TCP's Congestion Control Implementation in Linux Kernel

Somaya Arianfar
Aalto University
Somaya.Arianfar@aalto.fi

ABSTRACT

The Linux kernel implements various parts of the network stack. This paper is part of a joint attempt to describe the structure and the implementation details of the kernel code. In this specific paper, we explain parts of the Linux kernel code that deals with TCP's congestion control implementation. For this description we use the Linux kernel v 3.6.6. Our main focus is describing the most common pieces of the congestion control related code, which includes the congestion control framework itself, the interface between congestion control framework and rest of TCP, recovery state machine, and details of an example congestion control algorithms, TCP Cubic.

1. INTRODUCTION

The vast majority of the bytes on the Internet today are transmitted using Transmission Control Protocol (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[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. 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 congestion window size reaches a predefined threshold (`ssthresh`), the algorithm enters the congestion avoidance phase. During the congestion avoidance congestion window'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 in the form of New Reno[3] algorithm. Some of these added features include: Selective ACKs (SACKs) [1], Forward Acknowledgments (FACKs)[8], fast retransmit, and fast recovery. For more information regarding these features, the interested reader is referred to the IETF standard document on TCP's congestion control[1]. Additionally, throughout the years various other competing congestion control algorithms have been developed for TCP. Some of these algorithms include: Vegas[2], BIC[11], and Cubic[5].

These different congestion control 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 congestion control related code in kernel v.3.6.6 could be conceptually divided into 4 different categories: the congestion control framework itself, interface between congestion control framework and rest of TCP, recovery state machine, and details of different congestion control algorithms. Here we are going to briefly describe main data structures and related files used in TCP congestion control implementation.

2.1 Important Data Types

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.

`Open`: 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. TCP fast path eliminates the extra processing that is required on flagged packets or in the case of suspicious loss or out-of-order delivery.

`Disorder`: This state 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.

`CWR`: State `CWR` is entered to handle some Congestion Notification event, such as ECN or local device congestion.

`Recovery`: This state shows that the congestion window has been reduced, and the connection is fast-retransmit stage.

`Loss`: 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 (*cong_avoid)(struct sock *sk, u32
             ACK, u32 in_flight);
        /* 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
             tcp_ca_event ev);
        /* 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()`: An acknowledgment that acknowledges some new packets, results to a call to this function. Number of packets that are acknowledged by this acknowledgments is paseed through the `num_acked` argument.

`cong_avoid()`: This function is called every time an acknowledgment is received and the congestion control state allows 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 deal with the TCP code in the kernel are listed here. Many of these files could be found under `net/ipv4/` directory in the Linux kernel code, unless it is mentioned otherwise.

tcp.h: this files includes the TCP related definitions, including the data structures defined above. This file exist both in `include/net/` and `include/linux/` directories.

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 files deals with sending packets to the network. It contains some of the functions that are called from the congestion control framework.

tcp_ipv4.c: IPv4 TCP specific code. This function hands the relevant packets to the congestion control framework.

tcp_timer.c: implements timer management functions.

tcp_cong.c: implements pluggable TCP congestion control support and congestion control'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.

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 congestion like timeouts and Explicit Congestion Notification (ECN [10]) bits. Simple form of TCP signals congestion by packet drops. ECN, however, allows for congestion notification without dropping packets. In case of congestion, an ECN-aware router can mark in the IP header instead of dropping the packet. The receiver of the packet echoes back the congestion indication to the sender by setting ECN_Echo (ECE [10])) flag in the TCP header.

Our main focus here is on handling received packets. But first we briefly describe countermeasures that sender of a data packet takes to be able to handle ACKs and recognize congestion later on.

3.1 Recovery Handling Countermeasures

The main elements 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. In current kernel the retransmission queue is defined as a member of `struct sock` and under the name `write 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.

`sacke_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: Before describing the FACK algorithm to calculate the `lost_out`, we introduce another variable called `fackets_out`. In the code, `fackets_out` includes both SACKed packets up to the highest received SACK block so far and holes in between them. As soon as the FACK algorithm decides that something is lost, it decides that *all* not SACKed packets until the most forward SACK are lost. I.e.`lost_out = fackets_out - sacked_out` and `left_out = fackets_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. Reordering is often suspected with the arrival of duplicate Acks and SACKS.

- 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.

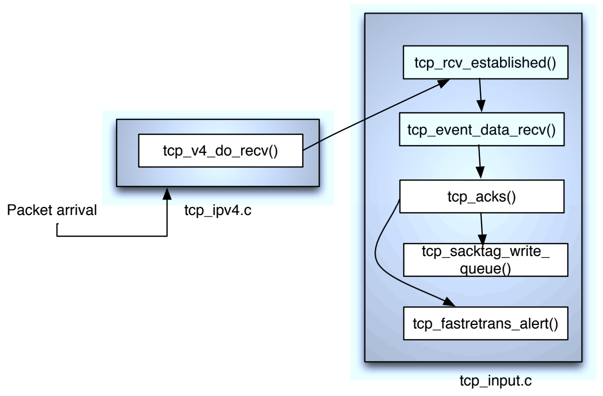

Figure 1: Packet reception to recovery state machine

Within TCP's retransmission logic[1]: with occurrence of the retransmission timeout, the TCP sender enters the retransmission timeout recovery where the congestion window is initialized to one segment and the whole window, which remains unacknowledged, is retransmitted. Therefore, After a RTO (retransmission timeout), when the whole queue is considered as lost, `lost_out` equals `packets_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 1, 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_rcv_established()` which in turn 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. Per default the quick ACK mechanism is enabled and ACK packets are triggered instantly to raise the congestion window fast especially for bulk data transfers. After a while when congestion window has grown enough delayed ACKs could be used to reduce the excessive protocol processing.

Incoming ACKs are processed in `tcp_acks()`. In this function the sequence numbers are processed to clarify what are the required actions after receiving an ACK. For instance some of the proper reactions could be: cleaning the retransmission queue, marking the SACKed packets, reacting to duplicate ACKs, reordering detection, and advancing congestion window. Here, we describe two of the most im-

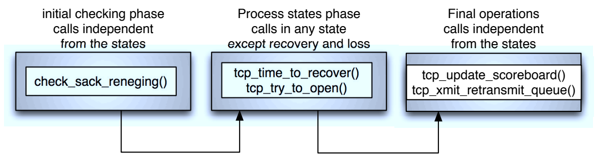
initial checking phase calls independent from the states
Process states phase calls in any state except recovery and loss
Final operations calls independent from the states
check_sack_reneging()
tcp_time_to_recover()
tcp_try_to_open()
tcp_update_scoreboard()
tcp_xmit_retransmit_queue()

Figure 2: Recovery related functional calls in tcp_fast_retrans_alert()

portant 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 basic recovery 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 the TCP connection has already experienced something unusual that has caused it to move from the connection open state to any other state in the recovery state machine as described below and in Sec. **??**.

As can be seen in Figure. 2 different set of functions are called to check the state of a connection and do the proper operations at each state. The most important function calls executed in `tcp_fastretrans_alert()` are 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 parameters 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 ECE flag in the packet header is set, then the state will switch to CWR. Then, congestion window 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 unacknowledged packets that have expired retransmission timers are marked as lost in this function. All the markings in this function triggers recounting for lost, sacked and left out packets.

`tcp_xmit_retransmit_queue()`: This function triggers 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 retransmits can occur due to both too early fast retransmit (reordering) and underestimated RTO. Analyzing timestamps and D-SACKs can identify the occurrence of such false retransmits. Detection of false retransmission and congestion window reduction could be undone and the recovery phase could be aborted. This logic is hidden inside several functions named `tcp_try_undo_<something>`.

The functions above are mainly used for recovery state machine, and getting around the retransmission queue when there is a need for retransmission. However, we discuss the implementation for calculating the actual amount of increase/decrease in the congestion window size in the next section.

4. CONGESTION CONTROL ALGORITHMS

The basic congestion control functionalities core functionality is defined in `tcp_cong.cc`. TCP's original Reno algorithm[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 calculations 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.

Figure 3 shows a high level abstraction of different congestion control algorithms are set and initialized in the code. Looking at the implementation from the users prospective, the only configurable part in this structure is the choice of congestion control 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

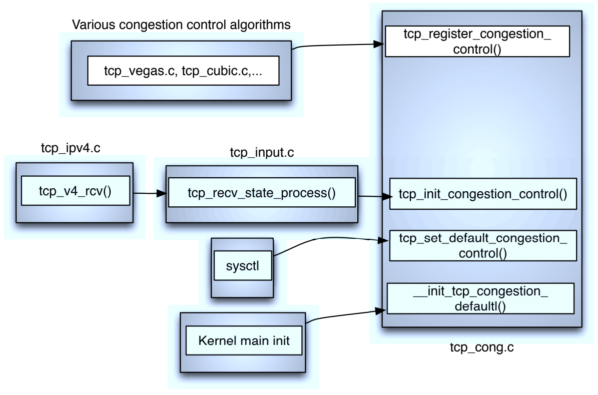
Various congestion control algorithms
tcp_vegas.c, tcp_cubic.c,...
tcp_register_congestion_control()
tcp_ipv4.c
tcp_v4_rcv()
tcp_input.c
tcp_recv_state_process()
tcp_init_congestion_control()
sysctl
tcp_set_default_congestion_control()
__init_tcp_congestion_defaultl()
Kernel main init
tcp_cong.c

Figure 3: TCP congestion control related setting in Linux kernel (v 3.6.6)

is set up by kernel initialization or through a `sysctl` command. After the congestion control algorithm is set, defined hooks in `tcp _congestion_ops` are used to access the relevant algorithm specific functions from the rest of the code.

Implementation of all the algorithms more or less depends on the calculation of flight size and estimated size of the congestion window. Flight size shows the amount of data that has been sent but not yet cumulatively acknowledged. Therefore, it could be used to progress the congestion window, or estimate the correct value for e.g. sshtresh. In optimal situation, the flight size should be a reflection of bandwidth delay product.

In the code the flight size is shown through the `in_flight` variable.

$$in\_flight = packets\_out + retrans\_out - left\_out$$

In the equation above, `packets_out` is the highest data segment transmitted (SND.NXT) minus the first unacknowledged segment (SND.UNA) counted in packets. As shown in the equation, the estimation also needs to consider the number of retransmitted packets as part of the flight size calculation.

Theoretically, the sum of `retrans_out` and `packets_out` should show the flight size at any moment in time. However, in practice because of the usage of SACKs and other features, `packets_out` in its own reflects also those packets that have left the network in form of SACKed packets or lost packets, and thus are not in the flight anymore. In the equation above, `left_out` is number of these packets that left network, but not ACKed yet.

$$left\_out = sacked\_out + lost\_out$$

4.1 TCP Cubic in theory

The original Reno algorithm for TCP congestion control have been designed in those days when both the link capacities and round trip times were limited. Now a days, it is a known problem that as the bandwidth delay product grows TCP's sluggish behavior in increasing the congestion window size could result to under-utilization of network resource.

TCP cubic is one of the newest modifications to the TCP congestion control algorithm that changes the linear window growth function of TCP to a cubic function, in order to improve the bandwidth utilization in case of high bandwidth delay product networks. It also achieves a better level of fairness among flows with different round trip times. All these attributes make the cubic the default congestion control algorithm in Linux.

As it comes from the name of this algorithm the window growth function is a cubic function of elapsed time since the last packet loss. The algorithm registers a W_max to be the window size where the last packet loss event has happened. The algorithm then performs a multiplicative decrease of congestion window by a constant decrease factor. Afterwards, when the algorithm enters into congestion avoidance phase from fast recovery, it starts to increase the window using the cubic function that we will describe later. The cubic growth continues in a concave form until the window size becomes equal to the W_max. After that, the cubic function turns into a convex profile and the convex window growth continues from there. The concave-convex style of window growth helps the stability and better utilization of network resources [5].

4.2 TCP Cubic in the code

As mentioned earlier different pluggable congestion control 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:

```
static struct tcp_congestion_ops cubictcp
    __read_mostly = {
      .init         = bictcp_init,
      .ssthresh     = bictcp_recalc_ssthresh,
      .cong_avoid   = bictcp_cong_avoid,
      .set_state    = bictcp_state,
      .undo_cwnd    = bictcp_undo_cwnd,
      .pkts_acked   = bictcp_acked,
      .owner        = THIS_MODULE,
      .name         = "cubic",
};
```

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:

$$slidingwindowratio = (15 * ratio + sample)/16$$

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()`. `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 congestion window 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 cubic equation:

$$CW(t) = C * (t - K)^3 + CW_max$$

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 current congestion window (CW) to CW_max when there is no further loss event and is calculated by using the following equation:

$$K = cubic_root(CW_max * beta/C)$$

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 congestion window 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 congestion window size, could be followed in the tcp_cubic.c file:

5. CONCLUSIONS

In this paper, we have described TCP's congestion control implementation in the Linux kernel v 3.6.6. Our main focus has been explaining the recovery state machine in the code and high level abstractions of congestion control's algorithm implementation.

6. REFERENCES

[1] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[2] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. Tcp vegas: new techniques for congestion detection and avoidance. *SIGCOMM Comput. Commun. Rev.*, 24(4):24–35, 1994.

[3] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, 2004.

[4] S. Ha and I. Rhee. Taming the elephants: New tcp slow start. *Comput. Netw.*, 55(9):2092–2110, June 2011.

[5] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, July 2008.

[6] V. Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM '88*, volume 18, pages 314–329, Stanford, CA, USA, Aug. 1988. ACM.

[7] C. Labovitz, D. McPherson, S. Iekel-Johnson, J. Oberheide, F. Jahanian, and M. Karir. Internet Observatory Report. *Proc. NANOG-47*, 2009.

[8] M. Mathis and J. Mahdavi. Forward acknowledgement: refining tcp congestion control. *SIGCOMM Comput. Commun. Rev.*, 26(4):281–291, Aug. 1996.

[9] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Jan. 1984.

[10] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, Sept. 2001.

[11] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *Proc. of IEEE INFOCOM 2004*, volume 4, pages 2514 – 2524, Hong Kong, March 2004.