TCP’s Congestion Control Implementation in Linux Kernel

Somaya Arianfar
Aalto University
Somaya.Arianfar@aalto.fi

ABSTRACT

Linux kernel implements different functionalities related to the network stack. One such functionality is TCP’s congestion control. In this paper, we describe parts of the linux kernel code that deals with TCP’s congestion control implementation. Our main focus here is going through different functions that are used during TCP’s congestion control operation. Although there are many similarities between TCP’s implementation in different kernel versions, this description uses the Linux kernel v 3.6.6.

1. INTRODUCTION

The vast majority of the bytes on the Internet today are transmitted using TCP [7]. As a transport protocol, TCP is expected to provide support for different functionalities such as segmentation, reliability, and congestion control.

TCP’s Congestion Control (CC)[6] is used to prevent congestion collapse[6, 9] in the network. To achieve this goal, TCP uses two basic elements: Acknowledgments (Ack) and Congestion Window (CW). Acknowledgments are used to acknowledge reception of data by the receiver. Congestion Window is used to estimate the bottleneck capacity and control the maximum data that can be unacknowledged and on the fly in a connection.

The congestion control logic in TCP, basically uses the Additive Increase Multiplicative Decrease (AIMD) [1] model for capacity probing. In basic AIMD every acknowledgment results to an increase of maximum MSS (Maximum Segment Size) bytes to the congestion control window, while once per Round Trip Time (RTT) every loss results to reducing the congestion control window to half.

There are two basic phases in the AIMD algorithm: slow start and congestion avoidance. Slow start is usually used at the beginning of a connection. At the slow start phase the congestion control window increases exponentially. After the CW size reaches a predefined threshold (ssthresh), the algorithm enters the congestion avoidance phase. During the congestion avoidance CW’s size doubles once per RTT, at maximum.

Some other concepts have been added to the basic congestion control algorithm [6] later on and then became part of the original algorithm (New Reno[3]). They areSelective ACKs (SACKs), Forward Acknowledgments (FACKs)[8], fast retransmit, and fast recovery. For more information interested readers are referred to [1]. Additionally, throughout the years different competing congestion control algorithms have been developed for TCP. Some of these algorithms include: Vegas[2], BIC[10], and Cubic[5].

These different CC algorithms for TCP, each apply their own tweaks to the basic AIMD model for better performance. Many of these algorithms are also implemented in Linux kernel. The original congestion control algorithm (New Reno) remains wired to the kernel code, while other algorithms could be plugged in, as we will describe later.

In this paper we attempt to describe part of the linux kernel code (v 3.6.6) that deals with the TCP congestion control. We start by describing the code structure and relevant files in the implementation. We then continue by describing the information flows between different function and explain the approach taken in the kernel to implement the algorithmic details.

2. THE CODE STRUCTURE

Linux kernel implements TCP and its different congestion control algorithms. Before going into the kernel implementation details, it is important to note that congestion control and reliability are intertwined functionalities both in TCP’s abstraction and in its kernel implementation. Therefore, the CC related code in kernel v.3.6.6 could be conceptually divided into 4 different categories: the CC framework itself, interface between CC framework and rest of TCP, recovery state machine, and details of different CC algorithms. Here we are going to briefly describe main data structures and related files used in TCP congestion control implementation.

2.1 Important Data Structures

2.1.1 tcp_ca_state

TCP’s congestion control implementation uses a state machine to keep and switch between different states of a connection for recovery purposes. These different states are defined in an enum type in tcp.h.

```c
enum tcp_ca_state {
    TCP_CA_Open = 0,
    #define TCPF_CA_Open (1<<TCP_CA_Open)
    TCP_CA_Disorder = 1,
    #define TCPF_CA_Disorder (1<<TCP_CA_Disorder)
    TCP_CA_CWR = 2,
    #define TCPF_CA_CWR (1<<TCP_CA_CWR)
    TCP_CA_Recovery = 3,
    #define TCPF_CA_Recovery (1<<TCP_CA_Recovery)
    TCP_CA_Loss = 4
    #define TCPF_CA_Loss (1<<TCP_CA_Loss)
};
```
When a connection is Open it is in a normal state, with no dubious events, therefore packets received at this state go through the fast path. The state Disorder is very similar to Open but requires more attention. It is entered when there are some SACKs or dupACKs. In this state some of the processing moves from fast path to the slow path.

State CWR is entered to handle some Congestion Notification event, such as ECN or local device congestion. The Recovery shows that the congestion window has been reduced, and the connection is fast-retransmit stage. State Loss shows that congestion window was reduced due to RTO timeout or SACK reneging.

2.1.2 tcp_congestion_ops

TCP congestion handler interface for different pluggable congestion control algorithms is described in struct tcp_congestion_ops, which is a structure of function call pointers. This structure is defined in tcp.h file.

```
struct tcp_congestion_ops {
    struct list_head list;
    unsigned long flags;
    /* initialize private data (optional) */
    void (*init)(struct sock *sk);
    /* cleanup private data (optional) */
    void (*release)(struct sock *sk);
    /* return slow start threshold (required) */
    u32 (*ssthresh)(struct sock *sk);
    /* lower bound for congestion window (optional) */
    u32 (*min_cwnd)(const struct sock *sk);
    /* do new cwnd calculation (required) */
    void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
    /* call before changing ca_state (optional) */
    void (*set_state)(struct sock *sk, u8 new_state);
    /* call when cwnd event occurs (optional) */
    void (*cwnd_event)(struct sock *sk, enum ca_state new_state);
    /* new value of cwnd after loss (optional) */
    u32 (*undo_cwnd)(struct sock *sk);
    /* hook for packet ACK accounting (optional) */
    void (*pkts_acked)(struct sock *sk, u32 num_ACKed, s32 rtt_us);
    /* get info for inet_diag (optional) */
    void (*get_info)(struct sock *sk, u32 ext, struct sk_buff *skb);
    char name[TCP_CA_NAME_MAX];
    struct module *owner;
};
```

Some of the most important function calls in this structure are as follows:

- init(): This function is called after the first acknowledgment is received and before the congestion control algorithm is called for the first time.
- pkts_acked(): A new acknowledgment results to a call to this function.
- cong_avoid(): This function is called every time an acknowledgment is received and there is possibility for congestion window to increase.
- undo_cwnd(): returns the congestion window of a flow, after a false loss detection (due to false timeout or packet reordering) is confirmed.

2.2 Files

The main files that handle the TCP code in the kernel are listed here. Many of these files could be found under net/ipv4/ folder in the Linux kernel code.

- tcp.h: this file includes the TCP related definitions, including the data structures defined above.
- tcp.c: includes general TCP code and covers the interface between different sockets and the rest of the TCP code.
- tcp_input.c: this is the biggest and most important file dealing with incoming packets from the network. It also contains the code for recovery state machine.
- tcp_output.c: this file deals with sending packets to the network. It contains some of the functions that are called from the CC framework.
- tcp_ipv4.c: IPv4 TCP specific code. This function hands the relevant packets to the CC framework.
- tcp_timer.c: implements timer management functions.
- tcp_cong.c: implements pluggable TCP congestion control support and CC’s core framework with default implementation of New Reno logic.
- tcp_[name of algorithm].c: these files implement different algorithm specific congestion control logic. For example, tcp_vegas.c implements the Vegas logic and tcp_cubic.c implements the TCP Cubic.

Figure 1 shows a high level abstraction of how these different files are organized in the code. Looking at the implementation from the users prospective, the only configurable part in this structure is the choice of CC algorithm. To achieve this goal, the implementation uses pluggable pieces of code in different files.

To register the pluggable congestion control algorithms, their implementation in different files such as tcp_vegas.c and tcp_cubic.c include a static record of struct tcp_congestion_ops to store and initialize the related function calls and algorithm’s name. All these implementations register themselves into the system by calling (hooking to) the tcp_register_congestion_control from tcp_cong.c. However the algorithm used for every connection is set up by kernel initialization or through a Sysctl command. After the congestion control algorithm is set through the main init function calls, it is then used either directly or through function calls from the tcp_input.c.

3. INFORMATION FLOW FOR THE RECOVERY STATE MACHINE

In this section, we describe what happens after a TCP connection is established and data and acknowledgment packets are exchanged. Adjustment of the congestion window and transition through the recovery state machine mainly depends on the reception of ACKs or specific signs of congest-
3.1 Recovery Handling Countermeasures

The main element of handling ACKs and recognizing congestion on the data sender side are retransmission queue and retransmission timer. Transmission of a data packet is always followed by placing a copy of that data packet in a retransmission queue. The reception of an ACK then results to removing related copies in the retransmission queue.

Each time a data packet is sent a retransmission timer is set for that packet. This timer counts down over time. In basic scenario, a packet is considered to be lost if its retransmission timer expires before an acknowledgment is received for that packet. In that case lost packets are retransmitted.

Anyhow, there are three tag bits, to mark packets in retransmission queue: SACKED (S), RETRANS (R) and LOST (L). Packets in queue with these bits set are counted in variables sacked_out, retrans_out and lost_out, correspondingly. While calculating the proper value for retrans_out for counting the number of retransmitted packets is pretty straight forward, marking right set of packets and calculating proper values for sacked_out and lost_out are a bit more complicated.

sacked_out counts the number of packets, which arrived to receiver out of order and hence not ACKed. With SACKs this number is simply amount of SACKed data. Without SACKs this is calculated through counting duplicate ACKs. For marking the lost packet and calculating the lost_out, there are essentially two algorithms:

- **FACK**: As soon as the algorithm decides that something is lost, it decides that _all_ not SACKed packets until the most forward SACK are lost. I.e. lost_out = packets_out - sacked_out and left_out = packets_out. It seems to be a correct estimate, if network does not reorder packets. But reordering can invalidate this estimation. There, the implementation uses FACK by default until reordering is suspected on the path.
- **NewReno**: when Recovery is entered, the assumption is that one segment is lost (classic Reno). When the connection is in Recovery state and a partial ACK arrives, the assumption turns to be that one more packet is lost (NewReno). This heuristics are the same in NewReno and SACK.

After a RTO (retransmission timeout), when all the queue is considered as lost, lost_out equals packets_out. Other congestion control related variables and parameters in the code include:

\[
\text{in\_flight} = \text{packets\_out} - \text{left\_out} + \text{retrans\_out}
\]

In the equation above, packets_out is highest data segment transmitted (SND.NXT) - the first unacknowledged segment (SND.UNA) counted in packets. While left_out is number of segments left network, but not ACKed yet.

\[
\text{left\_out} = \text{sacked\_out} + \text{lost\_out}
\]

3.2 Recovery State Machine

As noted earlier, functions in tcp_input.c deal with the received packets. Therefore, the function calls we describe here are mostly from tcp_input.c unless mentioned otherwise.

As can be seen in Figure 2, reception of a data packet at the end-host triggers a call to tcp_event_data_recv(). This function in itself is called by tcp_v4_do_rcv() in tcp_ipv4.c. The call to tcp_event_data_recv() results to measuring the MSS and RTT, and triggers an ACK (or SACK). The acking process benefits from two modes of operation:

- **Quick ACK**: It is used at the beginning of a TCP connection so that the congestion window can grow quickly.
- **Delayed ACK**: A connection can switch to this mode after a while. In this case an ACK is sent for multiple packets.

TCP switches between these two modes depending on the congestion experienced.

Incoming ACKs are processed in tcp_acks(). In this function the sequence numbers are processed to clean the
Packet arrival

Packet arrival

tcp_input.c

Figure 2: Packet reception to recovery state machine

retransmission queue, SACKs, ECN(ECE) flag, duplicate
ACKs, reordering detection, advancing CWND, and in gen-
eral new congestion control. Here, we describe two of
the most important functions that are called while processing
ACKs in tcp acks().

3.2.1 tcp_sacktag_write_queue()

Incoming SACKs are processed in tcp_sacktag
_write_queue(). In here tcp_is_sackblock_valid() tags the
retransmission queue when SACKs arrive. This
function is also used for sack block validation. SACK block
range validation checks that the received SACK block fits to
the expected sequence limits, i.e., it is between SND.UNA
and SND.NXT. There is another function to limit sacked_out
so that sum with lost_out isn’t ever larger than packets_out.

3.2.2 tcp_fastretrans_alert()

The main congestion control logic and its related state
transitions are implemented in tcp_fastretrans_alert() function. This function describes the Linux NewReno/SACK-
/FACK/ECN state machine and it is called from tcp acks() in case of dubious ACKs. Dubious ACKs occur either when
the congestion is seen for the first time or in other word the
arrived ACK is unusual e.g. SACK, or when TCP is already
in a congestion state (any state other than open).

The most important logic executed in tcp_fastretrans
_alert() is described in the following:
tcp_check_sack_reneging(): Packets in the
retransmission queue are marked when a SACK is received
(through another function as mentioned earlier). However,
if the received ACK/SACK points to a remembered SACK, it
probably relates to erroneous knowledge of SACK. tcp_check
_sack_reneging() function deals with such erroneous
situations.
tcp_time_to_recover(): This function checks pa-
rameters such as number of lost packets in a connection to
decide whether its the right time to move to Recovery state.
In other word, this function determines the moment when we
decide that hole is caused by loss, rather than by a reorder.
If it decides that is the recovery time; the CA State would
switch to Recovery.

tcp_try_to_open(): If its not yet the time to move
to recovery state, this function will check for switching the
state and other proper reactions based on the indication in
the packet. For example, if the packet is indicating ECE
then the state will switch to CWR. Then, CW will be reduced
by calling tcp_cwnd_down.
tcp_update_scoreboard(): This function will mark
the lost packets. Depending on the choice of SACK or FACK[]
all the packets which were not sacked (till the maximum seq
number sacked) might be marked as lost packets. Also unac-
nowledged packets that have expired retransmission timers
are marked as lost in this function. Some sort of recounting
for lost, sacked and left out packets is also triggered through
this function.
tcp_xmit_retransmit_queue(): This function trig-
gers retransmission of lost packets. It decides, _what_ we
should retransmit to fill holes, caused by lost packets.
tcp_try_undo_<something>(): The most logically
complicated part of algorithm is undo heuristics. False re-
transmits can occur due to both too early fast retransmit (re-
ordering) and underestimated RTO. Analyzing timestamps
and D-SACKs can identify the occurrence of such false re-
transmits. Detection of false retransmission and congestion
window reduction could be undone and the recovery phase
could be aborted. This logic is hidden inside several func-
tions named tcp try undo <something>.

The functions above are mainly used for recovery state
machine, and they describe how to move back and forth
in the CW, when there is a need for retransmission. How-
ever, we discuss the implementation for calculating the ac-
tual amount of increase/decrease in the CW size in the next
section.

4. CONGESTION CONTROL ALGORITHMS

The basic congestion control functionalities core function-
ality is defined in tcp_cong.cc. TCP’s original Reno al-
gorithm[6] is directly implemented in tcp_reno_cong
_avoid(). While in case of other algorithms, functions
such as tcp slow start() and tcp_cong_avoid_ai() move the congestion window forward depending on the cal-
culations done by different algorithm. These functions are
called from different places in the code, for example from
tcp_input.c or from any of the tcp [name of algorithm].c files.

As mentioned earlier different pluggable congestion con-
trol algorithms are implemented in tcp [name of algorithm].c files. They register themselves and their function calls to the
system through initiating an instance of tcp_congestion_ops. One of these algorithms, which we are going to explain here,
is TCP cubic. TCP cubic initiates its function calls in the
code as follows:

code
static struct tcp_congestion_ops cubictcp
__read_mostly = {
    .init = bictcp_init,
    .ssthresh = bictcp_recalc_ssthresh,

4
After the initialization phase, bictcp_acked() is called on every received acknowledgment and triggers proper function calls for increasing/decreasing the congestion window. This function basically tracks delays and delayed acknowledgment ratio based on the following:

\[
\text{slidingwindowratio} = \frac{(15 \times \text{ratio} + \text{sample})}{16}
\]

The reason for tracking delayed ACKs is the logic implemented in TCP cubic’s code. In Cubic, congestion window is always increased if the ACK is okay, and the flow is limited by the congestion window. If the receiver is using delayed acknowledgement, the code needs to adapt to that problem.

TCP cubic’s code integrates its own implementation for changing the congestion window size both at the slow start phase and at the congestion avoidance phase. Therefore, bictcp_acked() can also result to a call to hystart_update() at the slow start phase increases the congestion control window based on the HyStart [4] algorithm instead of the standard TCP slow start logic. In this implementation HyStart logic is triggered when CW is larger than some threshold (hystart_low_window read_mostly = 16).

Cubic Hystart uses RTT-based heuristics to exit slow start early on, before losses start to occur. Cubic HyStart use delays for congestion indication, but it exits slow starts at the detection of congestion and enters cubic’s standard congestion avoidance.

For the congestion avoidance phase, the window growth function of TCP cubic [5] uses the following equation:

\[
CW(t) = C \times (t - K)^3 + CW_{max}\]

where C is a CUBIC parameter, t is the elapsed time from the last window reduction, and K is the time period that the above function takes to increase CW to CW_max when there is no further loss event and is calculated by using the following equation:

\[
K = \text{cubic rooting}(CW_{max} \times \text{beta}/C)
\]

where beta is the multiplication decrease factor (at the time of a loss window decreases to beta \( CW_{max} \)).

In the code, bictcp_cong_avoid() is called during the congestion avoidance phase. The calculation of proper CW size at this stage is done based on the logic above and in bictcp_update() function. A summarized view of the function calls resulting to the ultimate final of the CW size, could be followed in this piece of code from the tcp_cubic.c file:

```c
static void bictcp_cong_avoid(struct sock *sk, u32 ack, u32 in_flight) {
    struct tcp_sock *tp = tcp_sk(sk);
    struct bictcp *ca = inet_csk_ca(sk);
    if (!tcp_is_cwnd_limited(sk, in_flight))
        return;
    if (tp->snd_cwnd <= tp->snd_ssthresh) {
        if (hystart && after(ack, ca->end_seq))
            bictcp_hystart_reset(sk);
        tcp_slow_start(tp);
    } else {
        bictcp_update(ca, tp->snd_cwnd);
        tcp_cong_avoid_ai(tp, ca->cnt);
    }
}
```