 in most circumstances[6]). If we then see a larger segment, the function `tcp_measure_rcv_mss` increases `tp->ack.rcv_mss` to its length. If the segment seen is smaller, however, we may still update `tp->ack.rcv_mss` to the segment length, if the segment is the same length as the last one seen, and either:

1. The segment length (with transport layer header) $\geq$ `TCP_MIN_RCVMSS + sizeof(struct tcphdr)`.
2. The segment length (with transport layer header) $\geq$ `TCP_MIN_MSS + sizeof(struct tcphdr)`, and it is not the case that any of the `FIN`, `URG`, `SYN`, or `PSH` flags have been set.

Note that it is always the case that `tp->ack.rcv_mss` $\leq$ `tp->advmss`, since the remote machine cannot send any segments larger than our advertised MSS.

6 Window Advertisements

The receive window advertisement is calculated at the time that the host sends a segment. This window update occurs in `net/ipv4/tcp_output.c::tcp_select_window`, such that we advertise the amount of free space in the receive buffer, clamped to `tp->rcv_ssthresh` and rounded to a multiple of the MSS. The important difference between the behaviour of Linux and that of BSD, is that of clamping the window advertisement.

Note that the advertised window is never reduced in size, in accordance with the RFC. Therefore, if large segments are initially received, opening up the window, but subsequent segments are significantly smaller, we rely on the receive queue collapsing to avoid running out of buffer space.

This window clamp is updated whenever data is received by the host. This is controlled by two fields from the TCP option block of the socket (`struct tcp_opt` in `include/net/sock.h`); `tp->window_clamp` is the maximum possible window (initially set to 65535, but reduced in the case of memory shortage [see below]), and `tp->rcv_ssthresh` is the current window clamp (named due to the 'slow start' nature of the window increase).

When a `connect()` is called on a socket (`net/ipv4/tcp_ipv4.c::tcp_v4_connect`), the function `include/net/tcp.h::tcp_select_initial_window` is called to initialise the window clamp. This sets `tp->window_clamp` to 65535, and both `tp->rcv_wnd` and `tp->rcv_ssthresh` to four times the MSS[7]. Once the connection is established, `tcp_init_buffer_space` [see above] ensures that `tp->window_clamp` is at most $\frac{3}{4}$ the size of the receive buffer[8] minus the size of the application buffer[9]. In addition, one segment (of advertised MSS size) is reserved from `tp->window_clamp`.

The algorithm that Linux uses to calculate the new value of `tp->rcv_ssthresh` is outlined below:

```
if (length of segment just received >= 128 bytes
    AND current_clamp < max_clamp
```

[6]In particular, this constrains `tp->ack.rcv_mss` to lie between `TCP_MIN_MSS` (default 88) and `TCP_MIN_RCVMSS` (default 536).

[7]The multiple of the MSS used ranges from 2 to 4, depending on the window scale. The Linux implementation follows RFC 2414.

[8]Assuming that `tp->rcvbuf` = 43689, this is 32766. See footnote 10.

[9]This is defined as $\frac{3}{4} \times$ (`tp->rcvbuf`) $\times 2^{-x}$, where x = `sysctl_tcp_app_win` (default 31).

```
     AND current_clamp < space in buffer
     AND no memory pressure)
{  let window = 3/4[10] size of receive buffer
   let truesize = 3/4 length of segment buffer
   if (length of segment data >= truesize)
   {  current_clamp += 2*MSS_1
   }
   else
   {  if (there exists an n such that:
            (1) window >= 2^n * current_clamp
            (2) truesize <= 2^n * length of segment data)
      {  current_clamp += 2*MSS_2
      }
   }
   clip current_clamp to max_clamp
}
```

This algorithm (in `net/ipv4/tcp_output.c::tcp_grow_window`), works on the basis that we do not want to increase the advertised window if we receive lots of small segments (i.e. interactive data flow), as the per-segment overhead (headers and the buffer control block) is very high. We could therefore exhaust the buffer if we were to receive a large window of data, spread across many small segments. In the converse case, when the segment we receive is large (i.e. bulk data flow), the data dominates the segment buffer size and so we wish to increase the window (to avoid the sender being throttled by our receive window).

One has to be careful in the intermediate case, where the segment data is of comparable length to the overheads. In this case, we need to test whether we are able to increase the window clamp without the possibility of the buffer overflowing due to too much overhead (assuming the current segment length continues). This is done by finding a factor, 2^n, such that we could split the receive buffer into 2^n smaller buffers; each larger than the current window clamp. If we can spread the overhead of the current segment amongst all these 'mini-buffers,' such that the segment's data length is larger than the overhead allocated to any of the buffers, then there is room for the advertised window to increase.

Note that two different MSS values are mentioned above. `MSS_1` is the MSS advertised by the host (`tp->advmss`), whereas `MSS_2` is the estimate of the MSS of the remote machine (`tp->ack.rcv_mss`). These are described in the previous section. The use of `tp->ack.rcv_mss` in the intermediate segment size case ensures that the advertised window is opened more gradually.

The result of the algorithm is three observable modes of operation, as shown graphically in Figure 1:

1. Small segments – if the segment size is less than 128 bytes, the advertised window remains constant at $4\times$`tp->advmss`.

2. Medium segments – if the segment size is between 128 and 647 bytes, the advertised window grows linearly by $2\times$`tp->ack.rcv_mss` on receipt of each packet. This continues up to some limit less than `tp->window_clamp`, as defined by the intermediate case of the above algorithm. Note that for segment sizes of less than 536 bytes, `tp->ack.rcv_mss` will not change from its initial value of 536.

[10]This proportion is $1 - 2^{-x}$, where x = `sysctl_tcp_adv_win_scale` (default 2).

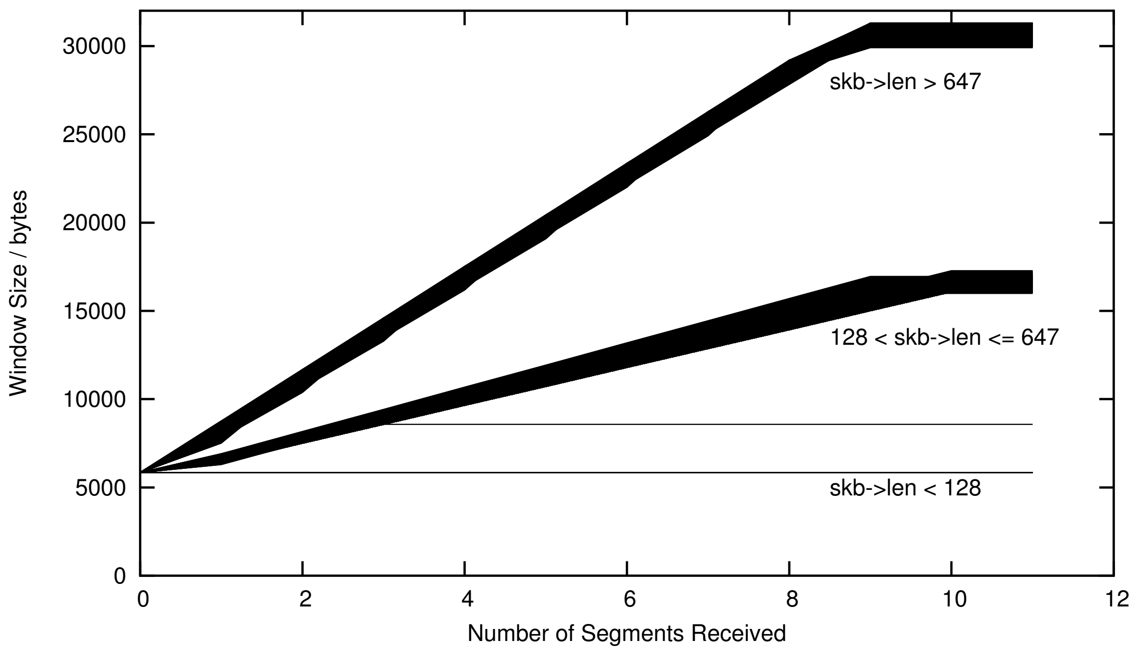

Figure 1: Advertised windows for varying segment sizes

3. Large segments – if the segment size is larger than 647 bytes, the advertised window grows linearly by 2×`tp->advmss` on receipt of each segment. This is limited by the minimum of `tp->window_clamp`, and $\frac{3}{4}$ the size of the free space in the receive buffer. Figure 1 shows the behaviour when this free space is not the limiting factor.

Whenever a segment is sent, the window to advertise is determined by `tcp_select_window`. This is calculated as the free space in the receive buffer, clamped to `tp->rcv_ssthresh`. However, we must round this to a multiple of `tp->ack.rcv_mss`[11], to allow the sender to fill the window with MSS-sized segments. Note that we round to `tp->ack.rcv_mss` even when we are increasing the window by `tp->advmss`[12]. To this end, the data in Figure 1 illustrates the fluctuations in behaviour, given that the actual windows are dependant upon the segment size.

The user may manually set the socket option `TCP_WINDOW_CLAMP`, setting `tp->window_clamp` to a value greater than half the size of the minimum receive buffer (`SOCK_MIN_RCVBUF`). Note that this maximum window clamp may only be set to zero if the socket is in the closed state.

7 Memory Pressure

To understand what happens when the state of memory pressure is reached, we need to consider the events when data is received by `net/ipv4/tcp_input.c::tcp_data_queue`. This function firstly attempts to place the segment into the user's `iovec` (i.e. if the user called `recv()` on the socket, and there is enough space in the buffer specified by the call). If we cannot do this (or we can only transfer part of the segment data across), it must be queued on the socket's receive queue.

The function `tcp_rmem_schedule` tests whether our memory bounds allow us to perform this queueing. If the size of the segment buffer is no greater than the space allocated forward

[11]The MSS value itself is clamped, if it is larger than the minimum of `tp->window_clamp` and the size of the socket's receive buffer.

[12]For example, if a 1000 byte segment is received, with an initial window of 5840, the window is increased to 8000 rather than 8760.

(sk->forward_alloc), we call net/ipv4/tcp.c::tcp_mem_schedule to do the real work. This tries to add the required amount of memory to tcp_memory_allocated , returning '1' if this is successful. There are a number of different circumstances considered, depending on how tcp_memory_allocated compares to the sysctl_tcp_mem values:

Condition	State	Memory Pressure	Success
> sysctl_tcp_mem[2]	over hard limit	yes	no
> sysctl_tcp_mem[1]	over soft limit	yes	yes/no[13]
< sysctl_tcp_mem[1]	under soft limit	yes/no	yes/no[14]
< sysctl_tcp_mem[0]	under limit	no	yes

If sk->rmem_alloc is greater than sk->rcvbuf (i.e. we've allocated more than the buffer size) or the call to tcp_rmem_schedule failed, then we try to prune the receive queue by calling tcp_prune_queue. If this succeeds, we call tcp_rmem_schedule_again, otherwise we drop the segment.

The tcp_prune_queue function firstly tests whether we have allocated more memory than the size of the recieve buffer (sk->rmem_alloc $\geq$ sk->rcvbuf). If this is the case, then we need to recalculate the memory bounds of the socket by calling tcp_clamp_window. Furthermore, even if we have not exceeded the socket memory bounds, if we are under memory pressure then we reset tp->rcv_ssthresh to four times the advertised MSS.

The interesting function to consider then, for our purposes, is tcp_clamp_window, as this is the only place beyond initialisation that tp->window_clamp is recalculated. The following steps are executed:

1. Add up the lengths of all the out of order segments.

2. If we have any out of order segments, expand sk->rcvbuf[15] to at least the actual amount allocated (sk->rmem_alloc), so long as this is less than sysctl_tcp_rmem[2], and we are not under memory pressure.

3. If, after this, we have still allocated more than the receive buffer size:

   (a) If there are no out of order segments, clip tp->window_clamp to the application window. This is calculated as the difference between the data we expect to next receive and the data received but unread (tp->rcv_nxt $-$ tp->copied_seq)[16], and is at least two times the advertised MSS.

   (b) Reset tp->rcv_ssthresh to two times the advertised MSS, clipped to the value of tp->window_clamp.

[13]'Yes', if sk->rmem_alloc is less than sysctl_tcp_rmem[0], or if the socket is using less than its equal share of sysctl_tcp_mem[2] pages shared amongst all the sockets on the host.

[14]As with footnote 13, but also 'yes' if we are not under memory pressure.

[15]We hope that the out of order segments will soon be removed from the queue, so we don't want to clamp the window unnecessarily.

[16]This is halved if we have allocated more than twice the receive buffer's memory, and if it exceeds tp->ack.rcv_mss, this value is subtracted from it.

8 Caching of Socket Statistics

Each entry in the routing table (`struct rtentry` in `include/linux/route.c`) contains various statistics that are cached between connections. These destination caches are stored in a `dst_entry` structure (`include/net/dst.h`). When `connect()` is called on a socket, the function `include/net/route.h::ip_route_connect` performs a routing table lookup, and sets `sk->dst_cache` as a pointer to that in the routing table entry. When the socket is closed, the function `include/net/tcp.h::tcp_update_metrics` updates its destination cache (storing among other things the RTT and the sender's congestion window). The advertised MSS (`tp->advmss`) is also cached, by `net/ipv4/route.c::rt_set_nexthop`.

The apparent behaviour seen in `net/ipv4/tcp_output.c::tcp_connect_init` is quite misleading:

```
if (!tp->window_clamp)
  tp->window_clamp = dst->window;
```

This suggests that the destination cache stores the maximum window clamp. However, this field is never set. This means that `tp->window_clamp` remains at zero after this point, and is initialised in the usual way [see above].

It may be useful to note that destination caching can be disabled by turning off the `DST_HOST` flag in `dst.h`, although this change requires the kernel to be recompiled. If the cache needs to be flushed, the command '`ip route flush cache`' will do so.

Acknowledgements

We acknowledge funding from EC FET-GC project IST-2001-33234 PEPITO, the Royal Society, and the Cambridge–MIT Institute.

A Summary of Important Fields

Below is a summary of the fields discussed:

• `sk->rcvbuf`	The size of socket `sk`'s receive buffer.
• `sk->sndbuf`	The size of socket `sk`'s send buffer.
• `tp->window_clamp`	The maximum possible receive window size.
• `tp->rcv_ssthresh`	The current receive window clamp.
• `tp->rcv_wnd`	The current receive window to advertise.
• `tp->advmss`	The MSS advertised by the host.
• `tp->ack.rcv_mss`	A lower-bound estimate of the peer's MSS.