L4S and Prague update

draft-briscoe-iccrg-prague-congestion-control (needs update)

Many contributors in Open-Source repositories:

L4STeam/linux: Kernel tree with TCP-Prague and DualPI2

L4STeam/udp_prague: UDP-Prague CC object and examples

Koen De Schepper
(koen.de_schepper@nokia-bell-labs.com)

Chia-Yu Chang
(chia-yu.chang@nokia-bell-labs.com)

Linux Mainlining L4S

Started August 2024 by Chia-Yu

Date	DualPI2 sched	DualPI2 tc iproute2	ACC_ECN 1/3	ACC_ECN 2/3	ACC_ECN 3/3	VirtIO spec	VirtIO driver	TCP-Prague tcp	TCP-Prague ss iproute2
Internal review	44 kernel patches 2 iproute2 patches 6 iterations								
Starting assumption	1 patch	1 patch	40 kernel patches			-	-	3 patches	1 patch
Patches	6	3	14 → 11 preparation	14 protocol	14 special cases	2	2	4	1
Iterations	v26	v12	v7	v16	v5	v10	v0	v0	v0
Net-next/ Linux-next	**July 2025**	**July 2025**	**March 2025**	**Sept 2025**					
mainline	**v6.17**	**v6.17**	**v6.15** **Not usable**	**(v6.18?)** **Not usable**	Activation of ACC_ECN	Updates in comments	Optimizes hypervisors	TCP-Prague	Stats for TCP-Prague
Status	**Success**	**Success**	**Success** 3 patches moved to 2/3	**Success** 1 patch moved to TCP-Prague	Ongoing: GRO packetdrill tests created	Blocked by ACC_ECN 3	Waiting for VirtIO Spec	Waiting for ACC_ECN	Waiting for TCP-Prague tcp

Links to available Linux kernels

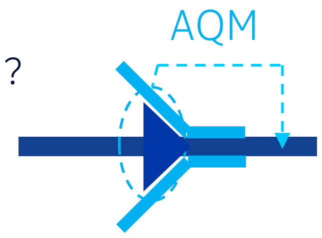
For TCP-Prague and ACC_ECN

- Linux 5.15 and Linux 6.1 is NOT actively maintained
- Linux 6.6
  - Linux stable: L4STeam/linux at l4steam-6.6.y
  - WSL2-Linux: minuscat/linux-msft-wsl-6.6.y
  - RPI Linux: minuscat/rpi-6.6.y
- Linux 6.12
  - Linux stable: L4STeam/linux at l4steam-6.12.y
  - Net-dev
  - Linux net-next: L4STeam/linux-net-next

Network improvements

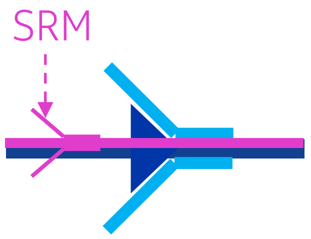
Controlling rate without queue with ARM: Active Rate Management

Prague packets don't need to be delayed to rate control, so why build a queue and do AQM?

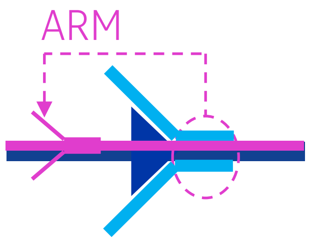


Rate can directly be controlled with marking policers instead of by shapers or schedulers

Static Rate Management (SRM) uses a static marking policer



Active Rate Management (ARM) can use a dynamic marking policer driven by a policy that can be aligned with scheduled flows



Rate Management has consequences

Impact on L4S CCs

Latencies can be very low, and stay low under max load so these Prague requirements are very important:

- RTT independence for low RTTs <25ms
- Pacing below minimum window of 2

  → CC should solve this

Less important are slow start overshoot

- if enough headroom, it does not create latency, just a short rate overshoot
- Still creates latency if only flow on an L4S Step or Slope AQM

  → Network can solve this, not sure if CC can without growing…

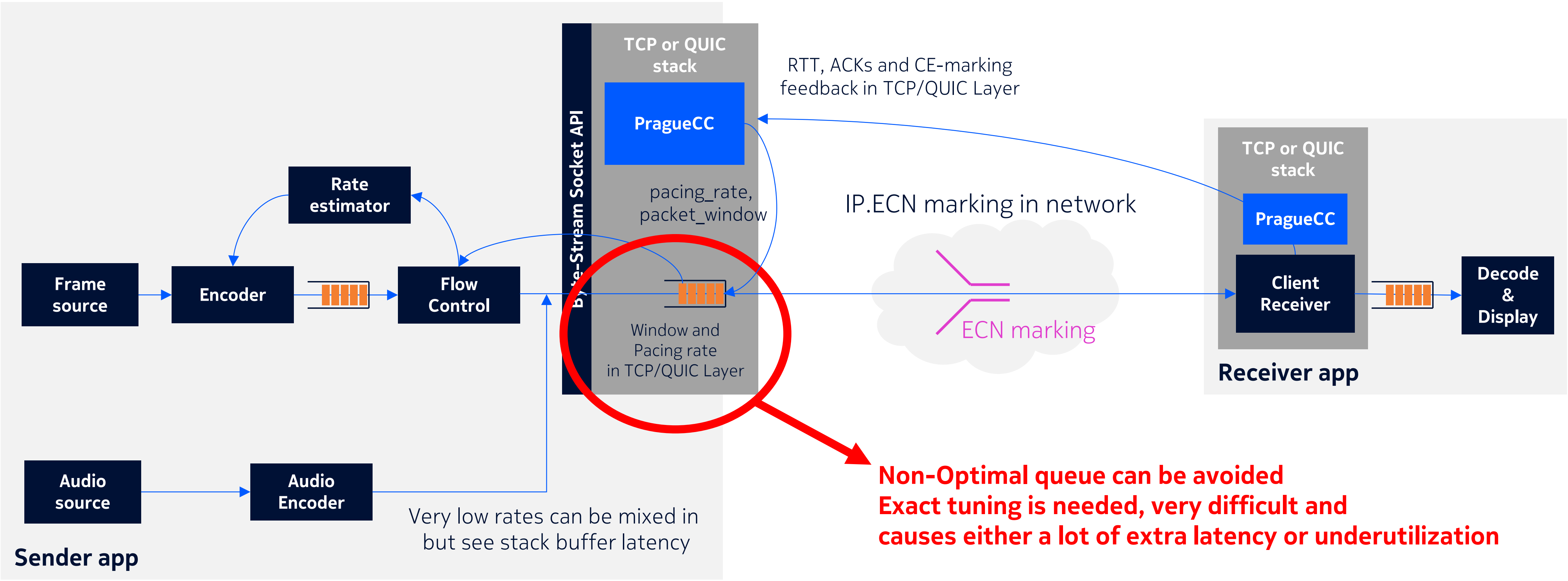
PragueCC for interactive apps

GitHub repository: L4STeam/udp_prague

- Same as TCP-Prague congestion control but nicely wrapped in an object that can be used in UDP-based applications
- Single source C++ prague_cc.h/cpp files define the PragueCC object with minimal dependencies
- Platform independent code (Windows, Linux, Android, FreeBSD, Apple, ...)
- Allows evolution of Prague without impacting the API (API can still evolve for now)
- Performs both Sender (Data) and Receiver (ACK) functions, supporting bidirectional data transfers
- Allows directly controlling application data generation rate and non-blocking delivery
- Example load generators present for sender and receiver (either run in client or server mode)
- Iperf2 integration at: Iperf 2 / Code / udp_prague, and ns-3 coming soon (currently in a private CableLabs git)

PragueCC with TCP or QUIC stacks

Stacks can use PragueCC object, but need non-optimal flow control

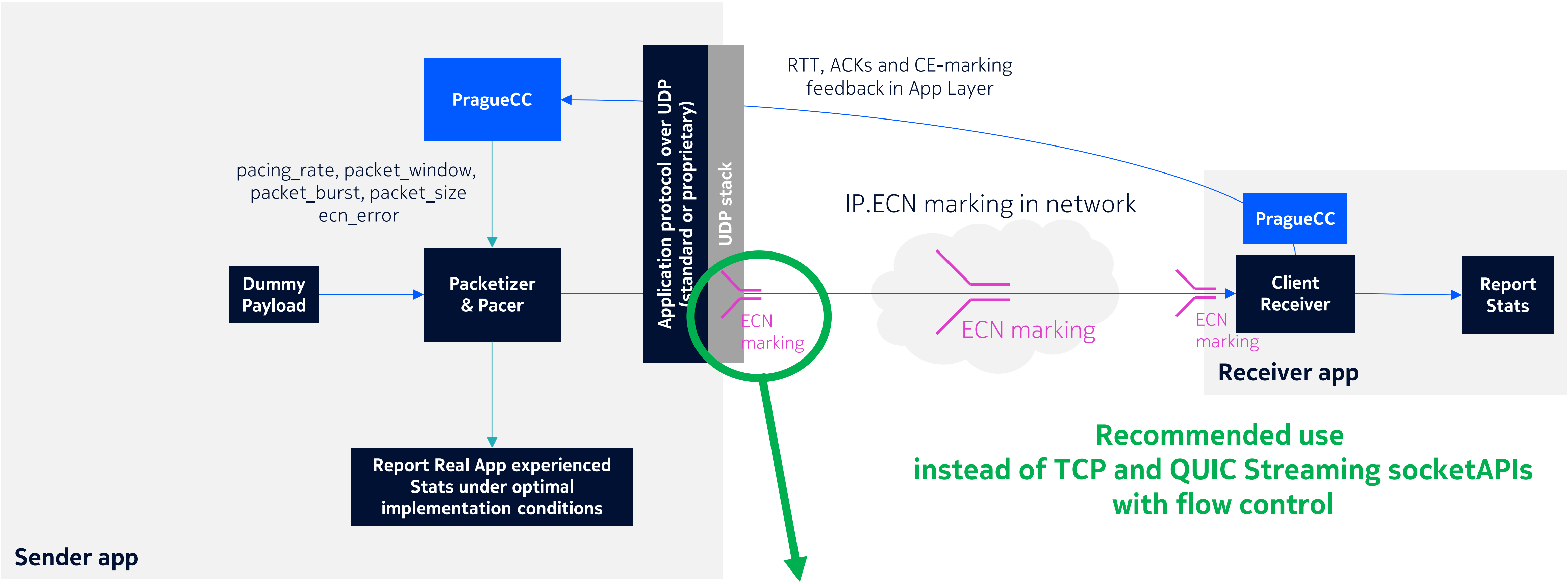


Non-Optimal queue can be avoided
Exact tuning is needed, very difficult and
causes either a lot of extra latency or underutilization

Users of L4S apps and measurement tools (iperf, RPM) feel and measure mainly this buffer

No-Queue use of PragueCC in UDP-Prague, iperf2 and ns3

Load generator tools can use PragueCC without other latencies and can report exact Stats

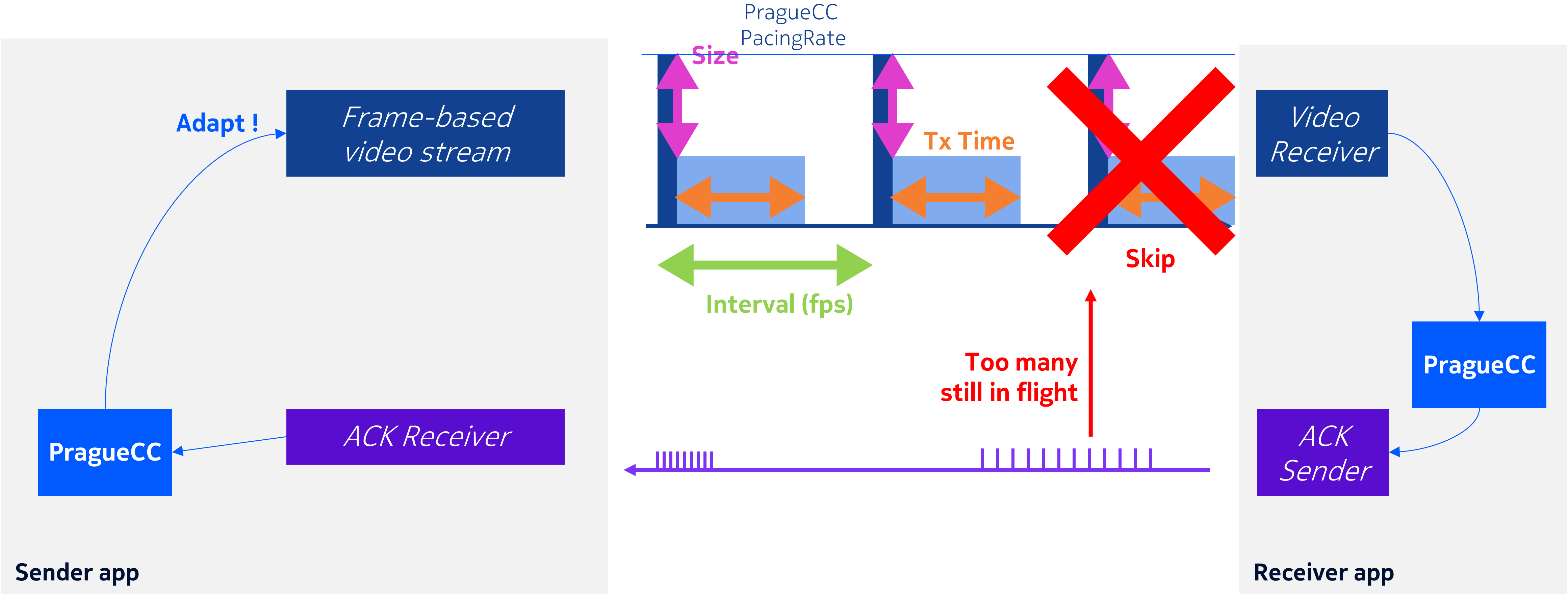


Recommended use instead of TCP and QUIC Streaming socketAPIs with flow control

Also, local NICs should mark instead of flow control

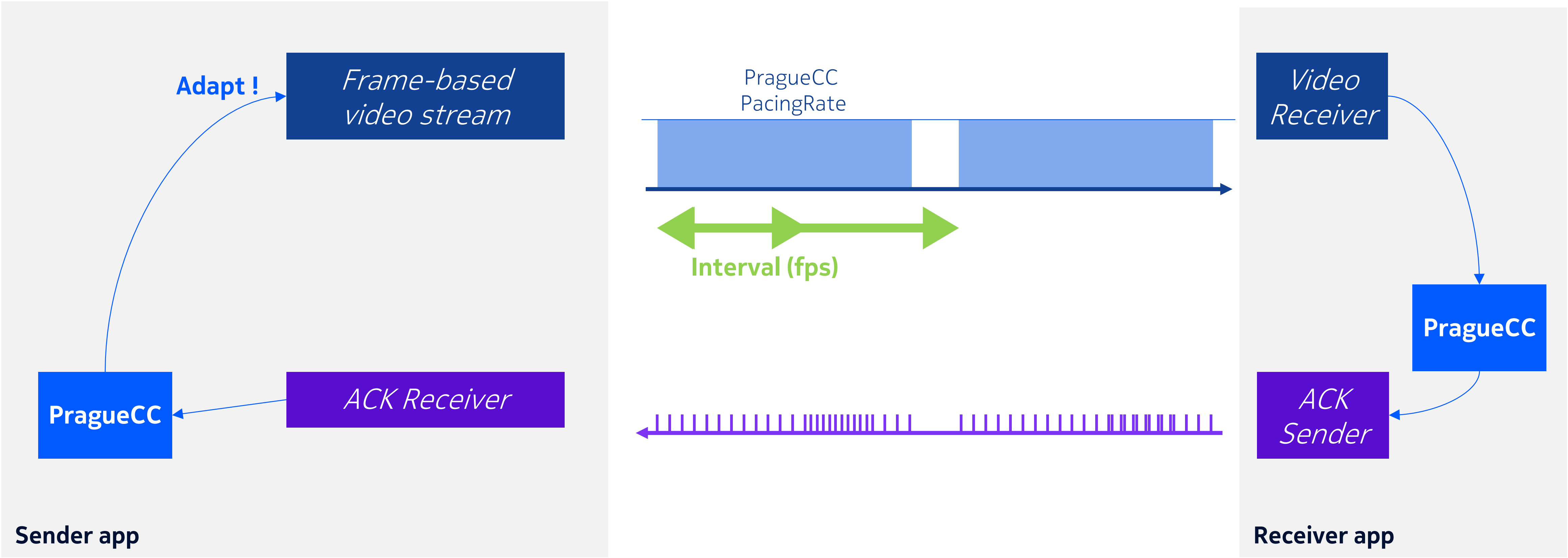
Video Traffic Type Adaptation

Many degrees of freedom for the app to adapt the **Frame**



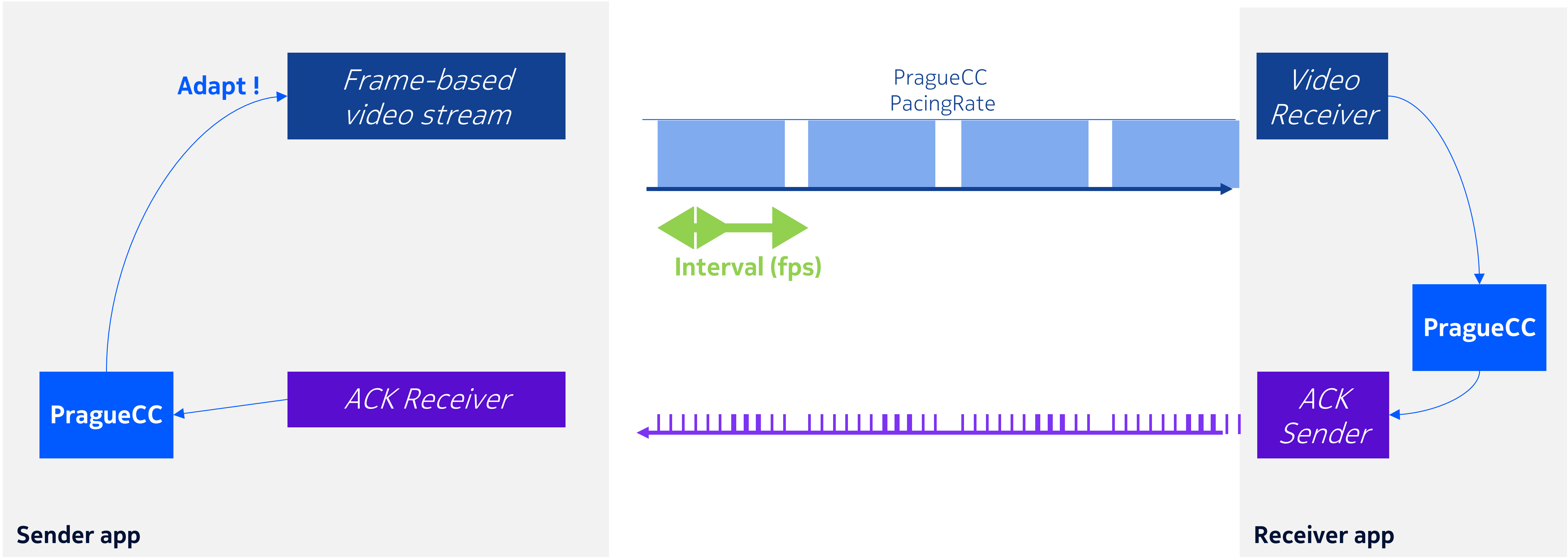
Video Traffic Type Adaptation

Many degrees of freedom for the app to adapt the **Frame**



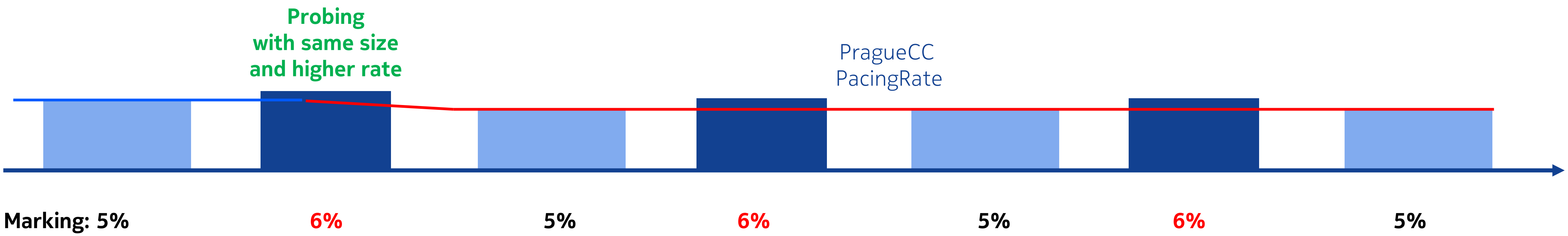
Video Traffic Type Adaptation

Many degrees of freedom for the app to adapt the **Frame**

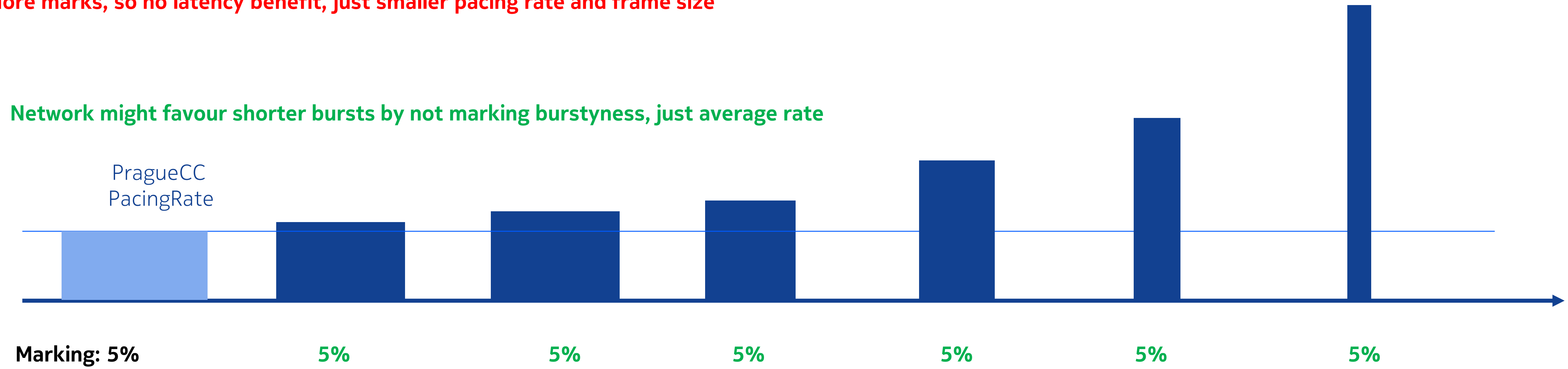


Video Traffic Type Adaptation

App can explore what lower transmission time gives: WIN-WIN if NW allows



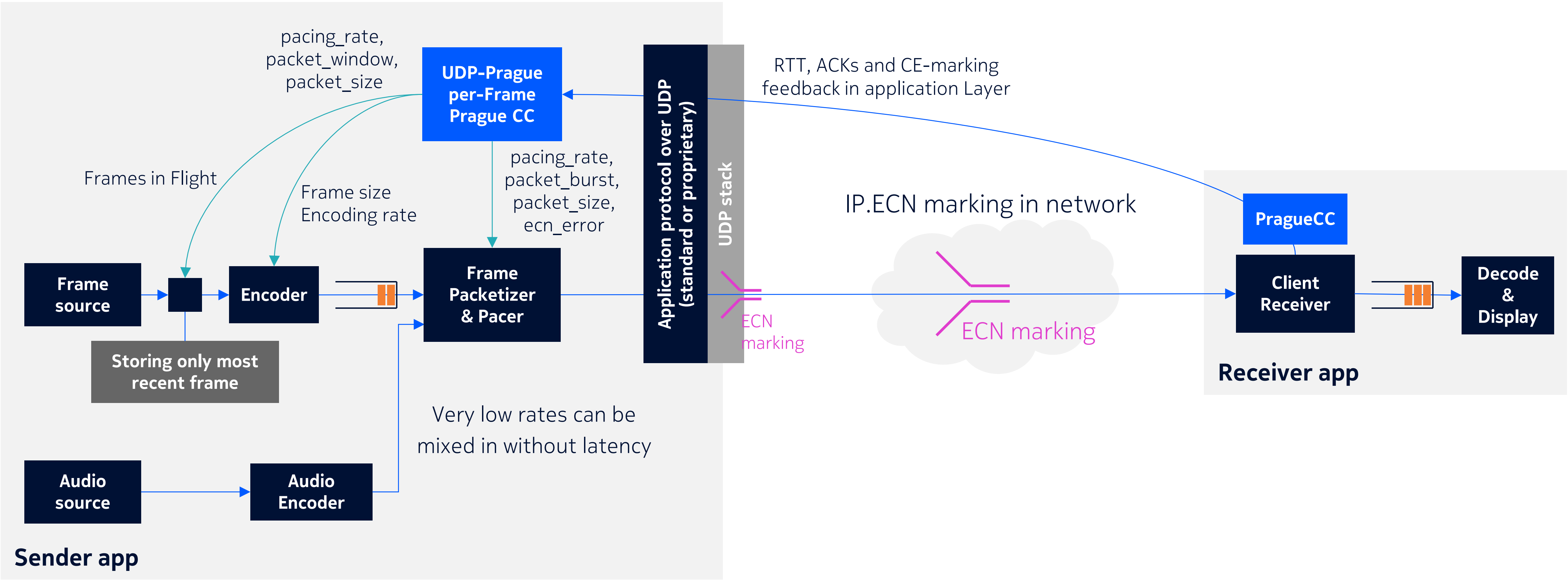
More marks, so no latency benefit, just smaller pacing rate and frame size



The App detects equal marks when pacing frames faster with same average rate, so lower user latency

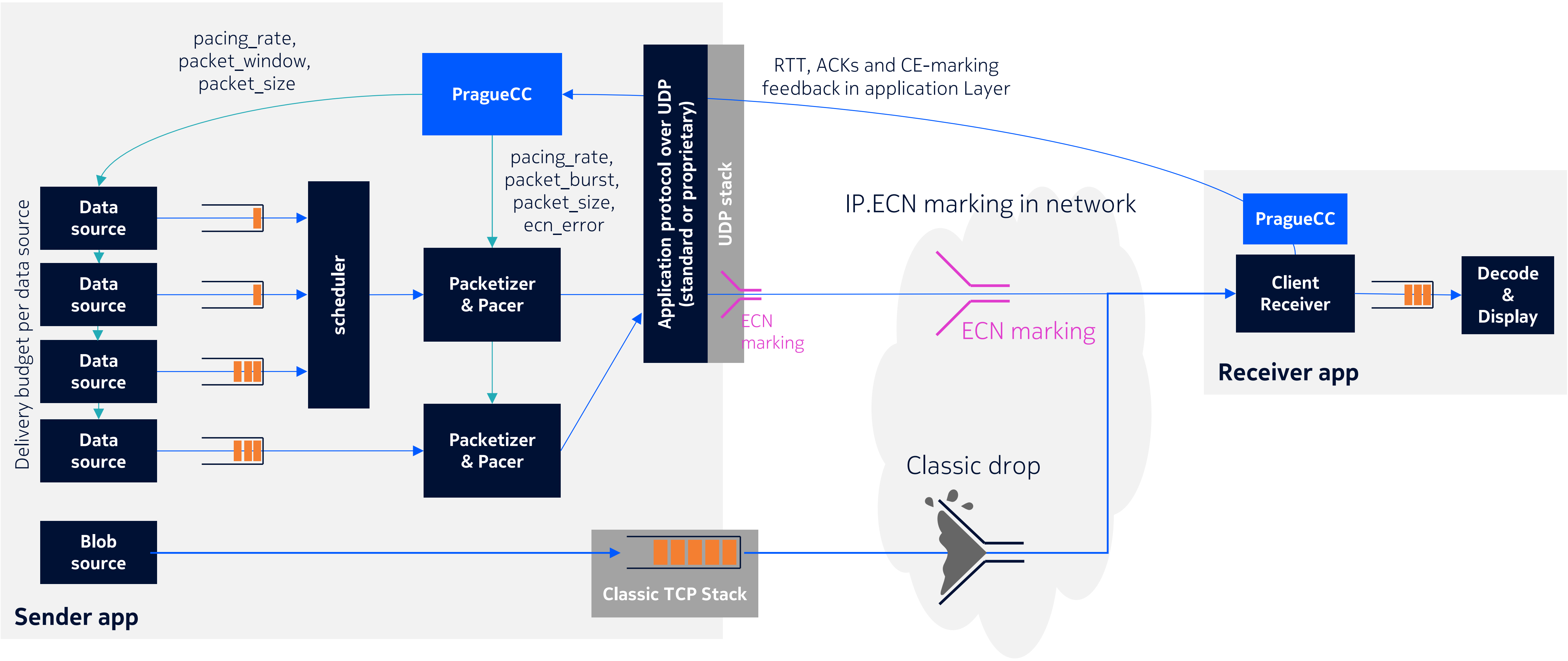
Real-Time app optimization with PragueCC

PragueCC uses ECN feedback directly to determine the next frame and its size



Real-Time app optimization with PragueCC

PragueCC uses ECN feedback directly to control the data source(s)



PragueCC object API and use

Event based: OnDataReceived

Common

For ACKing data

For SENDing data

```
struct message_t {
    time_tp timestamp;      // timestamp from peer
    time_tp echoed_timestamp;  // echoed_timestamp can be used to calculate the RTT

    count_tp packets_received; // echoed_packet counter
    count_tp packets_CE;       // echoed CE counter
    count_tp packets_lost;     // echoed lost counter
    bool error_L4S;            // receiver found a bleached/error ECN; stop using L4S_id on the sending packets!

    count_tp seq_nr;           // packet sequence number, should start with 1 and increase monotonic with packets sent
    // payload can come here...
};
```

event for ACKing data

```
void OnDataReceived(data_msg)
{
    // Pass the relevant data to the PragueCC object:
    pragueCC.PacketReceived(data_msg.timestamp, data_msg.echoed_timestamp);
    pragueCC.DataReceivedSequence(rcv_ecn, data_msg.seq_nr);

    // Return a corresponding acknowledge message
    pragueCC.GetTimeInfo(ack_msg.timestamp, ack_msg.echoed_timestamp, snd_ecn); // get timestamps and ecn bits to use
    pragueCC.GetACKInfo(ack_msg.packets_received, ack_msg.packets_CE, ack_msg.packets_lost, ack_msg.error_L4S);
    sendtoecn(ack_msg, snd_ecn, client); // if snd_ecn changed, it will set the new bits in the socket option

    // Process payload here
    ...
}
```

PragueCC object API and use

Event based: OnAckReceived, OnRetransmissionTimeOut, OnPaceTimeout

```
void OnAckReceived() {
    pragueCC.PacketReceived(ack_msg.timestamp, ack_msg.echoed_timestamp);
    pragueCC.ACKReceived(ack_msg.packets_received, ack_msg.packets_CE, ack_msg.packets_lost, seqnr, ack_msg.error_L4S, inflight);
    OnPaceTimeOut(); // check if we can send a packet
}

void OnPaceTimeout() {
    pragueCC.GetCCInfo(pacing_rate, packet_window, packet_burst, packet_size);
    count_tp inburst = 0;
    time_tp timeout = 0;
    time_tp startSend = 0;
    while ((inflight < packet_window) && (inburst < packet_burst) && (nextSend <= pragueCC.Now())) {
        pragueCC.GetTimeInfo(data_msg.timestamp, data_msg.echoed_timestamp, snd_ecn);
        if (startSend == 0) startSend = now;
        data_msg.seq_nr = seqnr; // also add payload here …
        sendto_ecn(data_msg, snd_ecn);
        inburst++;
        inflight++;
        seqnr++;
    }
    if (startSend != 0) { // we have send some data and are burst or window blocked
        nextSend = startSend + packet_size * inburst * 1000000 / pacing_rate;
        ScheduleEventAt(OnPaceTimeout, nextSend); // for the case we are (or become) not window blocked
    }
    if (inflight >= packet_window)
        ScheduleEventAt(OnRetransmissionTimeout, retransmission_timeout + pragueCC.Now());
}

void OnRetransmissionTimeOut() {
    pragueCC.ResetCCInfo(); // timeout, reset state
    OnPaceTimeOut(); // we will be able to send a packet again
}
```

Prague features

Applicable and aligned for both PragueCC and TCP-Prague

Previous available features:

- Marking based RTT-independence: down scaling the increase with (25ms / max(25ms, sRTT))^2 factor
- Minimum window and minimum rate scales down to 100kbps on any RTT <25 and lower on RTT>25ms
- Pacing per 250µs

New features:

- Rate or Window update
- +/-3% Sub-RTT pacing
- PRR-like reduction (TCP-Prague → PragueCC)

Still under test in PragueCC:

- Faster up to speed with AI to MI Growth switch (PragueCC)
- Loss based RTT-independence (PragueCC)

Not yet decided:

- Delay based?

Rate or Window update

If RTT is very small and minimum W=2, small jitter, becomes very high relative to base

If the Window is gradually updated, but the srtt varies a factor of 100, the pacing rate will also

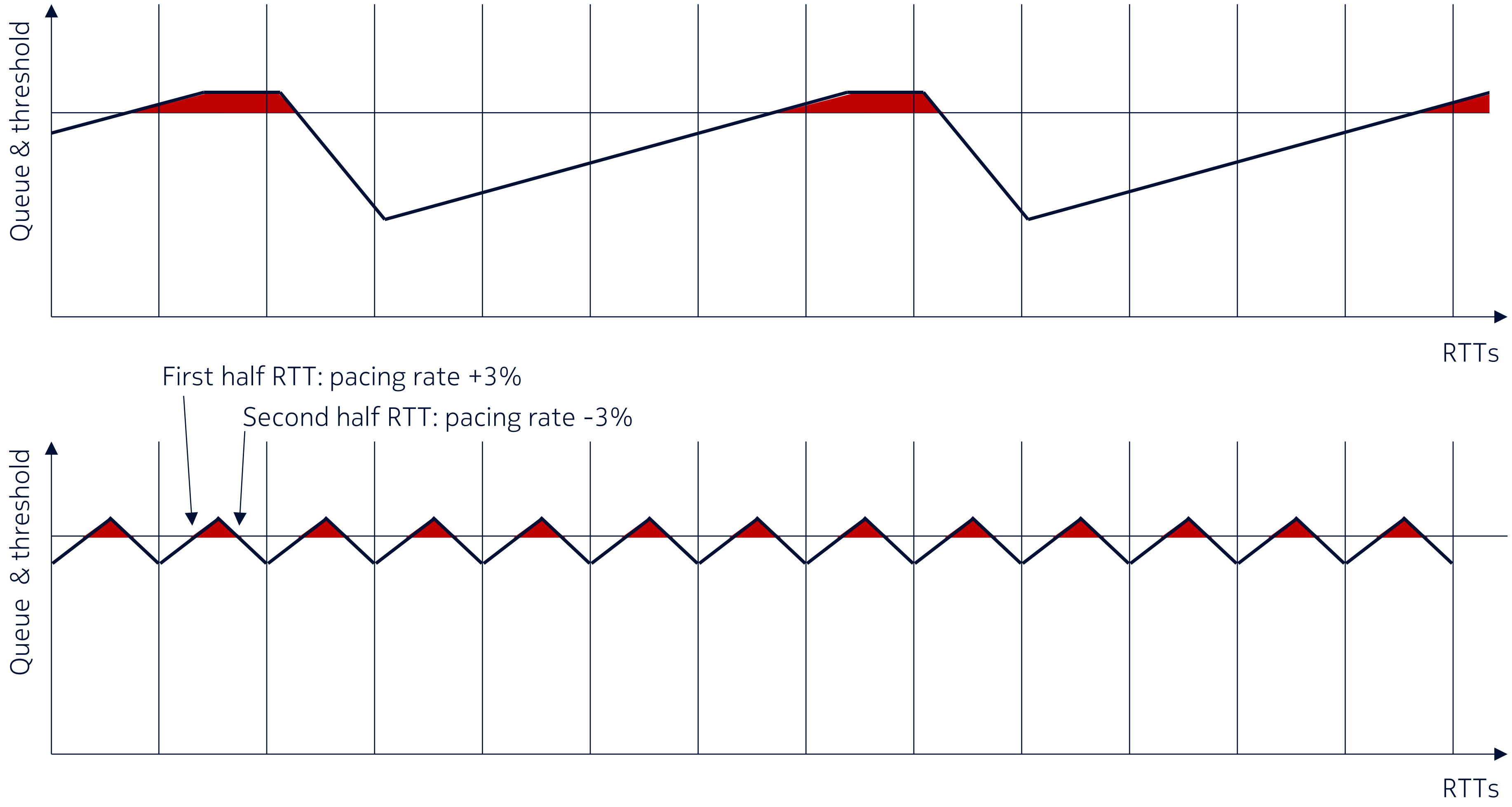
In that case, better update the pacing rate and derive the fractional window (in case it would become > 2p)

Switch to "update the pacing rate" happens if SRTT is below 2ms or below the pacing interval

```
// select the rate- or window-based update, but keep the rate stable on switching
   time_tp pacing_interval = m_packet_size * 1000000 / m_pacing_rate;     //calculate the max expected 'rtt' from pacing
   if ((m_srtt <= 2000) || (m_srtt <= pacing_interval))
       m_cca_mode = cca_prague_rate;                        // keep rate stable when large dip in m_srtt
   else {
       if (m_cca_mode == cca_prague_rate)
           m_fractional_window = m_srtt * m_pacing_rate;  // keep rate stable when large jump in m_srtt
       m_cca_mode = cca_prague_win;
   }
```

+/-3% Sub-RTT pacing

Easy fix, improves stable marking on a Step AQM (on a very stable link or simulation)



ON/OFF marking on a Step AQM

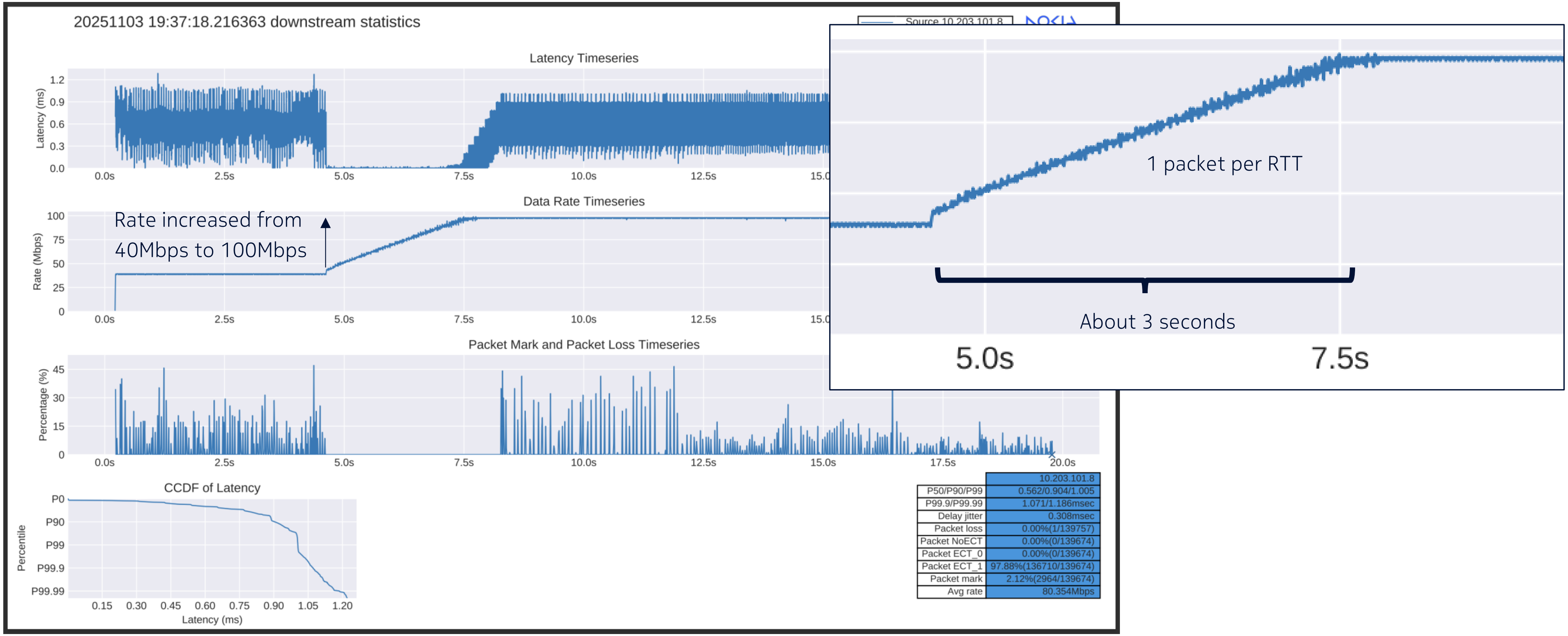
Stable per RTT marks (and rate)

Replaces the need for a Slope AQM

Improves RTT fairness

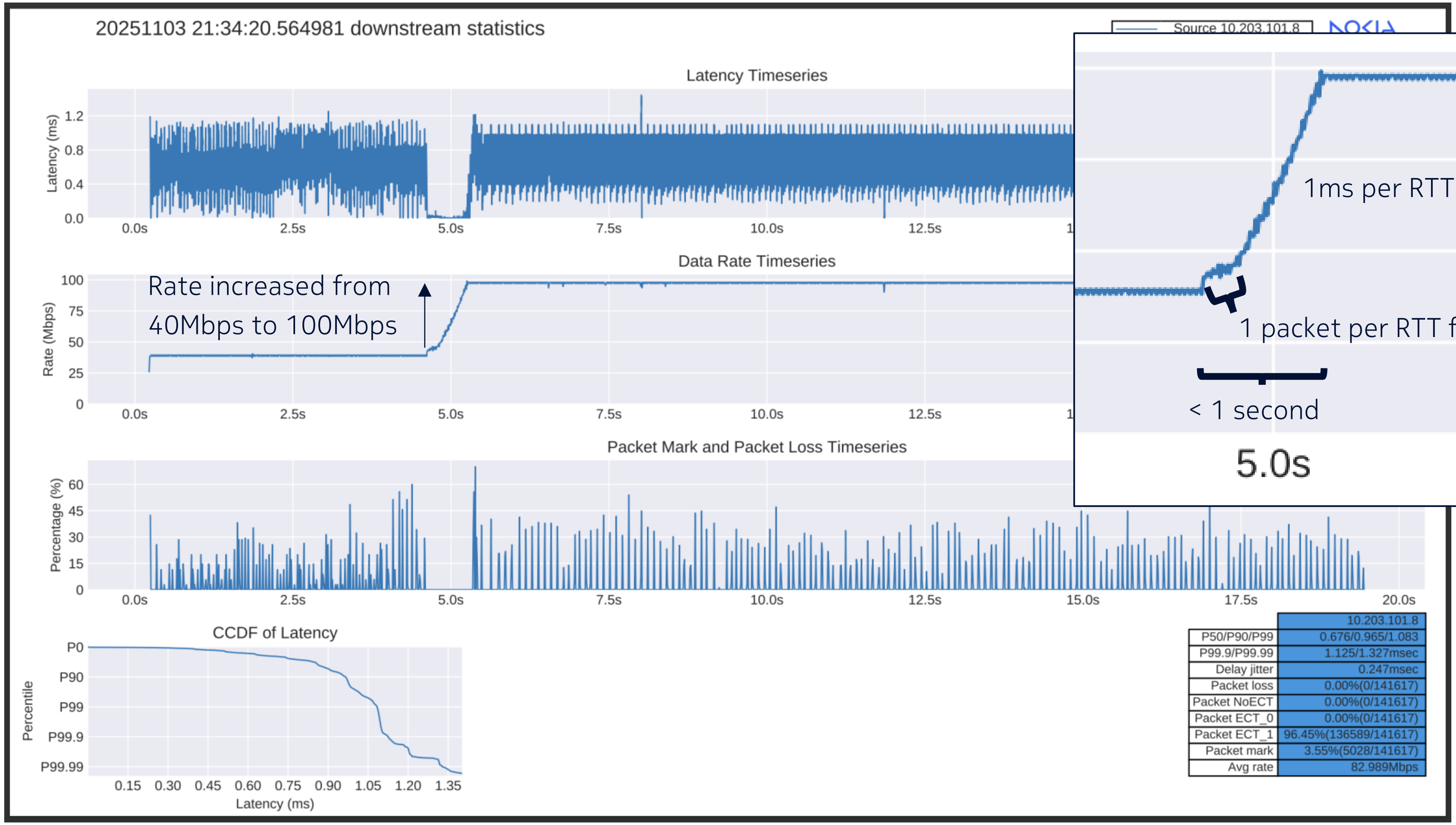
If bandwidth becomes available, Reno increase is too slow

If you don't get marks after the expected RTT, switch to MI with 1ms per RTT

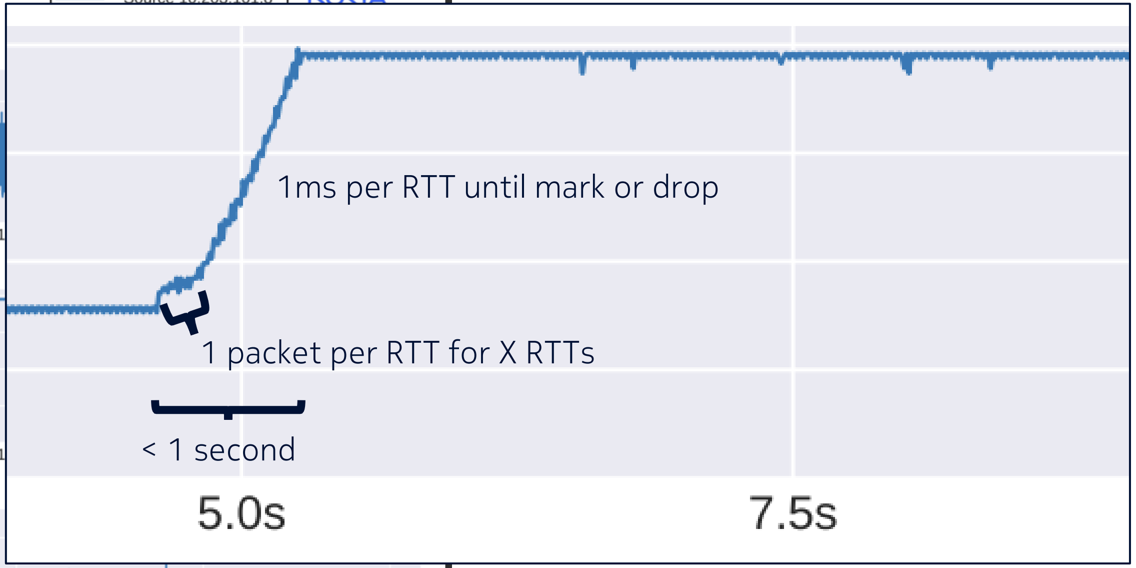


AI to MI Growth switch

If you don't get marks after the expected RTT, switch to MI with 1ms per RTT



	10.203.101.8
P50/P90/P99	0.676/0.965/1.083
P99.9/P99.99	1.125/1.327msec
Delay jitter	0.247msec
Packet loss	0.00%(0/141617)
Packet NoECT	0.00%(0/141617)
Packet ECT_0	0.00%(0/141617)
Packet ECT_1	96.45%(136589/141617)
Packet mark	3.55%(5028/141617)
Avg rate	82.989Mbps



Currently X = 7 + pacing_rate/15.36Mbps
Based on theoretical model of ON/OFF marking
Alternatives to be explored

count_tp rtts_to_growth = m_pacing_rate / RATE_STEP + MIN_STEP;

AI to MI Growth switch

Further observations

Main goal:

- Much faster getting new capacity

Bonus:

- Faster convergence: low-rate flows go faster in MI, high-rate flows don’t get the chance
- Not always active
- De-activated when +/-3% triggers marks faster than worst case ON/OFF equation

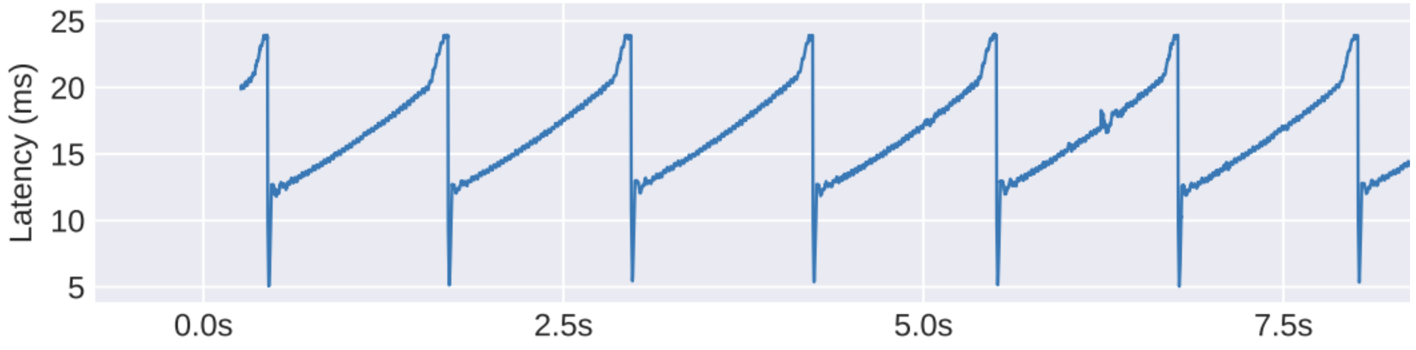
Downside:

- During conversion frequent 1ms overshoots over the marking threshold (not on SRM/ARM)

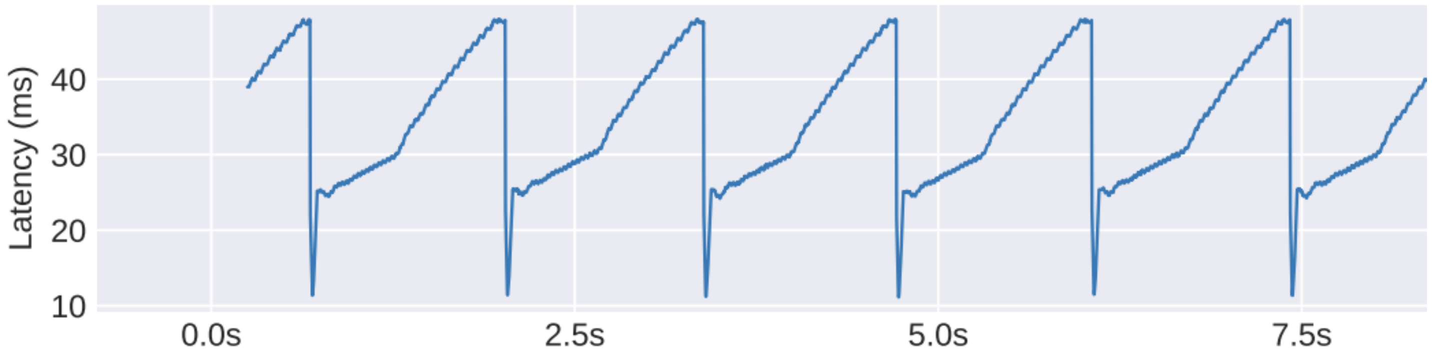
Loss based RTT-independent instead of Cubic

Same for loss: If no loss when expected, switch to MI with 1ms per RTT

Expected or wanted → an opportunity to make Prague better compete for higher RTTs



25ms taildrop queue



50ms taildrop queue

count_tp rtts_to_growth = m_pacing_rate / 2 / m_max_packet_size * REF_RTT / m_vrtt * REF_RTT / 1000000;

UDP-Prague and PragueCC

GitHub repository: L4STeam/udp_prague

Done also
- Overwritable virtual Now() method to support simulators or non-standard monotonic clocks
- PRR-like reduction

Todo:
- app limit window growth limit,
- RT-Prague API for Video?,
- Delay based policies?
- More examples
- ...

Let us know if you are using/evaluating this and have suggestions on API or CC improvements

Questions?

draft-briscoe-iccrg-prague-congestion-control

Many contributors in Open-Source repositories:

L4STeam/linux: Kernel tree with TCP-Prague and DualPI2

L4STeam/udp_prague: UDP-Prague CC object and examples (still under construction)

Koen De Schepper
(koen.de_schepper@nokia-bell-labs.com)

Chia-Yu Chang
(chia-yu.chang@nokia-bell-labs.com)