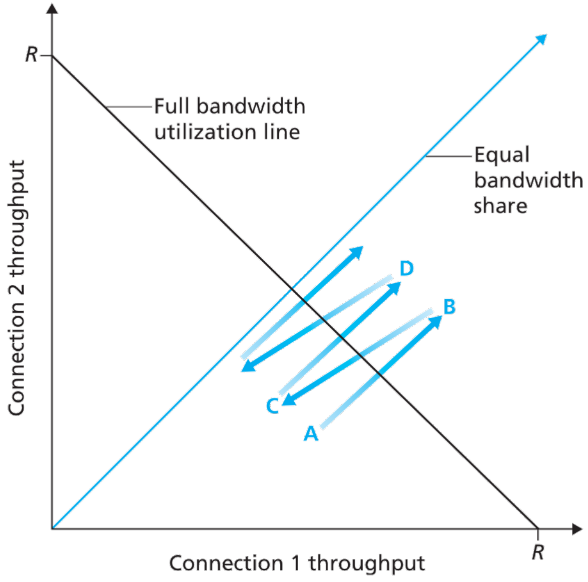
Counterfeiting Congestion Control Algorithms

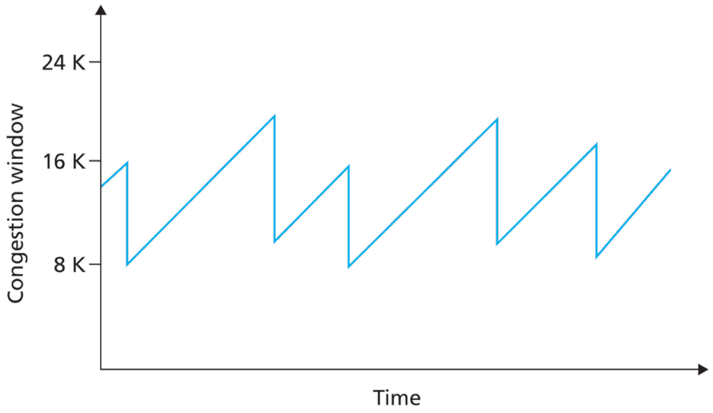
Margarida Ferreira, Akshay Narayan, Inês Lynce,

Ruben Martins, Justine Sherry

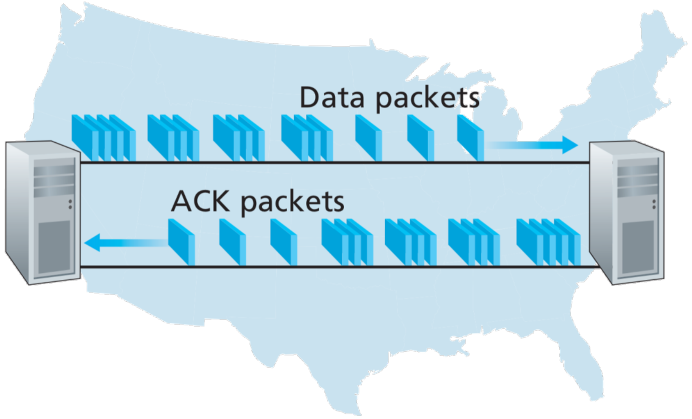
Desirable CCA properties



Fairness: whether competing applications share network bandwidth fairly



Stability: how stable bandwidth allocations are (or whether performance oscillates)



Utilization: whether network links are utilized efficiently

Pictures from Kurose, J. F., & Ross, K. W. (2001). *Computer networking: A top-down approach featuring the Internet*

Desirable CCA properties

Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms

Ranysha Ware, Carnegie Mellon University, rware@cs.cmu.edu
Matthew K. Mukerjee, Nefeli Networks, mukerjee@nefeli.io
Srinivasan Seshan, Carnegie Mellon University, srini@cs.cmu.edu
Justine Sherry, Carnegie Mellon University, sherry@cs.cmu.edu

ABSTRACT

The Internet community faces an explosion in new congestion control algorithms such as Copa, Sprout, PCC, and BBR. In this paper, we discuss considerations for deploying new algorithms on the Internet. While past efforts have focused on achieving 'fairness' or 'friendliness' between new algorithms and deployed algorithms, we instead advocate for an approach centered on quantifying and limiting harm caused by the new algorithm on the status quo. We argue that a harm-based approach is more practical, more future proof, and handles a wider range of quality metrics than traditional notions of fairness and friendliness.

ACM Reference Format:
Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Beyond Jain's Fairness Index: Setting the Bar For The Deployment of Congestion Control Algorithms. In *The 18th ACM Workshop on Hot Topics in Networks (HotNets '19), November 13–15, 2019, Princeton, NJ, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3365609.3365855

Axiomatizing Congestion Control

DORON ZARCHY, Hebrew University of Jerusalem
RADHIKA MITTAL, University of Illinois at Urbana-Champaign
MICHAEL SCHAPIRA, Hebrew University of Jerusalem
SCOTT SHENKER, UC Berkeley, ICSI

Toward Formally Verifying Congestion Control Behavior

Venkat Arun*, Mina Tahmasbi Arashloo†, Ahmed Saeed*, Mohammad Alizadeh*, Hari Balakrishnan*
MIT CSAIL* and Cornell University†
Email: ccac@mit.edu Website: https://projects.csail.mit.edu/ccac

Experimental Evaluation of BBR Congestion Control

Mario Hock, Roland Bless, Martina Zitterbart
Karlsruhe Institute of Technology
Karlsruhe, Germany
E-Mail: mario.hock@kit.edu, bless@kit.edu, zitterbart@kit.edu

Stability Analysis of Explicit Congestion Control Protocols

Hamsa Balakrishnan, Nandita Dukkipati, Nick McKeown and Claire J. Tomlin
Stanford University, Stanford, CA 94305
Email: {hamsa, nanditad, nickm, tomlin}@stanford.edu

Abstract

Much recent attention has been devoted to analyzing the stability of congestion control algorithms, in the context of TCP modifications (e.g., TCP/RED [10], [15], FAST [17]) and new protocols (e.g., XCP [21], RCP [8], TeXCP [20]). The control-theoretic framework used in most previous work is linear systems theory. The analyses assume that the system can be well approximated by linearization, and the linearization is then used to derive conditions for stability using techniques based on the Bode or Nyquist criteria.

We show that linearization is *not* a good approximation when the queue lengths are close to zero. Because the goal of several congestion control algorithms is to keep queue lengths small, the linearization turns out to be the most inaccurate precisely in the realm in which a good algorithm would hope to operate. We show, in the context of explicit congestion control protocols like XCP and RCP, that the stability region derived from traditional Nyquist analysis is not an accurate representation of the actual stability region. Using XCP as an example, we then show that modeling the congestion control algorithm as a switched linear control system with time delay, and using new Lyapunov stability conditions can provide sound and more general sufficient conditions for stability than previously derived. For piecewise linear systems with time-delay, the proposed conditions guarantee global stability. We show that the proposed framework can be used to analyze the stability of congestion control protocols in the presence of heterogeneous delays.

Stanford University Department of Aeronautics and Astronautics Report: SUDAAR No. 776, September 9, 2005. This research was supported by an NSF Career award (ECS-9985072). H. Balakrishnan was supported by a Stanford Graduate Fellowship.

Companies have deployed CCAs that are not fair

Google's Network Congestion Algorithm Isn't Fair, Researchers Say

But at least it's open source and transparent, unlike efforts by countless other companies, those same data scientists say.

CMU Researchers Find Google's New Congestion Control Algorithm Treats Data Unfairly

Tuesday, October 22, 2019 - by Daniel Tkacik

If the Internet had its own superhero, it might be the congestion control algorithm (CCA), an essential piece of code internet giants use to ensure that the web isn't crippled by a massive data traffic jam. They've been used since the 1980s to slow

The ISP Column

A column on things Internet

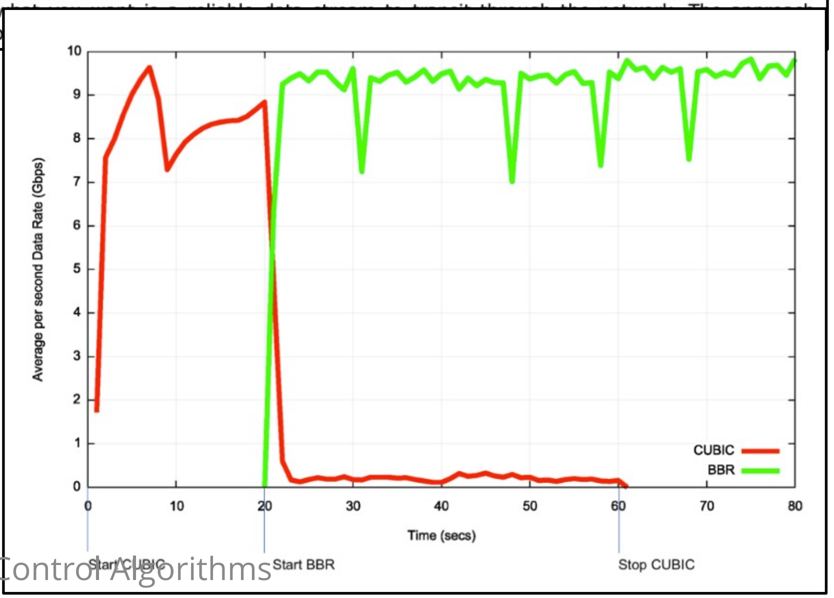
Other Formats:

BBR TCP

May 2017

Geoff Huston

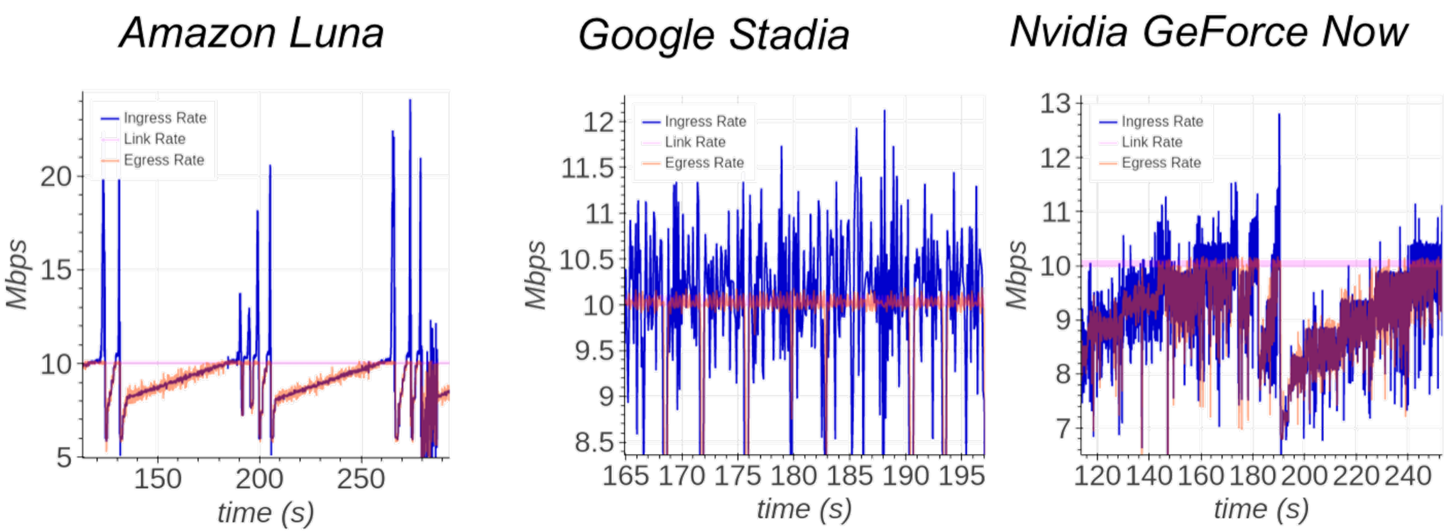
The Internet was built using an architectural principle of a simple network and agile edges. The basic approach was to assume that the network is a simple collection of buffered switches and circuits. As packets traverse the network they are effectively passed from switch to switch. The switch selects the next circuit to use to forward the packet closer to its intended destination. If the circuit is busy, the packet will be placed on a queue and processed later, when the circuit is available. If the queue is full, then the packet is discarded. This network behaviour is not entirely useful if w
adopted b



Companies are using new proprietary CCAs for different applications

- Video streaming
- Online gaming
- Videoconferencing

Steady State at Low Bandwidth



D. Caban, D. Ray and S.Seshan. Understanding Congestion Control for Cloud Game Streaming. CMU REU 2020

Our goal is to reverse engineer CCAs

How do we reverse engineer CCAs?
Program Synthesis

Program Synthesis

$$f(1, 2, 3) = 7$$

$$f(2, 3, 4) = 8$$

$$f(5, 3, 5) = 8$$

$$f(2, 4, 5) = 12$$

$$f(5, 15, 10) = 35$$

$$f?$$

Program Synthesis

$f(1, 2, 3) = 7$

$f(2, 3, 4) = 8$

$f(5, 3, 5) = 8$

$f(2, 4, 5) = 12$

$f(5, 15, 10) = 35$

Synthesizer

$f(x, y, z) = x + y*z/x$

CCA Synthesis

f(input) = output

CCA Synthesis

***CCA**(input) = output*

CCA Synthesis

*CCA(**network signals**) = output*

ACKs MSS

Sends RTT

…

CCA Synthesis

*CCA(network signals, **state**) = output*

CWND

SSThresh

…

CCA Synthesis

*CCA(network signals, **state**) = **new state***

CWND

SSThresh

…

CCA Synthesis

Network traces → Synthesizer → CCA

CCA code

```c
		tcp_rate_skb_delivered(sk, skb, sack->rate);

		/* Initial outgoing SYN's get put onto the write_queue
		 * just like anything else we transmit.  It is not
		 * true data, and if we misinform our callers that
		 * this ACK acks real data, we will erroneously exit
		 * connection startup slow start one packet too
		 * quickly.  This is severely frowned upon behavior.
		 */
		if (likely(!(scb->tcp_flags & TCPHDR_SYN))) {
			flag |= FLAG_DATA_ACKED;
		} else {
			flag |= FLAG_SYN_ACKED;
			tp->retrans_stamp = 0;
		}

		if (!fully_acked)
			break;

		tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);

		next = skb_rb_next(skb);
		if (unlikely(skb == tp->retransmit_skb_hint))
			tp->retransmit_skb_hint = NULL;
		if (unlikely(skb == tp->lost_skb_hint))
			tp->lost_skb_hint = NULL;
		tcp_highest_sack_replace(sk, skb, next);
		tcp_rtx_queue_unlink_and_free(skb, sk);
	}

	if (!skb)
		tcp_chrono_stop(sk, TCP_CHRONO_BUSY);

	if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
		tp->snd_up = tp->snd_una;

	if (skb) {
		tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);
		if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
			flag |= FLAG_SACK_RENEGING;
	}

	if (likely(first_ackt) && !(flag & FLAG_RETRANS_DATA_ACKED)) {
		seq_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, first_ackt);
		ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, last_ackt);

		if (pkts_acked == 1 && last_in_flight < tp->mss_cache &&
		    last_in_flight && !prior_sacked && fully_acked &&
		    sack->rate->prior_delivered + 1 == tp->delivered &&
		    !(flag & (FLAG_CA_ALERT | FLAG_SYN_ACKED))) {
			/* Conservatively mark a delayed ACK. It's typically
			 * from a lone runt packet over the round trip to
			 * a receiver w/o out-of-order or CE events.
			 */
			flag |= FLAG_ACK_MAYBE_DELAYED;
		}
	}
	if (sack->first_sackt) {
		sack_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->first_sackt);
		ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->last_sackt);
	}
	rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us,
					ca_rtt_us, sack->rate);

	if (flag & FLAG_ACKED) {
		flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
		if (unlikely(icsk->icsk_mtup.probe_size &&
			     !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
			tcp_mtup_probe_success(sk);
		}

		if (tcp_is_reno(tp)) {
			tcp_remove_reno_sacks(sk, pkts_acked, ece_ack);

			/* If any of the cumulatively ACKed segments was
			 * retransmitted, non-SACK case cannot confirm that
```

```c
			 * progress was due to original transmission due to
			 * lack of TCPCB_SACKED_ACKED bits even if some of
			 * the packets may have been never retransmitted.
			 */
			if (flag & FLAG_RETRANS_DATA_ACKED)
				flag &= ~FLAG_ORIG_SACK_ACKED;
		} else {
			int delta;

			/* Non-retransmitted hole got filled? That's reordering
			if (before(reord, prior_fack))
				tcp_check_sack_reordering(sk, reord, 0);

			delta = prior_sacked - tp->sacked_out;
			tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
		}
	} else if (skb && rtt_update && sack_rtt_us >= 0 &&
		   sack_rtt_us > tcp_stamp_us_delta(tp->tcp_mstamp,
						    tcp_skb_timestamp_us(skb)))
		/* Do not re-arm RTO if the sack RTT is measured from data sent
		 * after when the head was last (re)transmitted. Otherwise the
		 * timeout may continue to extend in loss recovery.
		 */
		flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
	}

	if (icsk->icsk_ca_ops->pkts_acked) {
		struct ack_sample sample = { .pkts_acked = pkts_acked,
					     .rtt_us = sack->rate->rtt_us,
					     .in_flight = last_in_flight };

		icsk->icsk_ca_ops->pkts_acked(sk, &sample);
	}

#if FASTRETRANS_DEBUG > 0
	WARN_ON((int)tp->sacked_out < 0);
	WARN_ON((int)tp->lost_out < 0);
	WARN_ON((int)tp->retrans_out < 0);
	if (!tp->packets_out && tcp_is_sack(tp)) {
		icsk = inet_csk(sk);
		if (tp->lost_out) {
			pr_debug("Leak l=%u %d\n",
				 tp->lost_out, icsk->icsk_ca_state);
			tp->lost_out = 0;
		}
		if (tp->sacked_out) {
			pr_debug("Leak s=%u %d\n",
				 tp->sacked_out, icsk->icsk_ca_state);
			tp->sacked_out = 0;
		}
		if (tp->retrans_out) {
			pr_debug("Leak r=%u %d\n",
				 tp->retrans_out, icsk->icsk_ca_state);
			tp->retrans_out = 0;
		}
	}
#endif
	return flag;
}

static void tcp_ack_probe(struct sock *sk)
{
	struct inet_connection_sock *icsk = inet_csk(sk);
	struct sk_buff *head = tcp_send_head(sk);
	const struct tcp_sock *tp = tcp_sk(sk);

	/* Was it a usable window open? */
	if (!head)
		return;
	if (!after(TCP_SKB_CB(head)->end_seq, tcp_wnd_end(tp))) {
		icsk->icsk_backoff = 0;
		icsk->icsk_probes_tstamp = 0;
		inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
		/* Socket must be waked up by subsequent tcp_data_snd_check().
		 * This function is not for random using!
		 */
```

```c
	} else {
		unsigned long when = tcp_probe0_when(sk, TCP_RTO_MAX);

		when = tcp_clamp_probe0_to_user_timeout(sk, when);
		tcp_reset_xmit_timer(sk, ICSK_TIME_PROBE0, when, TCP_RTO_MAX);
	}
}

static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
{
	return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
		inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
}

/* Decide wheather to run the increase function of congestion control. */
static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
{
	/* If reordering is high then always grow cwnd whenever data is
	 * delivered regardless of its ordering. Otherwise stay conservative
	 * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
	 * new SACK or ECE mark may first advance cwnd here and later reduce
	 * cwnd in tcp_fastretrans_alert() based on more states.
	 */
	if (tcp_sk(sk)->reordering > sock_net(sk)->ipv4.sysctl_tcp_reordering)
		return flag & FLAG_FORWARD_PROGRESS;

	return flag & FLAG_DATA_ACKED;
}

/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
			     int flag, const struct rate_sample *rs)
{
	const struct inet_connection_sock *icsk = inet_csk(sk);

	if (icsk->icsk_ca_ops->cong_control) {
		icsk->icsk_ca_ops->cong_control(sk, rs);
		return;
	}

	if (tcp_in_cwnd_reduction(sk)) {
		/* Reduce cwnd if state mandates */
		tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
	} else if (tcp_may_raise_cwnd(sk, flag)) {
		/* Advance cwnd if state allows */
		tcp_cong_avoid(sk, ack, acked_sacked);
	}
	tcp_update_pacing_rate(sk);
}

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline bool tcp_may_update_window(const struct tcp_sock *tp,
					const u32 ack, const u32 ack_seq,
					const u32 nwin)
{
	return	after(ack, tp->snd_una) ||
		after(ack_seq, tp->snd_wl1) ||
		(ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
}

/* If we update tp->snd_una, also update tp->bytes_acked */
static void tcp_snd_una_update(struct tcp_sock *tp, u32 ack)
{
	u32 delta = ack - tp->snd_una;

	sock_owned_by_me((struct sock *)tp);
	tp->bytes_acked += delta;
```

```c
	WRITE_ONCE(tp->rcv_nxt, seq);
}

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
				 u32 ack_seq)
{
	struct tcp_sock *tp = tcp_sk(sk);
	int flag = 0;
	u32 nwin = ntohs(tcp_hdr(skb)->window);

	if (likely(!tcp_hdr(skb)->syn))
		nwin <<= tp->rx_opt.snd_wscale;

	if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
		flag |= FLAG_WIN_UPDATE;
		tcp_update_wl(tp, ack_seq);

		if (tp->snd_wnd != nwin) {
			tp->snd_wnd = nwin;

			/* Note, it is the only place, where
			 * fast path is recovered for sending TCP.
			 */
			tp->pred_flags = 0;
			tcp_fast_path_check(sk);

			if (!tcp_write_queue_empty(sk))
				tcp_slow_start_after_idle_check(sk);

			if (nwin > tp->max_window) {
				tp->max_window = nwin;
				tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
			}
		}
	}

	tcp_snd_una_update(tp, ack);

	return flag;
}

static bool __tcp_oow_rate_limited(struct net *net, int mib_idx,
				   u32 *last_oow_ack_time)
{
	if (*last_oow_ack_time) {
		s32 elapsed = (s32)(tcp_jiffies32 - *last_oow_ack_time);

		if (0 <= elapsed && elapsed < net->ipv4.sysctl_tcp_invalid_ratelimit) {
			NET_INC_STATS(net, mib_idx);
			return true;	/* rate-limited: don't send yet! */
		}
	}

	*last_oow_ack_time = tcp_jiffies32;

	return false;	/* not rate-limited: go ahead, send dupack now! */
}

/* Return true if we're currently rate-limiting out-of-window ACKs and
 * thus shouldn't send a dupack right now. We rate-limit dupacks in
 * response to out-of-window SYNs or ACKs to mitigate ACK loops or DoS
 * attacks that send repeated SYNs or ACKs for the same connection. To
 * do this, we do not send a duplicate SYNACK or ACK if the remote
 * endpoint is sending out-of-window SYNs or pure ACKs at a high rate.
 */
bool tcp_oow_rate_limited(struct net *net, const struct sk_buff *skb,
			  int mib_idx, u32 *last_oow_ack_time)
{
	/* Data packets without SYNs are not likely part of an ACK loop. */
	if ((TCP_SKB_CB(skb)->seq != TCP_SKB_CB(skb)->end_seq) &&
	    !tcp_hdr(skb)->syn)
```

Most congestion control code is boilerplate

```c
/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
                             int flag, const struct rate_sample *rs)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cong_control) {
                icsk->icsk_ca_ops->cong_control(sk, rs);
                return;
        }

        if (tcp_in_cwnd_reduction(sk)) {
                /* Reduce cwnd if state mandates */
                tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
        } else if (tcp_may_raise_cwnd(sk, flag)) {
                /* Advance cwnd if state allows */
                tcp_cong_avoid(sk, ack, acked_sacked);
        }
        tcp_update_pacing_rate(sk);
}
```

CUBIC

```c
static void cubictcp_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct bictcp *ca = inet_csk_ca(sk);

        if (!tcp_is_cwnd_limited(sk))
                return;

        if (tcp_in_slow_start(tp)) {
                if (hystart && after(ack, ca->end_seq))
                        bictcp_hystart_reset(sk);
                acked = tcp_slow_start(tp, acked);
                if (!acked)
                        return;
        }
        bictcp_update(ca, tp->snd_cwnd, acked);
        tcp_cong_avoid_ai(tp, ca->cnt, acked);
}
```

VEGAS

```c
static void tcp_vegas_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct vegas *vegas = inet_csk_ca(sk);

        if (!vegas->doing_vegas_now) {
                tcp_reno_cong_avoid(sk, ack, acked);
                return;
        }

        if (after(ack, vegas->beg_snd_nxt)) {
                /* Do the Vegas once-per-RTT cwnd adjustment. */

                /* Save the extent of the current window so we can use this
                 * at the end of the next RTT.
                 */
                vegas->beg_snd_nxt  = tp->snd_nxt;

                /* We do the Vegas calculations only if we got enough RTT
                 * samples that we can be reasonably sure that we got
                 * at least one RTT sample that wasn't from a delayed ACK.
                 * If we only had 2 samples total,
                 * then that means we're getting only 1 ACK per RTT, which
                 * means they're almost certainly delayed ACKs.
                 * If  we have 3 samples, we should be OK.
                 */

                if (vegas->cntRTT <= 2) {
                        /* We don't have enough RTT samples to do the Vegas
                         * calculation, so we'll behave like Reno.
                         */
                        tcp_reno_cong_avoid(sk, ack, acked);
                } else {
                        u32 rtt, diff;
                        u64 target_cwnd;

                        /* We have enough RTT samples, so, using the Vegas
                         * algorithm, we determine if we should increase or
                         * decrease cwnd, and by how much.
                         */
```

BBR

```c
static void bbr_main(struct sock *sk, const struct rate_sample *rs)
{
        struct bbr *bbr = inet_csk_ca(sk);
        u32 bw;

        bbr_update_model(sk, rs);

        bw = bbr_bw(sk);
        bbr_set_pacing_rate(sk, bw, bbr->pacing_gain);
        bbr_set_cwnd(sk, rs, rs->acked_sacked, bw, bbr->cwnd_gain);
}

static void bbr_init(struct sock *sk)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct bbr *bbr = inet_csk_ca(sk);

        bbr->prior_cwnd = 0;
        tp->snd_ssthresh = TCP_INFINITE_SSTHRESH;
        bbr->rtt_cnt = 0;
```

CCA code

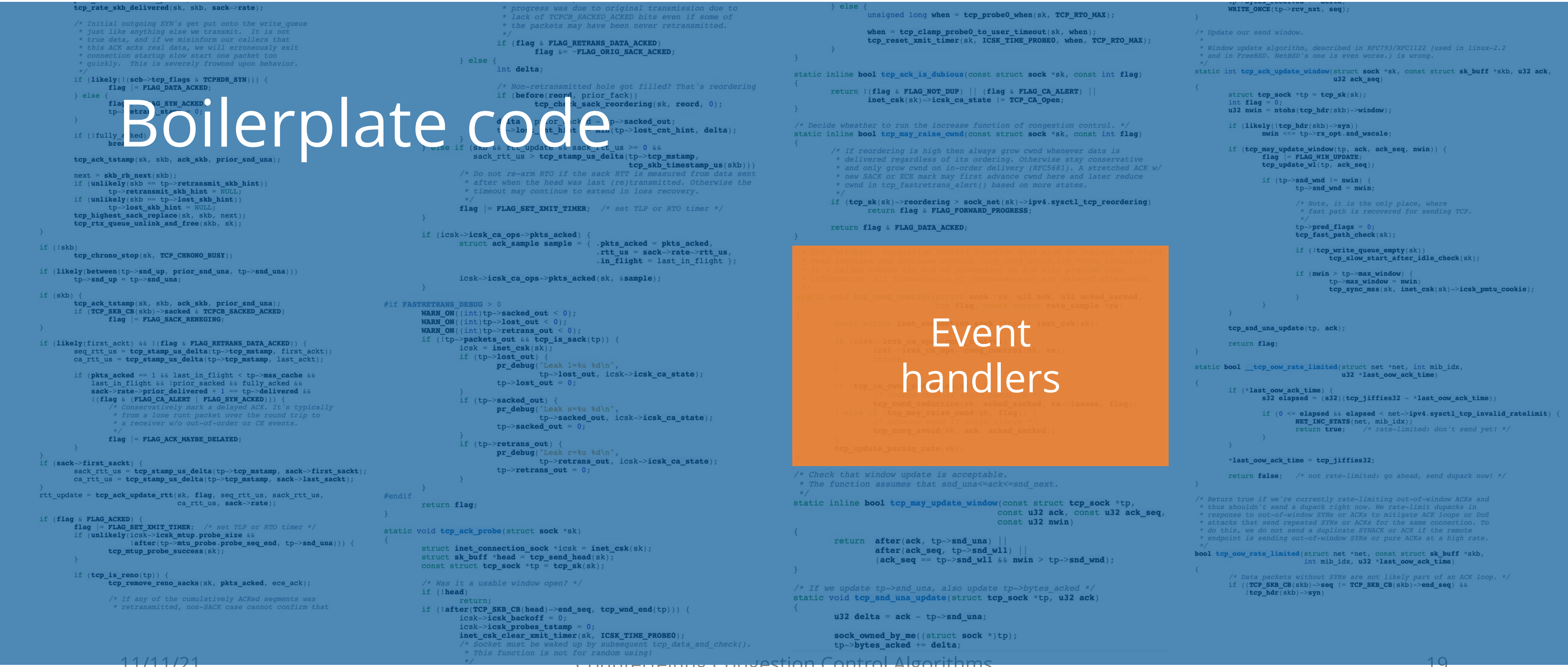
```c
        tcp_rate_skb_delivered(sk, skb, sack->rate);

        /* Initial outgoing SYN's get put onto the write_queue
         * just like anything else we transmit.  It is not
         * true data, and if we misinform our callers that
         * this ACK acks real data, we will erroneously exit
         * connection startup slow start one packet too
         * quickly.  This is severely frowned upon behavior.
         */
        if (likely(!(scb->tcp_flags & TCPHDR_SYN))) {
                flag |= FLAG_DATA_ACKED;
        } else {
                flag |= FLAG_SYN_ACKED;
                tp->retrans_stamp = 0;
        }

        if (!fully_acked)
                break;

        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);

        next = skb_rb_next(skb);
        if (unlikely(skb == tp->retransmit_skb_hint))
                tp->retransmit_skb_hint = NULL;
        if (unlikely(skb == tp->lost_skb_hint))
                tp->lost_skb_hint = NULL;
        tcp_highest_sack_replace(sk, skb, next);
        tcp_rtx_queue_unlink_and_free(skb, sk);
}

if (!skb)
        tcp_chrono_stop(sk, TCP_CHRONO_BUSY);

if (likely(between(tp->snd_up, prior_snd_una, tp->snd_una)))
        tp->snd_up = tp->snd_una;

if (skb) {
        tcp_ack_tstamp(sk, skb, ack_skb, prior_snd_una);
        if (TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED)
                flag |= FLAG_SACK_RENEGING;
}

if (likely(first_ackt) && !(flag & FLAG_RETRANS_DATA_ACKED)) {
        seq_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, first_ackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, last_ackt);

        if (pkts_acked == 1 && last_in_flight < tp->mss_cache &&
            last_in_flight && !prior_sacked && fully_acked &&
            sack->rate->prior_delivered + 1 == tp->delivered &&
            !(flag & (FLAG_CA_ALERT | FLAG_SYN_ACKED))) {
                /* Conservatively mark a delayed ACK. It's typically
                 * from a lone runt packet over the round trip to
                 * a receiver w/o out-of-order or CE events.
                 */
                flag |= FLAG_ACK_MAYBE_DELAYED;
        }
}
if (sack->first_sackt) {
        sack_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->first_sackt);
        ca_rtt_us = tcp_stamp_us_delta(tp->tcp_mstamp, sack->last_sackt);
}
rtt_update = tcp_ack_update_rtt(sk, flag, seq_rtt_us, sack_rtt_us,
                                ca_rtt_us, sack->rate);

if (flag & FLAG_ACKED) {
        flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        if (unlikely(icsk->icsk_mtup.probe_size &&
                     !after(tp->mtu_probe.probe_seq_end, tp->snd_una))) {
                tcp_mtup_probe_success(sk);
        }

        if (tcp_is_reno(tp)) {
                tcp_remove_reno_sacks(sk, pkts_acked, ece_ack);

                /* If any of the cumulatively ACKed segments was
                 * retransmitted, non-SACK case cannot confirm that
```

```c
                         * progress was due to original transmission due to
                         * lack of TCPCB_SACKED_ACKED bits even if some of
                         * the packets may have been never retransmitted.
                         */
                        if (flag & FLAG_RETRANS_DATA_ACKED)
                                flag &= ~FLAG_ORIG_SACK_ACKED;
                } else {
                        int delta;

                        /* Non-retransmitted hole got filled? That's reordering */
                        if (before(reord, prior_fack))
                                tcp_check_sack_reordering(sk, reord, 0);

                        delta = prior_sacked - tp->sacked_out;
                        tp->lost_cnt_hint -= min(tp->lost_cnt_hint, delta);
                }
        } else if (skb && rtt_update && sack_rtt_us >= 0 &&
                   sack_rtt_us > tcp_stamp_us_delta(tp->tcp_mstamp,
                                                    tcp_skb_timestamp_us(skb))) {
                /* Do not re-arm RTO if the sack RTT is measured from data sent
                 * after when the head was last (re)transmitted. Otherwise the
                 * timeout may continue to extend in loss recovery.
                 */
                flag |= FLAG_SET_XMIT_TIMER;  /* set TLP or RTO timer */
        }

        if (icsk->icsk_ca_ops->pkts_acked) {
                struct ack_sample sample = { .pkts_acked = pkts_acked,
                                             .rtt_us = sack->rate->rtt_us,
                                             .in_flight = last_in_flight };

                icsk->icsk_ca_ops->pkts_acked(sk, &sample);
        }

#if FASTRETRANS_DEBUG > 0
        WARN_ON((int)tp->sacked_out < 0);
        WARN_ON((int)tp->lost_out < 0);
        WARN_ON((int)tp->retrans_out < 0);
        if (!tp->packets_out && tcp_is_sack(tp)) {
                icsk = inet_csk(sk);
                if (tp->lost_out) {
                        pr_debug("Leak l=%u %d\n",
                                 tp->lost_out, icsk->icsk_ca_state);
                        tp->lost_out = 0;
                }
                if (tp->sacked_out) {
                        pr_debug("Leak s=%u %d\n",
                                 tp->sacked_out, icsk->icsk_ca_state);
                        tp->sacked_out = 0;
                }
                if (tp->retrans_out) {
                        pr_debug("Leak r=%u %d\n",
                                 tp->retrans_out, icsk->icsk_ca_state);
                        tp->retrans_out = 0;
                }
        }
#endif
        return flag;
}

static void tcp_ack_probe(struct sock *sk)
{
        struct inet_connection_sock *icsk = inet_csk(sk);
        struct sk_buff *head = tcp_send_head(sk);
        const struct tcp_sock *tp = tcp_sk(sk);

        /* Was it a usable window open? */
        if (!head)
                return;
        if (!after(TCP_SKB_CB(head)->end_seq, tcp_wnd_end(tp))) {
                icsk->icsk_backoff = 0;
                icsk->icsk_probes_tstamp = 0;
                inet_csk_clear_xmit_timer(sk, ICSK_TIME_PROBE0);
                /* Socket must be waked up by subsequent tcp_data_snd_check().
                 * This function is not for random using!
                 */
```

```c
        } else {
                unsigned long when = tcp_probe0_when(sk, TCP_RTO_MAX);

                when = tcp_clamp_probe0_to_user_timeout(sk, when);
                tcp_reset_xmit_timer(sk, ICSK_TIME_PROBE0, when, TCP_RTO_MAX);
        }
}

static inline bool tcp_ack_is_dubious(const struct sock *sk, const int flag)
{
        return !(flag & FLAG_NOT_DUP) || (flag & FLAG_CA_ALERT) ||
                inet_csk(sk)->icsk_ca_state != TCP_CA_Open;
}

/* Decide wheather to run the increase function of congestion control. */
static inline bool tcp_may_raise_cwnd(const struct sock *sk, const int flag)
{
        /* If reordering is high then always grow cwnd whenever data is
         * delivered regardless of its ordering. Otherwise stay conservative
         * and only grow cwnd on in-order delivery (RFC5681). A stretched ACK w/
         * new SACK or ECE mark may first advance cwnd here and later reduce
         * cwnd in tcp_fastretrans_alert() based on more states.
         */
        if (tcp_sk(sk)->reordering > sock_net(sk)->ipv4.sysctl_tcp_reordering)
                return flag & FLAG_FORWARD_PROGRESS;

        return flag & FLAG_DATA_ACKED;
}

/* The "ultimate" congestion control function that aims to replace the rigid
 * cwnd increase and decrease control (tcp_cong_avoid,tcp_*cwnd_reduction).
 * It's called toward the end of processing an ACK with precise rate
 * information. All transmission or retransmission are delayed afterwards.
 */
static void tcp_cong_control(struct sock *sk, u32 ack, u32 acked_sacked,
                             int flag, const struct rate_sample *rs)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cong_control) {
                icsk->icsk_ca_ops->cong_control(sk, rs);
                return;
        }

        if (tcp_in_cwnd_reduction(sk)) {
                /* Reduce cwnd if state mandates */
                tcp_cwnd_reduction(sk, acked_sacked, rs->losses, flag);
        } else if (tcp_may_raise_cwnd(sk, flag)) {
                /* Advance cwnd if state allows */
                tcp_cong_avoid(sk, ack, acked_sacked);
        }
        tcp_update_pacing_rate(sk);
}

/* Check that window update is acceptable.
 * The function assumes that snd_una<=ack<=snd_next.
 */
static inline bool tcp_may_update_window(const struct tcp_sock *tp,
                                         const u32 ack, const u32 ack_seq,
                                         const u32 nwin)
{
        return  after(ack, tp->snd_una) ||
                after(ack_seq, tp->snd_wl1) ||
                (ack_seq == tp->snd_wl1 && nwin > tp->snd_wnd);
}

/* If we update tp->snd_una, also update tp->bytes_acked */
static void tcp_snd_una_update(struct tcp_sock *tp, u32 ack)
{
        u32 delta = ack - tp->snd_una;

        sock_owned_by_me((struct sock *)tp);
        tp->bytes_acked += delta;
```

```c
        WRITE_ONCE(tp->rcv_nxt, seq);
}

/* Update our send window.
 *
 * Window update algorithm, described in RFC793/RFC1122 (used in linux-2.2
 * and in FreeBSD. NetBSD's one is even worse.) is wrong.
 */
static int tcp_ack_update_window(struct sock *sk, const struct sk_buff *skb, u32 ack,
                                 u32 ack_seq)
{
        struct tcp_sock *tp = tcp_sk(sk);
        int flag = 0;
        u32 nwin = ntohs(tcp_hdr(skb)->window);

        if (likely(!tcp_hdr(skb)->syn))
                nwin <<= tp->rx_opt.snd_wscale;

        if (tcp_may_update_window(tp, ack, ack_seq, nwin)) {
                flag |= FLAG_WIN_UPDATE;
                tcp_update_wl(tp, ack_seq);

                if (tp->snd_wnd != nwin) {
                        tp->snd_wnd = nwin;

                        /* Note, it is the only place, where
                         * fast path is recovered for sending TCP.
                         */
                        tp->pred_flags = 0;
                        tcp_fast_path_check(sk);

                        if (!tcp_write_queue_empty(sk))
                                tcp_slow_start_after_idle_check(sk);

                        if (nwin > tp->max_window) {
                                tp->max_window = nwin;
                                tcp_sync_mss(sk, inet_csk(sk)->icsk_pmtu_cookie);
                        }
                }
        }

        tcp_snd_una_update(tp, ack);

        return flag;
}

static bool __tcp_oow_rate_limited(struct net *net, int mib_idx,
                                   u32 *last_oow_ack_time)
{
        if (*last_oow_ack_time) {
                s32 elapsed = (s32)(tcp_jiffies32 - *last_oow_ack_time);

                if (0 <= elapsed && elapsed < net->ipv4.sysctl_tcp_invalid_ratelimit) {
                        NET_INC_STATS(net, mib_idx);
                        return true;    /* rate-limited: don't send yet! */
                }
        }

        *last_oow_ack_time = tcp_jiffies32;

        return false;   /* not rate-limited: go ahead, send dupack now! */
}

/* Return true if we're currently rate-limiting out-of-window ACKs and
 * thus shouldn't send a dupack right now. We rate-limit dupacks in
 * response to out-of-window SYNs or ACKs to mitigate ACK loops or DoS
 * attacks that send repeated SYNs or ACKs for the same connection. To
 * do this, we do not send a duplicate SYNACK or ACK if the remote
 * endpoint is sending out-of-window SYNs or pure ACKs at a high rate.
 */
bool tcp_oow_rate_limited(struct net *net, const struct sk_buff *skb,
                          int mib_idx, u32 *last_oow_ack_time)
{
        /* Data packets without SYNs are not likely part of an ACK loop. */
        if ((TCP_SKB_CB(skb)->seq != TCP_SKB_CB(skb)->end_seq) &&
            !tcp_hdr(skb)->syn)
```

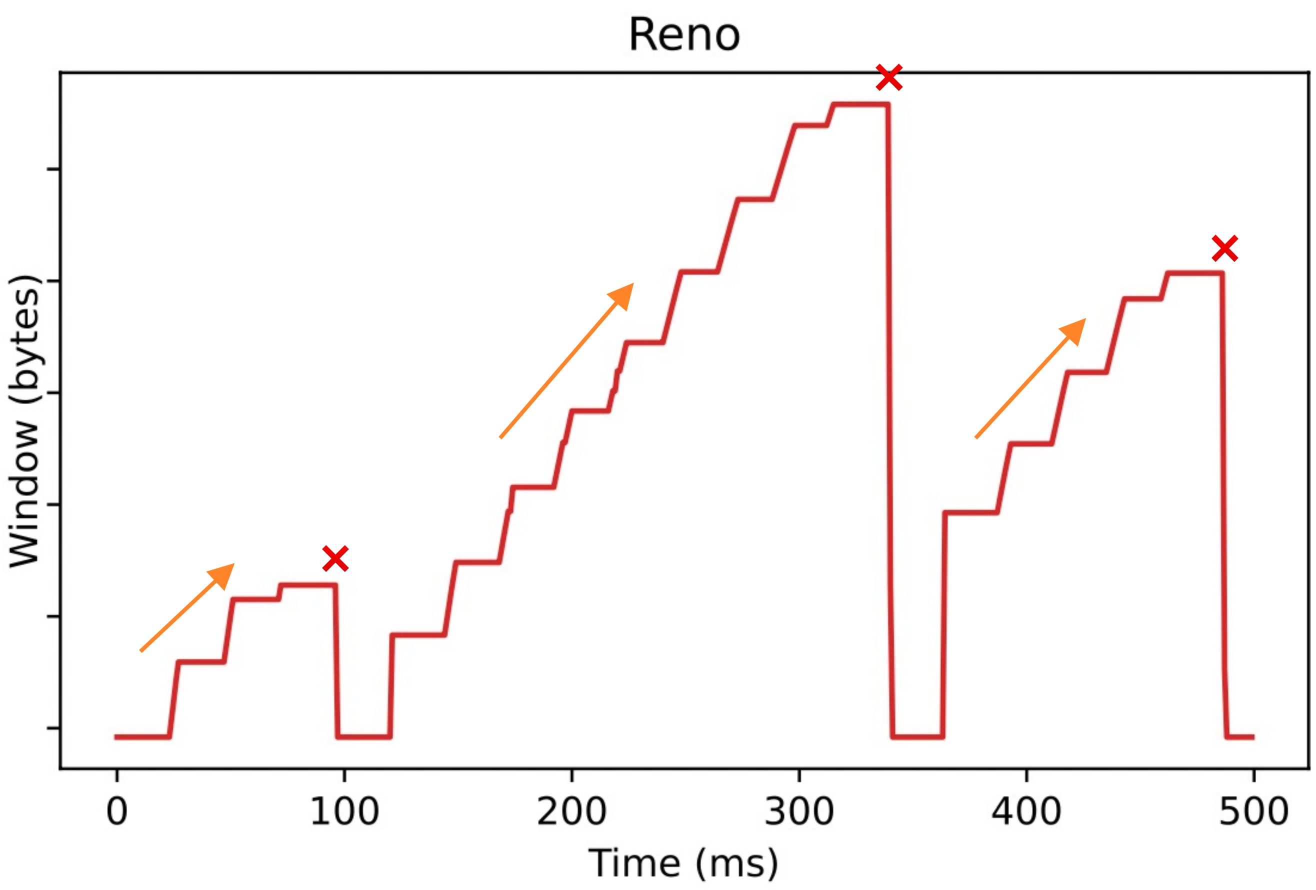
CCA code



CCA Synthesis

CCA Synthesis

Our 1st goal: synthesize a very basic version of Reno

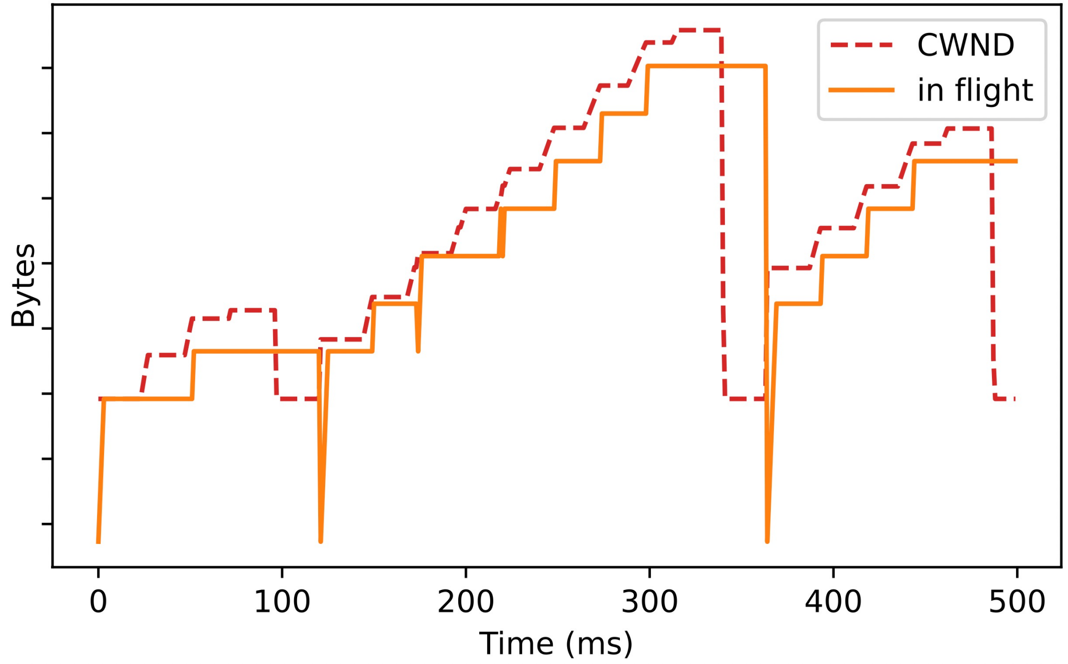


Two event handlers:

- ***win-ack*** - updates CWND when there is an ACK
- ***win-timeout*** - updates CWND there is a timeout

CCA Synthesis

Some inputs are unknown



? ? ✓ ✓ ?

h(CWND, ACKs, sent) = new CWND

Time	ACKs	Sent
1	-	1
2	-	1461
3	-	2921
4	-	4381
5	-	-
6	-	-
7	-	-
8	1461	5841
9	-	-
10	4381	7301
11	5841	8761
12	-	10221
	…	

Each timestep takes as input the previous timestep's output

$$h(CWND_0, ACKs_0, sent_0) = CWND_1$$
$$h(CWND_1, ACKs_1, sent_1) = CWND_2$$
$$h(CWND_2, ACKs_2, sent_2) = CWND_3$$
$$h(CWND_3, ACKs_3, sent_3) = CWND_4$$
$$h(CWND_4, ACKs_4, sent_4) = CWND_5$$

...

$$CWND_n = h(\ldots h(h(h(CWND_0, ACKs_0, sent_0), ACKs_1, sent_1), ACKs_2, sent_2)\ldots)$$

Time	ACKs	Sent
1	-	1
2	-	1461
3	-	2921
4	-	4381
5	-	-
6	-	-
7	-	-
8	1461	5841
9	-	-
10	4381	7301
11	5841	8761
12	-	10221
...		

Naïve approach

Naïve search

Candidate h functions:

- *win-ack = CWND + AKD * CWND /* x_1
- *win-timeout = CWND /* x_2

Trace a | Trace b | Trace c | Trace d

Are there win-ack and win-timeout handlers that fit the traces?

Z3

Naïve search

Candidate h functions:

- *win-ack = CWND + AKD * CWND / MSS*
- *win-timeout = CWND /* x_2

Trace a | Trace b

Trace c | Trace d

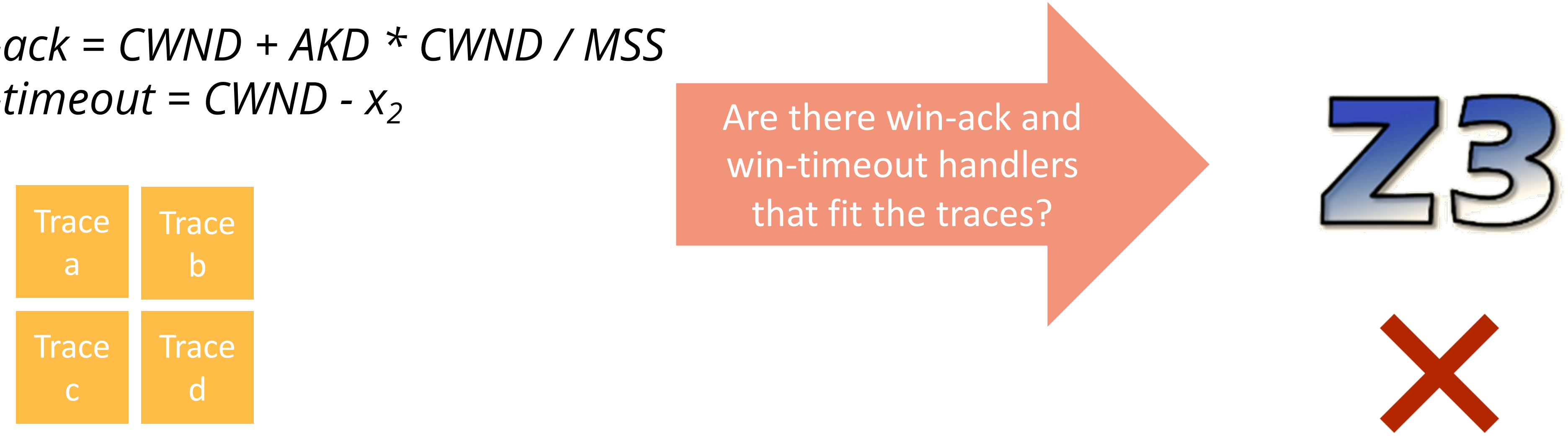
Are there win-ack and win-timeout handlers that fit the traces?

Z3

Naïve search

Candidate h functions:

- *win-ack = CWND + AKD * CWND / MSS*
- *win-timeout = CWND -* x_2



Naïve search

Candidate h functions:

- *win-ack = CWND + AKD - MSS*
- *win-timeout = CWND -* x_2

Trace a

Trace b

Trace c

Trace d

Are there win-ack and win-timeout handlers that fit the traces?

Naïve search

Candidate h functions:

- *win-ack = CWND + AKD * MSS / CWND*
- *win-timeout = CWND -* x_2

Trace a

Trace b

Trace c

Trace d

Are there win-ack and win-timeout handlers that fit the traces?

Naïve search

Candidate h functions:

- *win-ack = CWND + AKD * MSS / CWND*
- *win-timeout = CWND * w_0*

Trace a

Trace b

Trace c

Trace d

Are there win-ack and win-timeout handlers that fit the traces?

Z3

The search space is very, very large

~20,000 win-ack handlers

x

~20,000 win-timeout handlers

=

several hundred million possible CCAs

but we cannot do several hundred million solver calls.

CCA synthesis is challenging

Traditional synthesis:

$f(1, 2, 3) = 7$

$f(2, 3, 4) = 8$

$f(5, 3, 5) = 8$

$f(2, 4, 5) = 12$

Synthesizer

$f(x, y, z) = x + y*z/x$

CCA synthesis:

$CWND_n = h(\ldots h(h(h(CWND_0, ACKs, sent_0), ACKs, sent_1), ACKs, sent_2)\ldots)$

Synthesizer

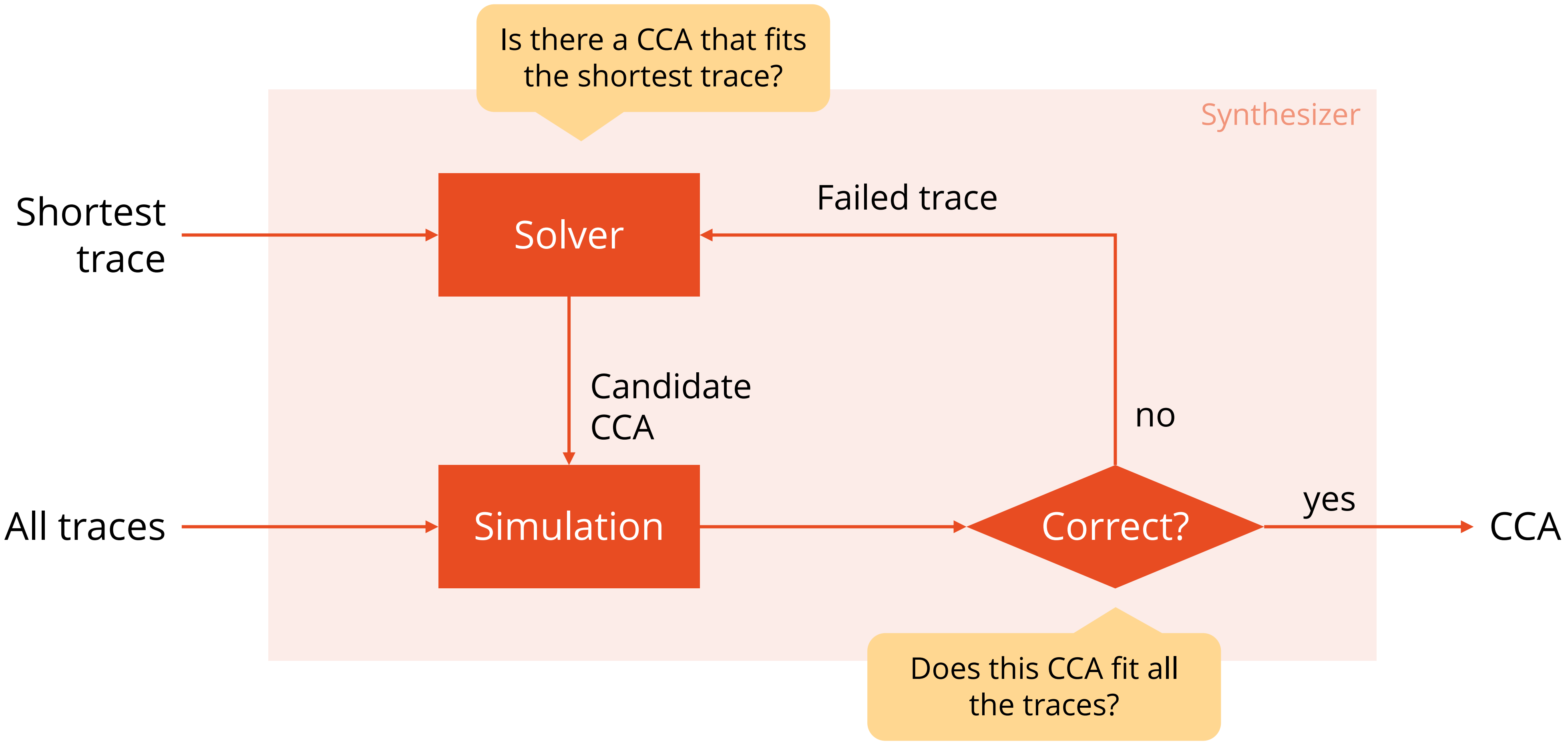
?

Our synthesizer: Mister 880

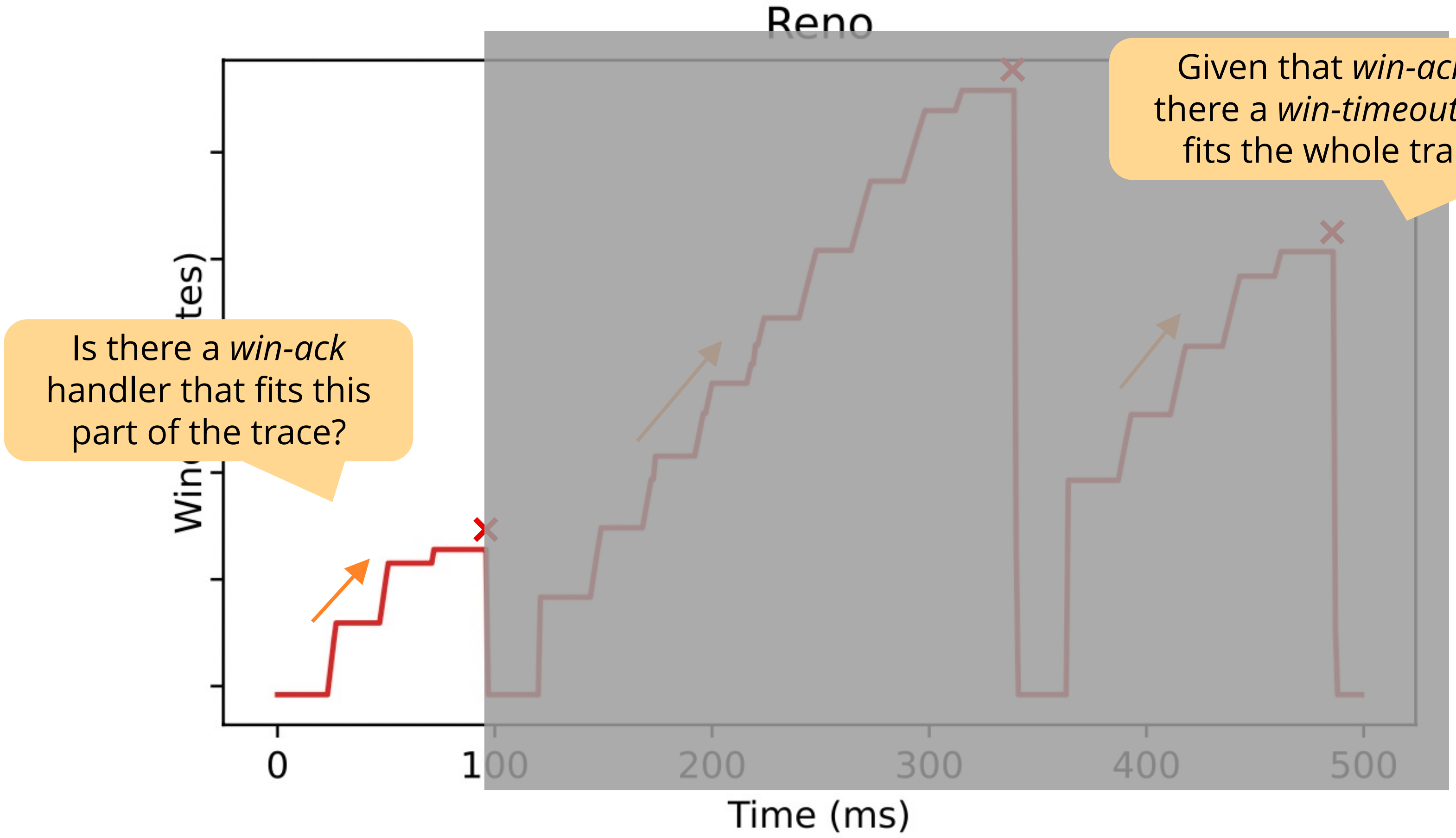
2 main goals:

- Make simpler solver calls
- Decrease search space

Start with shortest trace



Synthesize one handler at a time



***win-ack*:**
Updates CWND when there is an ACK

***win-timeout*:**
Updates CWND when there is a timeout

Do not consider functions that cannot be handlers

$$\underset{\text{bytes}^2}{CWND} = \underset{\text{bytes}}{CWND} * \underset{\text{bytes}}{AKD}$$

The result is not in bytes.

$$CWND = CWND / AKD * MSS$$

will never increase CWND.

Reduces search space by ~80%

Mister 880 divides the search into smaller, easier problems

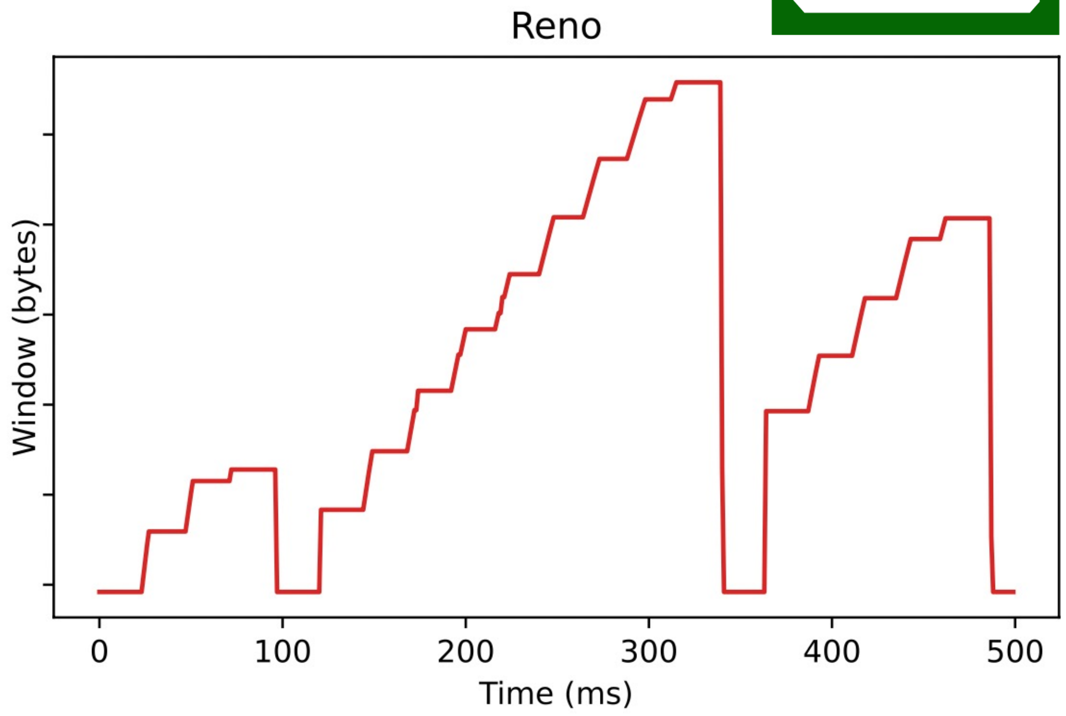
Main ideas:

- Start with shortest trace ✓simpler solver calls
- Synthesize one handler at a time ✓simpler solver calls
- Domain-specific knowledge ✓decrease search space

Simplified Reno

*win-ack(CWND, AKD, MSS) = CWND + AKD * MSS / CWND*

win-timeout(CWND, w_0*) =* w_0



Naïve approach without our optimizations did not finish.

Mister 880 synthesized in **13 minutes** on a 2015 MacBook Pro.

Counterfeiting Congestion Control Algorithms

Future directions:

- How can we work from Internet traces?
- How can we synthesize more complex CCAs?

Margarida Ferreira
margarida@cmu.edu