Network Working Group
Internet-Draft
Intended status: Informational
Expires: 30 August 2026

C. Huitema
Private Octopus Inc.
S. Nandakumar
C. Jennings
Cisco

26 February 2026

Design of Christian's Congestion Control Code (C4)
draft-huitema-ccwg-c4-design-03

Abstract

Christian's Congestion Control Code is a new congestion control algorithm designed to support Real-Time applications such as Media over QUIC. It is designed to drive towards low delays, with good support for the "application limited" behavior frequently found when using variable rate encoding, and with fast reaction to congestion to avoid the "priority inversion" happening when congestion control overestimates the available capacity. It pays special attention to the high jitter conditions encountered in Wi-Fi networks. The design emphasizes simplicity and avoids making too many assumption about the "model" of the network. The main control variables are the estimate of the data rate and of the maximum path delay in the absence of queues.

Table of Contents

1. Introduction

Christian's Congestion Control Code (C4) is a new congestion control algorithm designed to support Real-Time multimedia applications, specifically multimedia applications using QUIC [RFC9000] and the Media over QUIC transport [I-D.ietf-moq-transport]. These applications require low delays, and often exhibit a variable data rate as they alternate high bandwidth requirements when sending reference frames and lower bandwith requirements when sending differential frames. We translate that into 3 main goals:

* Drive towards low delays (see Section 2),

* Support "application limited" behavior (see Section 7),

* React quickly to changing network conditions (see Section 5).

The design of C4 is inspired by our experience using different congestion control algorithms for QUIC, notably Cubic [RFC9438], Hystart [HyStart], and BBR [I-D.ietf-ccwg-bbr], as well as the study of delay-oriented algorithms such as TCP Vegas [TCP-Vegas] and LEDBAT [RFC6817]. In addition, we wanted to keep the algorithm simple and easy to implement.

C4 assumes that the transport stack is capable of signaling to the congestion algorithms events such as acknowledgements, RTT measurements, ECN signals or the detection of packet losses. It also assumes that the congestion algorithm controls the transport stack by setting the congestion window (CWND) and the pacing rate.

C4 tracks the state of the network by keeping a small set of variables, the main ones being the "nominal rate", the "nominal max RTT", and the current state of the algorithm. The details on using and tracking the min RTT are discussed in Section 2.

The nominal rate is the pacing rate corresponding to the most recent estimate of the bandwidth available to the connection. The nominal max RTT is the best estimate of the maximum RTT that can occur on the network in the absence of queues. When we do not observe delay jitter, this coincides with the min RTT. In the presence of jitter, it should be the sum of the min RTT and the maximum jitter. C4 will compute a pacing rate as the nominal rate multiplied by a coefficient that depends on the state of the protocol, and set the CWND for the path to the product of that pacing rate by the max RTT. The design of these mechanisms is discussed in Section 5.

2. Studying the reaction to delays

The current design of C4 is the result of a series of experiments. Our initial design was to monitor delays and react to dellay increases in much the same way as congestion control algorithms like TCP Vegas or LEDBAT:

* monitor the current RTT and the min RTT

* if the current RTT sample exceed the min RTT by more than a preset margin, treat that as a congestion signal.

The "preset margin" is set by default to 10 ms in TCP Vegas and LEDBAT. That was adequate when these algorithms were designed, but it can be considered excessive in high speed low latency networks. For the initial C4 design, we set it to the lowest of 1/8th of the min RTT and 25ms.

The min RTT itself is measured over time. The detection of congestion by comparing delays to min RTT plus margin works well, except in two conditions:

* if the C4 connection is competing with a another connection that does not react to delay variations, such as a connection using Cubic,

* if the network exhibits a lot of latency jitter, as happens on some Wi-Fi networks.

We also know that if several connection using delay-based algorithms compete, the competition is only fair if they all have the same estimate of the min RTT. We handle that by using a "periodic slow down" mechanism.

2.1. Managing Competition with Loss Based Algorithms

Competition between Cubic and a delay based algorithm leads to Cubic consuming all the bandwidth and the delay based connection starving. This phenomenon force TCP Vegas to only be deployed in controlled environments, in which it does not have to compete with TCP Reno [RFC6582] or Cubic.

We handled this competition issue by using a simple detection algorithm. If C4 detected competition with a loss based algorithm, it switched to a "pig war" mode and stopped reacting to changes in delays -- it would instead only react to packet losses and ECN signals. In that mode, we used another algorithm to detect when the competition has ceased, and switch back to the delay responsive mode.

In our initial deployments, we detected competition when delay based congestion notifications leads to CWND and rate reduction for more than 3 consecutive RTT. The assumption is that if the competition reacted to delays variations, it would have reacted to the delay increases before 3 RTT. However, that simple test caused many "false positive" detections.

We refined this test to start the pig war if we observed 4 consecutive delay-based rate reductions and the nominal CWND was less than half the max nominal CWND observed since the last "initial" phase, or if we observed at least 5 reductions and the nominal CWND is less than 4/5th of the max nominal CWND.

We validated this test by comparing the ratio CWND/MAX_CWND for "valid" decisions, when we are simulating a competition scenario, and "spurious" decisions, when the "more than 3 consecutive reductions" test fires but we are not simulating any competition:

Ratio CWND/Max	valid	spurious
Average	30%	75%
Max	49%	100%
Top 25%	37%	91%
Media	35%	83%
Bottom 25%	20%	52%
Min	12%	25%
<50%	100%	20%

Table 1

Note that this validation was based on simulations, and that we cannot claim that our simulations perfectly reflect the real world. We will discuss in Section 3 how this imperfections lead us to use change our overall design.

Our initial exit competition algorithm was simple. C4 will exit the "pig war" mode if the available bandwidth increases.

2.2. Handling Chaotic Delays

Some Wi-Fi networks exhibit spikes in latency. These spikes are probably what caused the delay jitter discussed in [Cubic-QUIC-Blog]. We discussed them in more details in [Wi-Fi-Suspension-Blog]. We are not sure about the mechanism behind these spikes, but we have noticed that they mostly happen when several adjacent Wi-Fi networks are configured to use the same frequencies and channels. In these configurations, we expect the hidden node problem to result in some collisions. The Wi-Fi layer 2 retransmission algorithm takes care of these losses, but apparently uses an exponential back off algorithm to space retransmission delays in case of repeated collisions. When repeated collisions occur, the exponential backoff mechanism can cause large delays. The Wi-Fi layer 2 algorithm will also try to maintain delivery order, and subsequent packets will be queued behind the packet that caused the collisions.

In our initial design, we detected the advent of such "chaotic delay jitter" by computing a running estimate of the max RTT. We measured the max RTT observed in each round trip, to obtain the "era max RTT". We then computed an exponentially averaged "nominal max RTT":

```
nominal_max_rtt = (7 * nominal_max_rtt + era_max_rtt) / 8;
```

If the nominal max RTT was more than twice the min RTT, we set the "chaotic jitter" condition. When that condition was set, we stopped considering excess delay as an indication of congestion, and we changed the way we computed the "current CWND" used for the controlled path. Instead of simply setting it to "nominal CWND", we set it to a larger value:

```
target_cwnd = alpha*nominal_cwnd +
              (max_bytes_acked - nominal_cwnd) / 2;
```

In this formula, alpha is the amplification coefficient corresponding to the current state, such as for example 1 if "cruising" or 1.25 if "pushing" (see Section 5), and max_bytes_acked is the largest amount of bytes in flight that was succesfully acknowledged since the last initial phase.

The increased target_cwnd enabled C4 to keep sending data through most jitter events. There is of course a risk that this increased value will cause congestion. We limit that risk by only using half the value of max_bytes_ack, and by the setting a conservative pacing rate:

```
target_rate = alpha*nominal_rate;
```

Using the pacing rate that way prevents the larger window to cause big spikes in traffic.

The network conditions can evolve over time. C4 will keep monitoring the nominal max RTT, and will reset the "chaotic jitter" condition if nominal max RTT decreases below a threshold of 1.5 times the min RTT.

2.3. Monitor min RTT

Delay based algorithm rely on a correct estimate of the min RTT. They will naturally discover a reduction in the min RTT, but detecting an increase in the max RTT is difficult. There are known failure modes when multiple delay based algorithms compete, in particular the "late comer advantage".

In our initial design, the connections ensured that their min RTT is valid by occasionally entering a "slowdown" period, during which they set CWND to half the nominal value. This is similar to the "Probe RTT" mechanism implemented in BBR, or the "initial and periodic slowdown" proposed as extension to LEDBAT in [I-D.irtf-iccrg-ledbat-plus-plus]. In our implementation, the slowdown occurs if more than 5 seconds have elapsed since the previous slowdown, or since the last time the min RTT was set.

The measurement of min RTT in the period that follows the slowdown is considered a "clean" measurement. If two consecutive slowdown periods were followed by clean measurements larger than the current min RTT, we detect an RTT change and reset the connection. If the measurement results in the same value as the previous min RTT, C4 continue normal operation.

Some applications exhibit periods of natural slow down. This is the case for example of multimedia applications, when they only send differentially encoded frames. Natural slowdown was detected if an application sent less than half the nominal CWND during a period, and more than 4 seconds had elapsed since the previous slowdown or the previous min RTT update. The measurement that follows a natural slowdown was also considered a clean measurement.

A slowdown period corresponds to a reduction in offered traffic. If multiple connections are competing for the same bottleneck, each of these connections may experience cleaner RTT measurements, leading to equalization of the min RTT observed by these connections.

3. Simplifying the initial design

After extensive testing of our initial design, we felt we had drifted away from our initial "simplicity" tenet. The algorithms used to detect "pig war" and "chaotic jitter" were difficult to tune, and despite our efforts they resulted in many false positive or false negative. The "slowdown" algorithm made C4 less friendly to "real time" applications that prefer using stable estimated rates. These algorithms interacted with each other in ways that were sometimes hard to predict.

3.1. Chaotic jitter and rate control

As we observed the chaotic jitter behavior, we came to the conclusion that only controlling the CWND did not work well. we had a dilemma: either use a small CWND to guarantee that RTTs remain small, or use a large CWND so that transmission would not stall during peaks in jitter. But if we use a large CWND, we need some form of pacing to prevent senders from sending a large amount of packets too quickly. And then we realized that if we do have to set a pacing rate, we can simplify the algorithm.

Suppose that we compute a pacing rate that matches the network capacity, just like BBR does. Then, in first approximation, the setting the CWND too high does not matter too much. The number of bytes in flight will be limited by the product of the pacing rate by the actual RTT. We are thus free to set the CWND to a large value.

3.2. Monitoring the nominal max RTT

The observation on chaotic jitter leads to the idea of monitoring the maximum RTT. There is some difficulty here, because the observed RTT has three components:

* The minimum RTT in the absence of jitter

* The jitter caused by access networks such as Wi-Fi

* The delays caused by queues in the network

We cannot merely use the maximum value of the observed RTT, because of the queing delay component. In pushing periods, we are going to use data rate slightly higher than the measured value. This will create a bit of queuing, pushing the queing delay component ever higher -- and eventually resulting in "buffer bloat".

To avoid that, we can have periodic periods in which the endpoint sends data at deliberately slower than the rate estimate. This would enable a "clean" measurement of the Max RTT.

However, tests showed that only measuring the Max RTT during recovery periods is not reactive enough. For example, if the underlying RTT changes, we would need to wait up to 6 RTT before registering the change. In practice, we can measure the Max RTT in both the "recovery" and "cruising" periods, i.e., all the periods in which data is sent at most at the "nominal data rate".

If we are dealing with jitter, the clean Max RTT measurements will include whatever jitter was happening at the time of the measurement. It is not sufficient to measure the Max RTT once, we must keep the maximum value of a long enough series of measurement to capture the maximum jitter than the network can cause. But we are also aware that jitter conditions change over time, so we have to make sure that if the jitter diminished, the Max RTT also diminishes.

We solved that by measuring the Max RTT during the "recovery" periods that follow every "push". These periods occur about every 6 RTT, giving us reasonably frequent measurements. During these periods, we try to ensure clean measurements by setting the pacing rate a bit lower than the nominal rate -- 6.25% slower in our initial trials. We apply the following algorithm:

* compute the max_rtt_sample as the maximum RTT observed for packets sent during the recovery period.

* if the max_rtt_sample is more than max_jitter above running_min_rtt, reset it to running_min_rtt + max_jitter (by default, max_jitter is set to 250ms).

* if max_rtt_sample is larger than nominal_max_rtt, set nominal_max_rtt to that value.

* else, set nominal_max_rtt to:

```
nominal_max_rtt = gamma*max_rtt_sample +
                  (1-gamma)*nominal_max_rtt
```

The gamma coefficient is set to 1/8 in our initial trials.

3.2.1. Preventing Runaway Max RTT

Computing Max RTT the way we do bears the risk of "run away increase" of Max RTT:

* C4 notices high jitter, increases Nominal Max RTT accordingly, set CWND to the product of the increased Nominal Max RTT and Nominal Rate

* If Nominal rate is above the actual link rate, C4 will fill the pipe, and create a queue.

* On the next measurement, C4 finds that the max RTT has increased because of the queue, interprets that as "more jitter", increases Max RTT and fills the queue some more.

* Repeat until the queue become so large that packets are dropped and cause a congestion event.

Our proposed algorithm limits the Max RTT to at most running_min_rtt + max_jitter, but that is still risky. If congestion causes queues, the running measurements of min RTT will increase, causing the algorithm to allow for corresponding increases in max RTT. This would not happen as fast as without the capping to running_min_rtt + max_jitter, but it would still increase.

3.2.2. Initial Phase and Max RTT

During the initial phase, the nominal max RTT and the running min RTT are set to the first RTT value that is measured. This is not great in presence of high jitter, which causes C4 to exit the Initial phase early, leaving the nominal rate way too low. If C4 is competing on the Wi-Fi link against another connection, it might remain stalled at this low data rate.

We considered updating the Max RTT during the Initial phase, but that prevents any detection of delay based congestion. The Initial phase would continue until path buffers are full, a classic case of buffer bloat. Instead, we adopted a simple workaround:

* Maintain a flag "initial_after_jitter", initialized to 0.

* Get a measure of the max RTT after exit from initial.

* If C4 detects a "high jitter" condition and the "initial_after_jitter" flag is still 0, set the flag to 1 and re-enter the "initial" state.

Empirically, we detect high jitter in that case if the "running min RTT" is less that 2/5th of the "nominal max RTT".

3.3. Monitoring the nominal rate

The nominal rate is measured on each acknowledgement by dividing the number of bytes acknowledged since the packet was sent by the RTT measured with the acknowledgement of the packet, protecting against delay jitter as explained in Section 3.3.1, without additional filtering as discussed in Section 3.3.3.

We only use the measurements to increase the nominal rate, replacing the current value if we observe a greater filtered measurement. This is a deliberate choice, as decreases in measurement are ambiguous. They can result from the application being rate limited, or from measurement noises. Following those causes random decrease over time, which can be detrimental for rate limited applications. If the network conditions have changed, the rate will be reduced if congestion signals are received, as explained in Section 5.

3.3.1. Rate measurement

The simple algorithm protects from underestimation of the delay by observing that delivery rates cannot be larger than the rate at which the packets were sent, thus keeping the lower of the estimated receive rate and the send rate.

The algorithm uses four input variables:

* current_time: the time when the acknowledment is received.

* send_time: the time at which the highest acknowledged packet was sent.

* bytes_acknowledged: the number of bytes acknowledged by the receiver between send_time and current_time

* first_sent: the time at which the packet containing the first acknowledged bytes was sent.

The computation goes as follow:

```
ack_delay = current_time - send_time
send_delay = send_time - first_sent
measured_rate = bytes_acknowledged /
                max(ack_delay, send_delay)
```

This is in line with the specification of rate measurement in [I-D.ietf-ccwg-bbr].

We use the data rate measurement to update the nominal rate, but only if not congested (see Section 3.3.2)

```
if measured_rate > nominal_rate and not congested:
    nominal_rate = measured_rate
```

3.3.2. Avoiding Congestion Bounce

In our early experiments, we observed a "congestion bounce" that happened as follow:

* congestion is detected, the nominal rate is reduced, and C4 enters recovery.
* packets sent at the data rate that caused the congestion continue to be acknowledged during recovery.
* if enough packets are acknowledged, they will cause a rate measurement close to the previous nominal rate.
* if C4 accepts this new nominal rate, the flow will bounce back to the previous transmission rate, erasing the effects of the congestion signal.

Since we do not want that to happen, we specify that the nominal rate cannot be updated during congested periods, defined as:

* C4 is in "recovery" state,
* The recovery state was entered following a congestion signal, or a congestion signal was received since the beginning of the recovery era.

3.3.3. Not filtering the measurements

There is some noise in the measurements of the data rate, and we protect against that noise by retaining the maximum of the ack_delay and the send_delay. During early experiments, we considered smoothing the measurements for eliminating that noise.

The best filter that we could defined operated by smoothing the inverse of the data rate, the "time per byte sent". This works better because the data rate measurements are the quotient of the number of bytes received by the delay. The number of bytes received is easy to assert, but the measurement of the delays are very noisy. Instead of trying to average the data rates, we can average their inverse, i.e., the quotients of the delay by the bytes received, the times per byte. Then we can obtain smoothed data rates as the

inverse of these times per byte, effectively computing an harmonic average of measurements over time. We could for example compute an exponentially weighted moving average of the time per byte, and use the inverse of that as a filtered measurement of the data rate.

We do not specify any such filter in C4, because while filtering will reduce the noise, we will also delay any observation, resulting into a somewhat sluggish response to change in network conditions. Experience shows that the precaution of using the max of the ack delay and the send delay as a divider is sufficient for stable operation, and does not cause the response delays that filtering would.

3.4. Early Congestion Modification

We want C4 to handle Early Congestion Notification in a manner compatible with the L4S design. For that, we monitor the evolving ratio of CE marks that the L4S specification designates as alpha (we use ecn_alpha here to avoid confusion), and we detect congestion if the ratio grows over a threshold.

We did not find a recommended algorithm for computing ecn_alpha in either [RFC9330] or [RFC9331], but we could get some concrete suggestions in [I-D.briscoe-iccrg-prague-congestion-control]. That draft, now obsolete, suggests updating the ratio once per RTT, as the exponential weighted average of the fraction of CE marks per packet:

```
frac = nb_CE / (nb_CE + nb_ECT1)
ecn_alpha += (frac - ecn_alpha)/16
```

This kind of averaging introduces a reaction delay. The draft suggests mitigating that delay by preempting the averaging if the fraction is large:

```
if frac > 0.5:
    ecn_alpha = frac
```

We followed that design, but decided to update the coefficient after each acknowledgement, instead of after each RTT. This is in line with our implementation of "delayed acknowledgements" in QUIC, which results in a small number of acknowledgements per RTT.

The reaction of C4 to an excess of CE marks is similar to the reaction to excess delays or to packet losses, see Section 5.

4. Competition with other algorithms

We saw in Section 2.1 that delay based algorithms required a special "escape mode" when facing competition from algorithms like Cubic. Relying on pacing rate and max RTT instead of CWND and min RTT makes this problem much simpler. The measured max RTT will naturally increase as algorithms like Cubic cause buffer bloat and increased queues. Instead of being shut down, C4 will just keep increasing its max RTT and thus its running CWND, automatically matching the other algorithm’s values.

We verified that behavior in a number of simulations. We also verified that when the competition ceases, C4 will progressively drop its nominal max RTT, returning to situations with very low queuing delays.

4.1. No need for slowdowns

The fairness of delay based algorithm depends on all competing flows having similar estimates of the min RTT. As discussed in Section 2.3, this ends up creating variants of the latecomer advantage issue, requiring a periodic slowdown mechanism to ensure that all competing flow have chance to update the RTT value.

This problem is caused by the default algorithm of setting min RTT to the minimum of all RTT sample values since the beginning of the connection. Flows that started more recently compute that minimum over a longer period, and thus discover a larger min RTT than older flows. This problem does not exist with max RTT, because all competing flows see the same max RTT value. The slowdown mechanism is thus not necessary.

Removing the need for a slowdown mechanism allows for a simpler protocol, better suited to real time communications.

5. React quickly to changing network conditions

Our focus is on maintaining low delays, and thus reacting quickly to changes in network conditions. We can detect some of these changes by monitoring the RTT and the data rate, but experience with the early version of BBR showed that completely ignoring packet losses can lead to very unfair competition with Cubic. The L4S effort is promoting the use of ECN feedback by network elements (see [RFC9331]), which could well end up detecting congestion and queues more precisely than the monitoring of end-to-end delays. C4 will thus detect changing network conditions by monitoring 3 congestion control signals:

1. Excessive increase of measured RTT (above the nominal Max RTT),

2. Excessive rate of packet losses (but not mere Probe Time Out, see Section 5.1),

3. Excessive rate of ECN/CE marks

If any of these signals is detected, C4 enters a "recovery" state. On entering recovery, C4 reduces the nominal_rate by a factor "beta":

```
# on congestion detected:
nominal_rate = (1-beta)*nominal_rate
```

The cofficient beta differs depending on the nature of the congestion signal. For packet losses, it is set to 1/4, similar to the value used in Cubic. For delay based losses, it is proportional to the difference between the measured RTT and the target RTT divided by the acceptable margin, capped to 1/4. If the signal is an ECN/CE rate, we may use a proportional reduction coefficient in line with [RFC9331], again capped to 1/4.

During the recovery period, target CWND and pacing rate are set to a fraction of the "nominal rate" multiplied by the "nominal max RTT". The recovery period ends when the first packet sent after entering recovery is acknowledged. Congestion signals are processed when entering recovery; further signals are ignored until the end of recovery.

Network conditions may change for the better or for the worse. Worsening is detected through congestion signals, but increases can only be detected by trying to send more data and checking whether the network accepts it. Different algorithms have done two ways: pursuing regular increases of CWND until congestion finally occurs, like for example the "congestion avoidance" phase of TCP RENO; or periodically probe the network by sending at a higher rate, like the Probe Bandwidth mechanism of BBR. C4 adopt the periodic probing approach, in particular because it is a better fit for variable rate multimedia applications (see details in Section 7).

5.1. Do not react to Probe Time Out

QUIC normally detect losses by observing gaps in the sequences of acknowledged packet. That’s a robust signal. QUIC will also inject "Probe time out" packets if the PTO timeout elapses before the last sent packet has not been acknowledged. This is not a robust congestion signal, because delay jitter may also cause PTO timeouts. When testing in "high jitter" conditions, we realized that we should not change the state of C4 for losses detected solely based on timer,

and only react to those losses that are detected by gaps in acknowledgements.

5.2. Update the Nominal Rate after Pushing

C4 configures the transport with a larger rate and CWND than the nominal CWND during "pushing" periods. The peer will acknowledge the data sent during these periods in the round trip that followed.

When we receive an ACK for a newly acknowledged packet, we update the nominal rate as explained in Section 3.3.

This strategy is effectively a form of "make before break". The pushing only increase the rate by a fraction of the nominal values, and only lasts for one round trip. That limited increase is not expected to increase the size of queues by more than a small fraction of the bandwidth*delay product. It might cause a slight increase of the measured RTT for a short period, or perhaps cause some ECN signaling, but it should not cause packet losses -- unless competing connections have caused large queues. If there was no extra capacity available, C4 does not increase the nominal CWND and the connection continues with the previous value.

6. Driving for fairness

Many protocols enforce fairness by tuning their behavior so that large flows become less aggressive than smaller ones, either by trying less hard to increase their bandwidth or by reacting more to congestion events. We considered adopting a similar strategy for C4.

The aggressiveness of C4 is driven by several considerations:

* the frequency of the "pushing" periods,

* the coefficient alpha used during pushing,

* the coefficient beta used during response to congestion events,

* the delay threshold above a nominal value to detect congestion,

* the ratio of packet losses considered excessive,

* the ratio of ECN marks considered excessive.

We clearly want to have some or all of these parameters depend on how much resource the flow is using. There are know limits to these strategies. For example, consider TCP Reno, in which the growth rate of CWND during the "congestion avoidance" phase" is inversely

proportional to its size. This drives very good long term fairness, but in practice it prevents TCP Reno from operating well on high speed or high delay connections, as discussed in the "problem description" section of [RFC3649]. In that RFC, Sally Floyd was proposing using a growth rate inversely proportional to the logarithm of the CWND, which would not be so drastic.

In the initial design, we proposed making the frequency of the pushing periods inversely proportional to the logarithm of the CWND, but that gets in tension with our estimation of the max RTT, which requires frequent "recovery" periods. We would not want the Max RTT estimate to work less well for high speed connections! We solved the tension in favor of reliable max RTT estimates, and fixed to 4 the number of Cruising periods between Recovery and Pushing. The whole cycle takes about 6 RTT.

We also reduced the default rate increase during Pushing to 6.25%, which means that the default cycle is more on less on par with the aggressiveness of RENO when operating at low bandwidth (lower than 34 Mbps).

6.1. Absence of constraints is unfair

Once we fixed the push frequency and the default increase rate, we were left with responses that were mostly proportional to the amount of resource used by a connection. Such design makes the resource sharing very dependent on initial conditions. We saw simulations where after some initial period, one of two competing connections on a 20 Mbps path might settle at a 15 Mbps rate and the other at 5 Mbps. Both connections would react to a congestion event by dropping their bandwidth by 25%, to 15 or 3.75 Mbps. And then once the condition eased, both would increase their data rate by the same amount. If everything went well the two connection will share the bandwidth without exceeding it, and the situation would be very stable -- but also very much unfair.

We also had some simulations in which a first connection will grab all the available bandwidth, and a late comer connection would struggle to get any bandwidth at all. The analysis showed that the second connection was exiting the initial phase early, after encountering either excess delay or excess packet loss. The first connection was saturating the path, any additional traffic did cause queuing or losses, and the second connection had no chance to grow.

This "second comer shut down" effect happened particularly often on high jitter links. The established connections had tuned their timers or congestion window to account for the high jitter. The second connection was basing their timers on their first

measurements, before any of the big jitter events had occured. This caused an imbalance between the first connection, which expected large RTT variations, and the second, which did not expect them yet.

These shutdown effects happened in simulations with the first connection using either Cubic, BBR or C4. We had to design a response, and we first turned to making the response to excess delay or packet loss a function of the data rate of the flow.

6.2. Introducing a sensitivity curve

In our second design, we attempted to fix the unfairness and shutdowns effect by introducing a sensitivity curve, computing a "sensitivity" as a function of the flow data rate. Our first implementation is simple:

* set sensitivity to 0 if data rate is lower than 50000B/s
* linear interpolation between 0 and 0.92 for values between 50,000 and 1,000,000B/s.
* linear interpolation between 0.92 and 1 for values between 1,000,000 and 10,000,000B/s.
* set sensitivity to 1 if data rate is higher than 10,000,000B/s

The sensitivity index is then used to set the value of delay and loss thresholds. For the delay threshold, the rule is:

```
delay_fraction = 1/16 + (1 - sensitivity)*3/16
delay_threshold = min(25ms, delay_fraction*nominal_max_rtt)
```

For the loss threshold, the rule is:

```
loss_threshold = 0.02 + 0.50 * (1-sensitivity);
```

For the CE mark threshold, the rule is:

```
loss_threshold = 1/32 + 1/32 * (1-sensitivity);
```

This very simple change allowed us to stabilize the results. In our competition tests we see sharing of resource almost equitably between C4 connections, and reasonably between C4 and Cubic or C4 and BBR. We do not observe the shutdown effects that we saw before.

There is no doubt that the current curve will have to be refined. We have a couple of such tests in our test suite with total capacity higher than 20Mbps, and for those tests the dependency on initial conditions remain. We will revisit the definition of the curve, probably to have the sensitivity follow the logarithm of data rate.

6.3. Cascade of Increases

We sometimes encounter networks in which the available bandwidth changes rapidly. For example, when a competing connection stops, the available capacity may double. With low Earth orbit satellite constellations (LEO), it appears that ground stations constantly check availability of nearby satellites, and switch to a different satellite every 10 or 15 seconds depending on the constellation (see [ICCRG-LEO]), with the bandwidth jumping from 10Mbps to 65Mbps.

Because we aim for fairness with RENO or Cubic, the cycle of recovery, cruising and pushing will only result in slow increases increases, maybe 6.25% after 6 RTT. This means we would only double the bandwidth after about 68 RTT, or increase from 10 to 65 Mbps after 185 RTT -- by which time the LEO station might have connected to a different orbiting satellite. To go faster, we implement a "cascade": if the previous pushes at 6.25% was successful, the next pushing will use 25% (see Section 7.2), or an intermediate value if the observed ratio of ECN marks is greater than 0. If three successive pushes all result in increases of the nominal rate, C4 will reenter the "startup" mode, during which each RTT can result in a 100% increase of rate and CWND.

7. Supporting Application Limited Connections

C4 is specially designed to support multimedia applications, which very often operate in application limited mode. After testing and simulations of application limited applications, we incorporated a number of features.

The first feature is the design decision to only lower the nominal rate if congestion is detected. This is in contrast with the BBR design, in which the estimate of bottleneck bandwidth is also lowered if the bandwidth measured after a "probe bandwidth" attempt is lower than the current estimate while the connection was not "application limited". We found that detection of the application limited state was somewhat error prone. Occasional errors end up with a spurious reduction of the estimate of the bottleneck bandwidth. These errors can accumulate over time, causing the bandwidth estimate to "drift down", and the multimedia experience to suffer. Our strategy of only reducing the nominal values in reaction to congestion notifications much reduces that risk.

The second feature is the "make before break" nature of the rate updates discussed in Section 5.2. This reduces the risk of using rates that are too large and would cause queues or losses, and thus makes C4 a good choice for multimedia applications.

C4 adds two more features to handle multimedia applications well: coordinated pushing (see Section 7.1), and variable pushing rate (see Section 7.2).

7.1. Coordinated Pushing

As stated in Section 6, the connection will remain in "cruising" state for a specified interval, and then move to "pushing". This works well when the connection is almost saturating the network path, but not so well for a media application that uses little bandwidth most of the time, and only needs more bandwidth when it is refreshing the state of the media encoders and sending new "reference" frames. If that happens, pushing will only be effective if the pushing interval coincides with the sending of these reference frames. If pushing happens during an application limited period, there will be no data to push with and thus no chance of increasing the nominal rate and CWND. If the reference frames are sent outside of a pushing interval, the rate and CWND will be kept at the nominal value.

To break that issue, one could imagine sending "filler" traffic during the pushing periods. We tried that in simulations, and the drawback became obvious. The filler traffic would sometimes cause queues and packet losses, which degrade the quality of the multimedia experience. We could reduce this risk of packet losses by sending redundant traffic, for example creating the additional traffic using a forward error correction (FEC) algorithm, so that individual packet losses are immediately corrected. However, this is complicated, and FEC does not always protect against long batches of losses.

C4 uses a simpler solution. If the time has come to enter pushing, it will check whether the connection is "application limited", which is simply defined as testing whether the application send a "nominal CWND" worth of data during the previous interval. If it is, C4 will remain in cruising state until the application finally sends more data, and will only enter the the pushing state when the last period was not application limited.

7.2. Variable Pushing Rate

C4 tests for available bandwidth at regular pushing intervals (see Section 6), during which the rate and CWND is set at 25% more than the nominal values. This mimics what BBR is doing, but may be less than ideal for real time applications. When in pushing state, the application is allowed to send more data than the nominal CWND, which causes temporary queues and degrades the experience somewhat. On the other hand, not pushing at all would not be a good option, because the connection could end up stuck using only a fraction of the available capacity. We thus have to find a compromise between operating at low capacity and risking building queues.

We manage that compromise by adopting a variable pushing rate:

* If pushing at 25% did not result in a significant increase of the nominal rate, the next pushing will happen at 6.25%

* If pushing at 6.25% did result in some increase of the nominal CWIN, the next pushing will happen at 25%, otherwise it will remain at 6.25%

If the observed ratio of ECN-CE marks is greater than zero, we will use it to modulate the amount of pushing. We leave the pushing rate at 6.25% if the previous pushing attempt was not successful, but otherwise we pick a value intermediate between 25% (if 0 ECN marks) and 6.25% (if the ratio of ECN marks approaches the threshold).

As explained in Section 6.3, if three consecutive pushing attempts result in significant increases, C4 detects that the underlying network conditions have changed, and will reenter the startup state.

The "significant increase" mentioned above is a matter of debate. Even if capacity is available, increasing the send rate by 25% does not always result in a 25% increase of the acknowledged rate. Delay jitter, for example, may result in lower measurement. We initially computed the threshold for detecting "significant" increase as 1/2 of the increase in the sending rate, but multiple simulation shows that was too high and caused lower performance. We now set that threshold to 1/4 of the increase in the sending rate.

7.3. Pushing rate and Cascades

The choice of a 25% push rate was motivated by discussions of BBR design. Pushing has two parallel functions: discover the available capacity, if any; and also, push back against other connections in case of competition. Consider for example competition with Cubic. The Cubic connection will only back off if it observes packet losses, which typically happen when the bottleneck buffers are full. Pushing at a high rate increases the chance of building queues, overfilling the buffers, causing losses, and thus causing Cubic to back off. Pushing at a lower rate like 6.25% would not have that effect, and C4 would keep using a lower share of the network. This is why we will always pushed at 25% in the "pig war" mode.

The computation of the interval between pushes is tied to the need to compete nicely, and follows the general idea that the average growth rate should mimic that of RENO or Cubic in the same circumstances. If we pick a lower push rate, such as 6.25% or maybe 12.5%, we might be able to use shorter intervals. This could be a nice compromise: in normal operation, push frequently, but at a low rate. This would not create large queues or disturb competing connections, but it will let C4 discover capacity more quickly. Then, we could use the "cascade" algorithm to push at a higher rate, and then maybe switch to startup mode if a lot of capacity is available. This is something that we intend to test, but have not implemented yet.

7.4. Adaptation to ECN/L4S

Tests with L4S active queue management showed the tension between the periodic updates and L4S goal to minimize queue sizes. Typical L4S deployment start marking packets with ECN/CE when the queue size is about 1.5ms, and increase the mark rate progressively as the queue size increases, reaching 100% when the queue size is about 2ms. If C4 pushes at 25% every 6 RTT, and if the bandwidth estimate is accurate, the queue size will increase by 25% of the RTT during the first roundtrip, before any correction signal can be applied. The increased marking rate will affect all connections sharing the bottleneck, which is not desirable.

L4S is tuned for the "Prague" algorithm, which increases CWIN by one packet every RTT. In a typical trial with a 20ms RTT and a 100 Mbps data rate, it takes 0.12ms to send a packet, and thus 12.5 RTT before building a queue of 1.5ms. In the same conditions, C4 would have increased the rate by 25% after 6 RTT in the aggressive scenario, thus triggering a high rate of marking.

The cascade process made the problem even worse. If a push at 6.25% does increase the nominal rate, the next push will be at 25%. If that push and the next one did increase the nominal rate, C4 will reenter the initial phase, even if some of the pushes did cause ECN/CE marks. The initial phase will then cause a lot of packet losses, which will degrade performance.

To mitigate this issue, we had to add a "very low" pushing mode, setting the pushing rate to only 3.125% if the previous push resulted in a high rate of ECN/CE marks. We also replaced the somewhat adhoc "count of successive probes" by the management of a "probe level", defining 4 levels:

* level 0: pushing at 3.125%, spend 1 cycle in cruising before pushing.

* level 1: pushing at 6.25%, spend 4 cycles in cruising before pushing.

* level 2: pushing at 25%, spend at most 1 cycle in cruising before pushing.

* level 3: pushing at 25%, spend at most 1 cycle in cruising before pushing.

The "probe level" is updated after the recovery phase as follow:

* if the previous probe was successful and did not result in a high rate of ECN/CE marks, increase the probe level by 1. If the probe level was already at 3, reenter the startup phase.

* if the previous probe was successful but did result in a high rate of ECN/CE marks, remain at the same probe level.

* if the previous probe was not successful but did not result in a high rate of ECN/CE marks, stay at probe level 0 if already at that level, otherwise move back to probe level 1.

* if the previous probe was not successful and did result in a high rate of ECN/CE marks, move to probe level 0.

This logic treats the CE marking differently from other congestion signals, because the CE marks are an intentional indication of congestion by the network, and is thus less ambiguous than delay increases or packet losses, which can be caused by other factors such as delay jitter or random transmission issues. Simulations show that this logic allows to quickly discover the available capacity in L4S networks, whithout spuriously reentering the startup phase and causing packet losses. It is equivalent to the previous logic when the network does not support L4S.

8. Revisiting the Initial Phase

Our November 2025 design of C4 included a "rate based" initial phase, during which C4 will send at twice the "nominal rate", monitor acknowledgments and increase the nominal rate if measurements increase, and exit if congestion is detected or if the measurements do not increase for 3 consecutive RTT. That algorithm works well in most scenario, but we were observing early exits in "high delay jitter" scenarios, such as Wi-Fi networks with lots of packet collisions.

After observing that phenomenon, we realized that the rate based algorithm was failing in case of high delay jitter because it was setting the CWND to the product of pacing rate and the "nominal" max RTT. The nominal Max RTT was set to a fixed value, observed either before the initial phase or on the first roundtrip in that phase. It would work if the initial phase started during a high jitter event and the initial RTT was large enough, but in many case it was not and became a limiting factor.

8.1. Why not increasing Max RTT during Initial phase?

In the initial phase, the algorithm tries to discover the bandwidth and does not yet have a good estimate of delay jitter, which typically requires a series of measurements. In these conditions, it is easy to underestimate the max RTT. On the other hand, the flow is deliberately probing at a high data rate. If the algorithm allows updates of max RTT during that phase, the risks of spiraling into buffer boat are very high, but if the CWND remains too low, the risk of exiting startup with a severely underestimated data rate is also very high.

We tried to develop simple rules to classify the delay measurements between caused by jitter, and caused by congestion. If we could do that, we would be able to increase the max RTT safely, when appropriate. However, we could not find variables that were both easy to monitor and well correlated with the actual cause of the delay.

8.2. Building a robust initial estimator

The "rate based" initial estimator requires estimating both the data rate and the max RTT simultaneously. In contrast, the "CWND based" initial estimator use in algorithms like Reno or Cubic only requires estimating the CWND, plus a possibly loose estimate of the data rate. The Reno algorithm is remarkably simple: just increase the CWND by the number of bytes acknowledged, without any explicit dependency on the measured latency.

The Reno algorithm terminates when packet losses are observed, leading to bufferbloat. Hystart improves that by terminating when the measured delays start increasing, but this can lead to early exit in case of delay jitter. The rate based algorithm terminate when the measured bandwidth stops growing, which provides good results. Our proposal is to combine a Reno like growth of the CWND with a rate-control like exit condition.

Of course, things are not that simple. The "rate" test only stops the growth of the CWND after the third "non growing" round. If CWND doubles after each round it becomes excessive, buffers fill up, and lots of packets are lost. We dealt with that problem by essentially freezing the increases of after the first "non growing" round. If a larger measurement happens before 3 RTT, the increases resume, otherwise, C4 exits the initial phase.

When the initial phase completes, we retain as estimate of the data rate the highest value measured so far. We also want to obtain a reasonable estimate of the "max RTT". In the Reno logic, the "ssthresh" is set to half the CWND value before congestion is detected. C4 will not use the ssthresh variable after exiting the Initial phase, but it can set the max RTT to the quotient of ssthresh by the final rate estimate.

9. State Machine

The state machine for C4 has the following states:

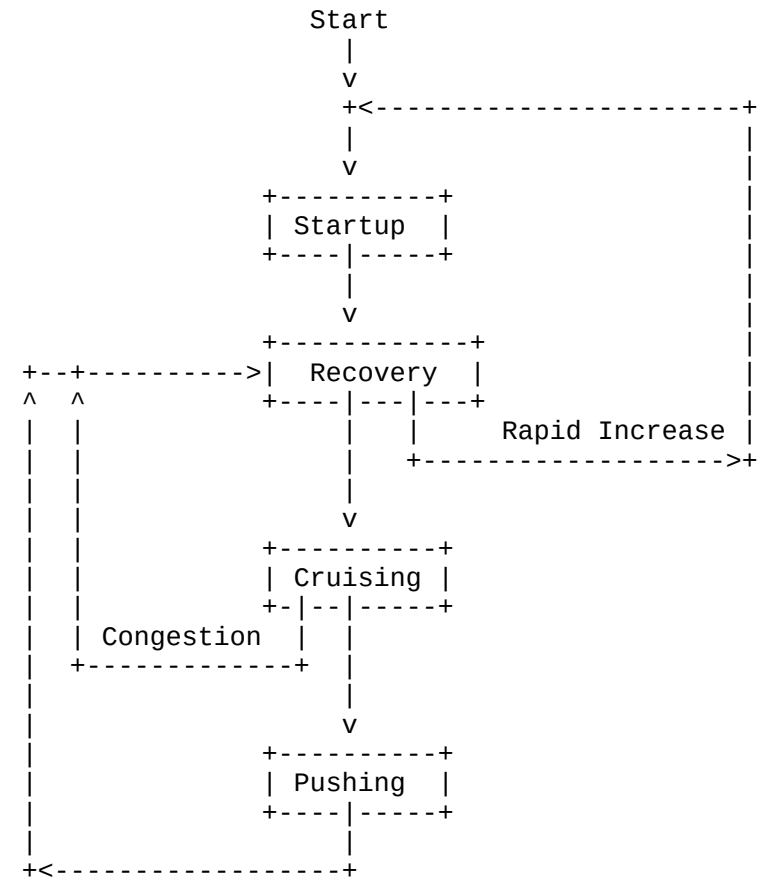
* "startup": the initial state, during which the CWND is set to twice the "nominal_CWND". The connection exits startup if the "nominal_cwnd" does not increase for 3 consecutive round trips. When the connection exits startup, it enters "recovery".

* "recovery": the connection enters that state after "startup", "pushing", or a congestion detection in a "cruising" state. It remains in that state for at least one roundtrip, until the first packet sent in "recovery" is acknowledged. Once that happens, the connection goes back to "startup" if the last 3 pushing attemps have resulted in increases of "nominal rate", or enters "cruising" otherwise.

* "cruising": the connection is sending using the "nominal_rate" and "nominal_max_rtt" value. If congestion is detected, the connection exits cruising and enters "recovery" after lowering the value of "nominal_cwnd". Otherwise, the connection will remain in "cruising" state until at least 4 RTT and the connection is not "app limited". At that point, it enters "pushing".

* "pushing": the connection is using a rate and CWND 25% larger than "nominal_rate" and "nominal_CWND". It remains in that state for one round trip, i.e., until the first packet send while "pushing" is acknowledged. At that point, it enters the "recovery" state.

These transitions are summarized in the following state diagram.



10. Security Considerations

We do not believe that C4 introduce new security issues. Or maybe there are, such as what happen if applications can be fooled in going to fast and overwhelming the network, or going to slow and underwhelming the application. Discuss!

11. IANA Considerations

This document has no IANA actions.

12. Informative References

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[I-D.ietf-moq-transport]
Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell, "Media over QUIC Transport", Work in Progress, Internet-Draft, draft-ietf-moq-transport-16, 13 January 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-moq-transport-16>.

[RFC9438] Xu, L., Ha, S., Rhee, I., Goel, V., and L. Eggert, Ed., "CUBIC for Fast and Long-Distance Networks", RFC 9438, DOI 10.17487/RFC9438, August 2023, <https://www.rfc-editor.org/info/rfc9438>.

[I-D.ietf-ccwg-bbr]
Cardwell, N., Swett, I., and J. Beshay, "BBR Congestion Control", Work in Progress, Internet-Draft, draft-ietf-ccwg-bbr-04, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-ccwg-bbr-04>.

[RFC6817] Shalunov, S., Hazel, G., Iyengar, J., and M. Kuehlewind, "Low Extra Delay Background Transport (LEDBAT)", RFC 6817, DOI 10.17487/RFC6817, December 2012, <https://www.rfc-editor.org/info/rfc6817>.

[RFC6582] Henderson, T., Floyd, S., Gurtov, A., and Y. Nishida, "The NewReno Modification to TCP’s Fast Recovery Algorithm", RFC 6582, DOI 10.17487/RFC6582, April 2012, <https://www.rfc-editor.org/info/rfc6582>.

[RFC3649] Floyd, S., "HighSpeed TCP for Large Congestion Windows", RFC 3649, DOI 10.17487/RFC3649, December 2003, <https://www.rfc-editor.org/info/rfc3649>.

[TCP-Vegas]
Brakmo, L. S. and L. L. Peterson, "TCP Vegas: end to end congestion avoidance on a global Internet", IEEE Journal on Selected Areas in Communications (Volume: 13, Issue: 8, October 1995) , October 1995, <https://ieeexplore.ieee.org/document/464716>.

[HyStart] Ha, S. and I. Rhee, "Taming the elephants: New TCP slow start", Computer Networks vol. 55, no. 9, pp. 2092-2110 , June 2011, <https://doi.org/10.1016/j.comnet.2011.01.014>.

[Cubic-QUIC-Blog]
Huitema, C., "Implementing Cubic congestion control in Quic", Christian Huitema’s blog , November 2019, <https://www.privateoctopus.com/2019/11/11/implementing-cubic-congestion-control-in-quic/>.

[Wi-Fi-Suspension-Blog]
Huitema, C., "The weird case of the wifi latency spikes", Christian Huitema’s blog , May 2023, <https://www.privateoctopus.com/2023/05/18/the-weird-case-of-wifi-latency-spikes.html>.

[I-D.irtf-iccrg-ledbat-plus-plus]
Balasubramanian, P., Ertugay, O., Havey, D., and M. Bagnulo, "LEDBAT++: Congestion Control for Background Traffic", Work in Progress, Internet-Draft, draft-irtf-iccrg-ledbat-plus-plus-06, 29 January 2026, <https://datatracker.ietf.org/doc/html/draft-irtf-iccrg-ledbat-plus-plus-06>.

[RFC9330] Briscoe, B., Ed., De Schepper, K., Bagnulo, M., and G. White, "Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture", RFC 9330, DOI 10.17487/RFC9330, January 2023, <https://www.rfc-editor.org/info/rfc9330>.

[RFC9331] De Schepper, K. and B. Briscoe, Ed., "The Explicit Congestion Notification (ECN) Protocol for Low Latency, Low Loss, and Scalable Throughput (L4S)", RFC 9331, DOI 10.17487/RFC9331, January 2023, <https://www.rfc-editor.org/info/rfc9331>.

[I-D.briscoe-iccrg-prague-congestion-control]
De Schepper, K., Tilmans, O., Briscoe, B., and V. Goel, "Prague Congestion Control", Work in Progress, Internet-Draft, draft-briscoe-iccrg-prague-congestion-control-04, 24 July 2024, <https://datatracker.ietf.org/doc/html/draft-briscoe-iccrg-prague-congestion-control-04>.

[ICCRG-LEO]
Lai, Z., Li, Z., Wu, Q., Li, H., and Q. Zhang, "Mind the Misleading Effects of LEO Mobility on End-to-End Congestion Control", Slides presented at ICCRG meeting during IETF 122 , March 2025, <https://datatracker.ietf.org/meeting/122/materials/slides-122-iccrg-mind-the-misleading-effects-of-leo-mobility-on-end-to-end-congestion-control-00>.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Christian Huitema
Private Octopus Inc.
Email: huitema@huitema.net

Suhas Nandakumar
Cisco
Email: snandaku@cisco.com

Cullen Jennings
Cisco
Email: fluffy@iii.ca