

An Empirical Study of Ageing in the Cloud

Tanya Shreedhar
Wireless Systems Lab, IIIT-Delhi
tanyas@iiitd.ac.in

Sanjit K. Kaul
Wireless Systems Lab, IIIT-Delhi
skkaul@iiitd.ac.in

Roy D. Yates
WINLAB, Rutgers University
ryates@winlab.rutgers.edu

Abstract—We quantify, over inter-continental paths, the ageing of TCP packets, throughput and delay for different TCP congestion control algorithms containing a mix of loss-based, delay-based and hybrid congestion control algorithms. In comparing these TCP variants to ACP+, an improvement over ACP, we shed better light on the ability of ACP+ to deliver timely updates over fat pipes and long paths. ACP+ estimates the network conditions on the end-to-end path and adapts the rate of status updates to minimize age. It achieves similar average age as the *best* (age wise) performing TCP algorithm but at end-to-end throughputs that are two orders of magnitude smaller. We also quantify the significant improvements that ACP+ brings to age control over a shared multiaccess channel.

I. INTRODUCTION

The challenge of age control over an end-to-end path in the cloud is to adapt the rate of status updates entering the path so that there are as many status updates in transit as possible while no update waits behind another in a router queue. This is in contrast to Transmission Control Protocol (TCP) loss-based congestion control algorithms that aim for high throughput by having as many packets as possible in each queue without exceeding the available queue occupancy. This allows an end-to-end flow to achieve a rate equal to the bottleneck bandwidth; however, this is at the expense of large delays and eventual losses due to excessive queuing at the bottleneck.

Recently, requirements of low latency have led to the proposal of hybrid congestion control mechanisms such as BBR [4]. At its stated ideal point of operation, BBR would have TCP packets delivered to the receiver at the bottleneck link rate, while each packet would experience an average delay as that experienced by a packet if only it was sent over the path. Intriguingly, this would satisfy the goal of age control by resulting in the highest rate of packet delivery at the receiver and lowest packet delays.

In our prior work [14], we had proposed the Age Control Protocol (ACP), a transport layer protocol that regulates the rate at which status updates are sent by an application over an end-to-end path. By abstracting away an end-to-end path as a series of queues, we had argued that a good age control algorithm must try to have as many status updates in transit as possible while trying to ensure that the updates don't wait for previously queued prior updates from the application.

Figure 1 provides an illustration of a good age control strategy in action for an end-to-end path of three identical queues, each with deterministic service times. Figures 1a and 1b, respectively, have too many and too few updates, resulting in high age. Figure 1c shows the snapshot one would

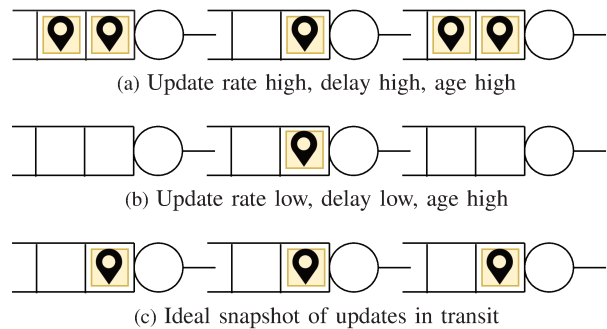


Figure 1: An illustration of queue occupancy and its impact on age.

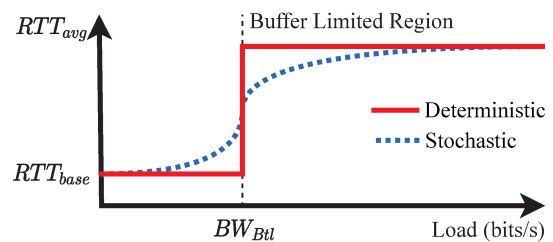
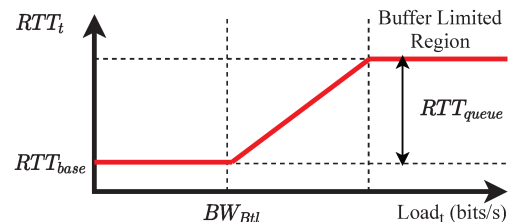


Figure 2: An illustration of how round-trip times vary as a function of the offered load. While (a) shows the change in instantaneous RTT as the load increases, (b) shows the steady-state average behavior at a chosen load.

expect to see with a good age control algorithm sending updates over the three-queue network. Of course, the picture becomes more complicated when the queues have random service times. For example, with a pair of M/M/1 queues in tandem, the average number of packets queued in the system at minimum age was shown to be ≈ 1.6 updates [13].

While applications have diversified significantly over the past few decades, TCP congestion control remains the primary

mechanism by which end hosts share the available bandwidth. For the purposes of TCP, the end-to-end path may be abstracted away as a link with bottleneck bandwidth BW_{Btl} and a round-trip propagation time of RTT_{base} [4]. Figure 2a provides an illustration, akin to that in [4, Figure 1], of the instantaneous round-trip time RTT_t at time t as a function of the current offered load (the effective rate at which TCP is sending bytes). As long as the offered load is smaller than BW_{Btl} , the TCP packets see a low RTT of RTT_{base} . Once the offered load becomes larger than BW_{Btl} , the TCP packets that arrive at the link's queue see increasingly more packets waiting for service ahead of them. This results in a linear increase in RTT_t until the queue becomes buffer limited, the RTT saturates and TCP packets arriving at a full queue are dropped.

Traditionally, TCP's congestion control allows for an increasing number of unacknowledged bytes from an application to flow through the network pipe until one or more bytes are lost due to the resulting congestion. Such a loss-based congestion control algorithm keeps increasing the offered load until a packet is lost as a result of the link operating in the buffer limited region. The flow will achieve a throughput equal to the bottleneck bandwidth, but packets in the flow will suffer large round-trip times, especially when the link has a large buffer.

Figure 2a suggests that one would like to operate at the lower “knee” in the curve, i.e., close to the bottleneck throughput BW_{Btl} at low delays. In fact, delay-based and hybrid congestion control algorithms such as the recently proposed *Bottleneck Bandwidth and Round-trip propagation time* (BBR) protocol, attempt this by using the round-trip time to detect congestion early, before a loss occurs due to buffer unavailability at a certain router along the path. Note that this combination of a throughput of BW_{Btl} and round-trip times of RTT_{base} is in fact achieved by the snapshot in Figure 1c that illustrates a good age control algorithm in action.

Of course, as was observed in [9] in relation to the stated point of operation of BBR, when a path is better modeled by a stochastic service facility, the average round-trip times at the maximum achievable throughput of BW_{Btl} could be much larger than RTT_{base} . Figure 2b provides an illustration of steady-state average RTT as a function of average load. The red and blue curves, respectively, correspond to a deterministic and a stochastic service facility.

This shift in congestion control algorithms from keeping the pipe full to “keeping the pipe just full, but no fuller” [9], motivates this empirical study of how the information at a receiver would age if updates were transmitted over the cloud using the congestion control algorithms. However, we must be careful as (a) TCP doesn't regulate the rate of generation of packets by the status updating application, (b) it is a stream-based protocol and has no notion of update packets. As illustrated in Figure 3, an application writes a stream of bytes to the TCP sender's buffer. TCP creates segments from these bytes in a first-come-first-serve manner. TCP segments are delivered to the TCP receiver. At any time, TCP allows a total of up to a current congestion window size of bytes to be

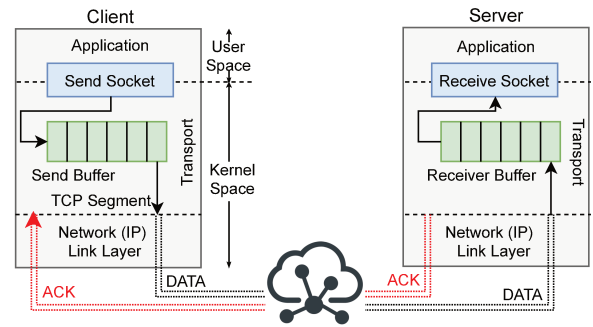


Figure 3: TCP Network Stack

in transit in the network. The TCP receiver sends an ACK to inform the sender of the last segment received.

To stay focused on evaluating how scheduling TCP segments over an end-to-end path would age updates at a receiver, we assume that a TCP segment, when created, contains fresh information. Specifically, we ignore the ageing of bytes while they wait in the TCP send buffer. One way of achieving this in practice would be to have the application provide freshly generated information (as in a generate-at-will model [16]) to be incorporated in a TCP segment just as TCP schedules it for sending.

Here, we approximate the age of the segment when it arrives at the TCP receiver to be the RTT of the segment, which is calculated based on the time of receipt of the TCP ACK that acknowledges receipt of the segment. Further, we approximate the inter-delivery time of segments at the receiver by the inter-delivery times of the corresponding ACKs. The RTT(s) and the inter-delivery times together allow us to come up with an estimate of the time-average of age at the receiver that results from a chosen congestion control algorithm, using the graphical method of time-average age calculation [8].

Last but not the least, we would like to minimize the impact of packet loss due to link transmission errors on our evaluation of congestion control. Given our focus on paths in the cloud, specifically between AWS data centers, we observe a very small percentage of loss, and that too because of buffer overflows in routers that result in the process of congestion control estimating the bottleneck bandwidth.

Our specific contributions include:

- 1) We propose the ACP+ algorithm and explain the changes in it with respect to ACP. (§ II)
- 2) We provide an empirical study of age, throughput and delay trade-offs obtained when using state-of-the-art TCP congestion control algorithms to transport updates over an end-to-end path in the cloud.

We evaluate a mix of loss-based (Reno [11] and CUBIC [5]), delay-based (Vegas [3]) and hybrid congestion control algorithms (YeAH [2] and BBR [4]) for different settings of receiver buffer size. We compare the performance of the TCP algorithms with that of ACP+. We show that ACP+ does well to estimate the network conditions on the end-to-end path and appropriately adapts the rate of

status updates sent over the path to keep age at the receiver close to the minimum. (§ III)

- 3) We show that ACP+ provides significant improvements for an end-to-end path that includes a multiaccess hop shared by many status updating applications using ns3 simulations. (§ IV)

Related Work: The past five years have seen work on various aspects of the age of information; recent surveys can be found in [10], [16]. That said, there is limited systems research [7], [12], [14], [15] on ageing of information and its optimization in real-world networks. In [15], authors discuss the age of information (AoI) in real-networks where a source is sending updates to a monitor over different access networks. The key takeaway from that work is the need for an AoI optimizer that can adapt to changing network topologies and delays. The Age Control Protocol (ACP) was proposed in [12], [14]. ACP is a transport-layer solution that works in an application-independent and network-transparent manner. ACP attempts to minimize the age of information of a source update at a monitor connected over an end-to-end path on the Internet. Very recently, in [7] the authors proposed under the name of WiFresh a MAC-layer and an application-layer solution to ageing of updates over a wireless network. While both [14] and [7] look at ageing of updates on the Internet, they differ in their approach and scope. ACP is a transport layer solution that works by adapting the source generation rate without any specific knowledge of the access network or any network hop to the monitor, whereas, WiFresh is a scheduling solution designed for WiFi networks.

II. THE ACP+ CONTROL ALGORITHM

ACP+ is an improved version of ACP [14]. Just like ACP, it is a transport layer protocol that operates on the end-hosts. It uses UDP as a substrate and benefits from its unreliability feature to send updates from a source to a monitor over the Internet. Since ACP+ is quite similar to ACP, we provide a succinct summary of the algorithm and highlight the salient differences between ACP+ and ACP¹.

The ACP+ source appends a header to the update containing a *timestamp* field that stores the time it was generated. ACP+ suggests to the source the generation rate of updates. For this, it must estimate network conditions over the end-to-end path to the monitor. This estimation is enabled by having the ACP+ monitor send back an ACK to the ACP+ source for every received update. The ACK contains the timestamp of the update it is acknowledging.

Consistent with AoI freshness metrics, ACP+ discards an *out-of-sequence* packet at the monitor and an *out-of-sequence* ACK at the source. Algorithm 1 details the ACP+ control algorithm that is used to set the source's update rate λ_k at time t_k . The control algorithm executing at time t_k , uses an estimate of the time average update age ($\bar{\Delta}_k$) at the monitor and the time average of backlog ($\bar{B}_k$) calculated over the interval (t_{k-1}, t_k) , where k indexes the current control epoch.

¹The design principles and details about ACP can be found in [13], [14].

Algorithm 1 ACP+ Control Algorithm

```

1: INPUT:  $b_k, \delta_k, \bar{T}, B_k$ 
2: INIT:  $flag \leftarrow 0, \gamma \leftarrow 0$ 
3: while true do
4:   if  $b_k > 0 \ \&\& \ \delta_k > 0$  then
5:     if  $flag == 1$  then
6:        $\gamma = \gamma + 1$ 
7:       MDEC( $\gamma$ ):  $b_{k+1}^* = -(1 - 2^{-\gamma})B_k$ 
8:     else
9:       DEC:  $b_{k+1}^* = -1$ 
10:     $flag \leftarrow 1$ 
11:   else if  $b_k > 0 \ \&\& \ \delta_k < 0$  then
12:     INC:  $b_{k+1}^* = 1$ 
13:      $flag \leftarrow 0, \gamma \leftarrow 0$ 
14:   else if  $b_k < 0 \ \&\& \ \delta_k > 0$  then
15:     INC:  $b_{k+1}^* = 1$ 
16:      $flag \leftarrow 0, \gamma \leftarrow 0$ 
17:   else if  $b_k < 0 \ \&\& \ \delta_k < 0$  then
18:     if  $flag == 1 \ \&\& \ \gamma > 0$  then
19:       MDEC( $\gamma$ ):  $b_{k+1}^* = -(1 - 2^{-\gamma})B_k$ 
20:     else
21:       DEC:  $b_{k+1}^* = -1$ 
22:      $flag \leftarrow 0, \gamma \leftarrow 0$ 
23:   UPDATELAMBDA( $b_{k+1}^*$ )
24:   wait  $\bar{T}$ 

25: function UPDATELAMBDA( $b_{k+1}^*$ )
26:    $\lambda_k = \frac{1}{\bar{T}} + \frac{b_{k+1}^*}{\bar{R}\bar{T}\bar{T}}$ 
27:   if  $\lambda_k < 0.75 * \lambda_{k-1}$  then
28:      $\lambda_k = 0.75 * \lambda_{k-1}$  ▷ Minimum  $\lambda$  threshold
29:   else if  $\lambda_k > 1.25 * \lambda_{k-1}$  then
30:      $\lambda_k = 1.25 * \lambda_{k-1}$  ▷ Maximum  $\lambda$  threshold
31:   return  $\lambda_k$ 

```

ACP+ uses RTT(s) of updates for age estimation and maintains an exponentially weighted moving average (EWMA) $\bar{RTT}$ of measured RTT(s). RTT is calculated for every update whose ACK is received. It is the time between the generation of the update and reception of the corresponding ACK. The ACP+ source also keeps an estimate $\bar{Z}$ of the EWMA of the time elapsed at the monitor between reception of consecutive updates. This time between reception of consecutive updates is approximated by the source as the time elapsed between its reception of the corresponding ACK(s). As seen in line 26 of Algorithm 1, and explained later, $\bar{RTT}$ and $\bar{Z}$ are used to calculate the source update rate λ_k .

The length $\bar{T}$ of a control epoch is set as $\bar{T} = 10/\lambda_k$. This ensures at least 10 packets are sent by the source using the updated λ_k . The source updates $\bar{RTT}$, $\bar{Z}$ and $\bar{T}$ every time an ACK is received. At every control epoch $k > 1$, at time t_k , the ACP+ source calculates the differences $\delta_k = \bar{\Delta}_k - \bar{\Delta}_{k-1}$ and $b_k = \bar{B}_k - \bar{B}_{k-1}$.

At the source, ACP+ chooses an action u_k at the k^{th} epoch that targets a change b_{k+1}^* in average backlog over an interval of length $\bar{T}$ with respect to the k^{th} interval. The actions, may be broadly classified into (i) additive increase (INC), (ii) additive decrease (DEC) and (iii) multiplicative decrease (MDEC). MDEC corresponds to a set of actions MDEC(γ), where $\gamma = \{1, 2, \dots\}$.

The ACP+ source targets a reduction in average backlog over the next control interval in case either $\{b_k > 0, \delta_k > 0\}$

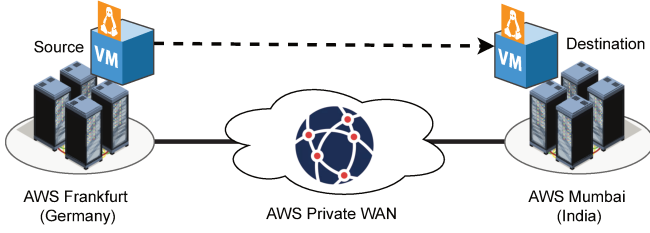


Figure 4: An illustration of the real experiment topology on the AWS E2 cloud network. The client machine (both ACP+/TCP) was in AWS Frankfurt, Germany, and the server was in AWS Mumbai, India. The instances were connected via the AWS Private WAN.

(line 4 in the algorithm) or $\{b_k < 0, \delta_k < 0\}$ (line 17). The ACP+ source targets an increase in average backlog over the next control interval in case either $\{b_k > 0, \delta_k < 0\}$ (line 11) or $\{b_k < 0, \delta_k > 0\}$ (line 14). Algorithm 1 summarizes how ACP+ chooses its action u_k as a function of b_k and δ_k to achieve the desired b_{k+1}^* . Next, ACP+ calls the function `UPDATELAMBDA` (line 23) that sets the rate at which the source must send updates to achieve the target backlog b_{k+1}^* (line 26). However, to restrict the unnecessary oscillations close to the age minimizing λ , we use `min` and `max` thresholds (lines 27-29) to restrict the range of λ_k .

The most significant change in ACP+ over ACP is in the function `UPDATELAMBDA`. ACP used a step size parameter κ in `(INC)` and `(DEC)`. Instead of ± 1 , the desired change in backlog was set to $\pm \kappa$; however, we found κ was remarkably difficult to set. For example, while in our real experiments with ACP $\kappa = 1$ worked well, in simulations with shorter paths, with small round-trip times but not so small propagation times, age control went haywire. For the latter, a small value of $\kappa = 0.25$ ensured proper updating of the update rate λ_k . The κ based update could also result in λ_k (calculated in line 26) becoming very small or even negative. To avoid this, we clamped the minimum λ_k to be at least one packet per round-trip time. Unfortunately, this resulted in high age in settings where multiple ACP paths shared a constrained access. ACP+ doesn't use such a minimum. Instead `UPDATELAMBDA` restricts the step change in λ . This, as shown in § IV, results in significant improvements in age achieved when a large number of ACP+ sources send updates over a shared multiaccess.

III. REAL-WORLD SETUP AND EXPERIMENTS

In this section, we describe the experimental setup and methodology, followed by an empirical comparison of ACP+ and the chosen congestion control algorithms with respect to the metrics of age, throughput and delay. The ACP+ client and server codes are available at [1].

Methodology: Figure 4 shows the real experiment topology. All our experiments over the Internet used two T2.micro instances in the AWS EC2 cloud network. Both instances are configured with one virtual CPU, 1 GB RAM and a 1 Gbps Ethernet link connected to the AWS private WAN. One of the instances was in the AWS Frankfurt (Germany) data-center, while the other was deployed in the AWS Mumbai (India)

data-center. Each instance ran a virtual machine with Ubuntu 18.04 LTS with Linux kernel version 5.3. We confirmed through periodic `traceroute` that the underlying network between our two chosen instances was served by the AWS private WAN.

We describe our measurement methodology next. For both the ACP+ and TCP experiments, we deployed the sender in AWS Frankfurt and the receiver in AWS Mumbai. For each chosen congestion control algorithm, we investigated the impact of different receive buffer sizes on the performance of the congestion control algorithms by changing `default` and `maximum` values of `r_mem` in the Linux kernel. The space available in the receiver buffer limits the maximum amount of bytes that any congestion control algorithm may send to the TCP receiver.

For the TCP experiments, we used `iPerf3` for packet generation and `Wireshark` for packet captures. To ensure that all algorithms saw similar network conditions we ran multiple iterations of ACP+, TCP BBR, TCP CUBIC, TCP Reno, TCP Vegas and TCP YeAH, in that exact order, one after the other. For each TCP variant in the stated order, we further ran different receive buffer settings. Each run of the experiment lasted 200 s. Considering that end-to-end RTT is ≈ 110 ms in our setup, TCP spends a majority of the transfer time in the steady-state phase.

Results: We show results from 40 runs each of ACP+, BBR-d1m1, BBR-d1m3, BBR-d5m5, CUBIC, Reno, Vegas and YeAH. For each run, we show the average age, throughput, and average delay (round-trip time). In the above list, we have BBR run with three different receiver buffer settings. BBR-d1m1 denotes the smallest default and maximum values of the receiver buffer (`r_mem`). In BBR-d1m3, the default is the same as BBR-d1m1 but the maximum is three times larger. Similarly, in BBR-d5m5 both the default and the maximum is five times that in BBR-d1m1. For all other TCP algorithms, the results are shown for a default and a maximum five times that of BBR-d1m1. In general, one would expect a larger receiver buffer to allow the TCP algorithm to have a larger number of bytes in flight as long as the network doesn't become the bottleneck.

Queue Waiting Delays Dominate: Figure 5 shows the impact of TCP segment lengths on delay. As is seen, segment length and delays are uncorrelated for all the TCP algorithms. This observation can be explained by the fact that the delays in the network are almost entirely because of the time spent in router queues awaiting transmission. The transmission times (propagation delays), which are about 20 ms, are a small fraction in comparison. It may be worth noting that the TCP segment lengths are chosen by the TCP algorithm and often change during a TCP session. In the figure, we show segment lengths averaged over a run.

Delay vs. Age: Figure 6 shows a scatter of (delay, age) for the chosen runs. We see that BBR-d5m5 sees both age and delays larger than the rest. Amongst the rest, from the figure, it is apparent that ACP+ achieves delays and ages smaller than all algorithms other than BBR-d1m1. BBR-d1m1 achieves a

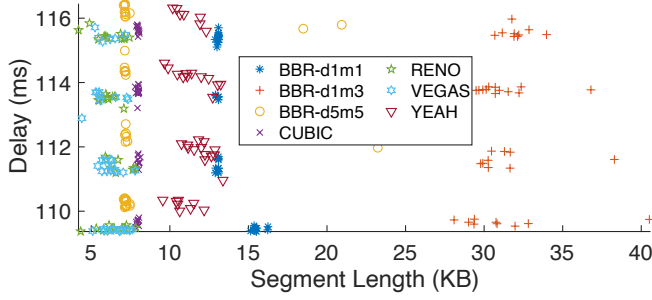


Figure 5: TCP segment length vs. delay obtained for the runs of the different algorithms.

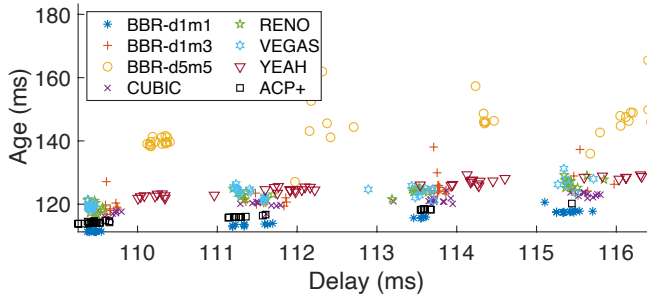


Figure 6: Delay vs. age for the different runs of the chosen algorithms.

slightly smaller age than ACP+.

In fact, the age and delay achieved by BBR-d1m1, averaged over all runs, are 114.5 ms and 112.33 ms, respectively. The corresponding values for ACP+ are 115.5 ms and 110.79 ms. The next smallest age is achieved by CUBIC and is ≈ 121 ms. Reno, Vegas and BBR-d1m3 achieve higher ages than CUBIC, with YeAH achieving the highest age of about 125 ms among them. BBR-d1m4, BBR-d1m5 and BBR-d5m5 achieve ages larger than 140 ms. Only BBR-d5m5 is shown.

ACP+ vs. BBR-d1m1: Before we delve further into the relative performances of ACP+ and BBR-d1m1, let's consider Figure 7 in which we show the (throughput, age) values achieved by the different algorithms. We omit BBR-d5m5 from the figure as it resulted in high age values (average larger than 140 ms) and also did not yield very good throughput. BBR-d1m3 achieves the highest throughput. In fact, its throughput of about 200 Mbps is twice the next highest value of about 110 Mbps achieved by BBR-d1m1. The average age when using BBR-d1m3 is 123.5 ms in contrast to the 114.5 ms obtained when using BBR-d1m1.

Interestingly, the throughput obtained by ACP+ is a low of 0.77 Mbps in contrast to 110 Mbps obtained using BBR-d1m1 ($\approx 141\times$ the ACP+ throughput). This stark difference is partly explained by the segment² sizes used by BBR-d1m1, on an average about 14 KB, in comparison to the constant 1024 byte payload of an ACP+ packet. This difference still leaves an unexplained factor of about 10. This is explained by an average inter-ACK time of 10.4 ms for ACP+ in comparison

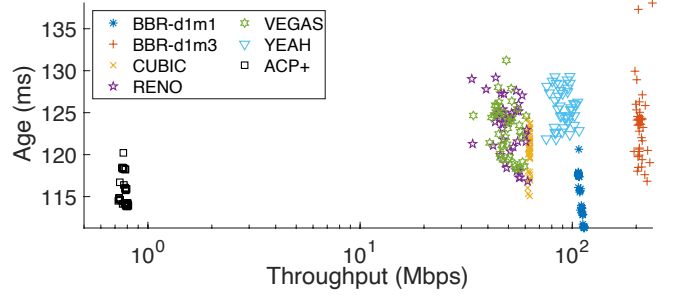


Figure 7: Throughput vs. age for the different runs of the chosen algorithms.

to a much smaller 1.16 ms for BBR-d1m1 that results from BBR-d1m1 attempting to achieve high throughputs.

To summarize, ACP+ results in an average age of 115.5 ms, an average delay of 110.79 ms, an average throughput of 0.77 Mbps and an inter-ACK time of 10.4 ms. The corresponding values for BBR-d1m1 are 114.5 ms, 112.33 ms, 110 Mbps and 1.16 ms. *ACP+ achieves an almost similar age as BBR-d1m1, however, at a significantly lower throughput.* The similar age at a much larger inter-ACK time is explained by the fact (observed in our experiments) that while a very low or high rate of updates results in high age, age stays relatively flat in response to a large range of update rates in between. It turns out that ACP+ tends to settle in the flat region closer to where increasing the rate of updates stops reducing age. This much reduced throughput of ACP+ is especially significant in the context of shared access, allowing a larger number of end-to-end ACP+ flows to share an access without it becoming a bottleneck.

The BBR Puzzle: What could explain the low age achieved by BBR-d1m1? We observe that the average delay of 112.33 ms when using BBR-d1m1 is the same as that obtained by a *Lazy* (introduced in [14]) status updating protocol we ran alongside the others, which sends an update once every round-trip time. One would expect *Lazy* to achieve a round-trip time of RTT_{base} (see Figure 2a). This tells us that BBR-d1m1's flow on an average saw an RTT of RTT_{base} . While it obtained a low throughput of 100 Mbps, it seems to have kept the pipe full enough. This low throughput was an accidental consequence of the receiver buffer size settings of BBR-d1m1, which disallowed the congestion control algorithm to push bytes into the network at a larger rate. The higher throughput achieved by BBR-d1m3, as observed earlier, came with a higher age, however.

IV. SIMULATIONS SETUP AND RESULTS

We used the network simulator ns3³ together with the YansWiFiPhyHelper⁴. The base network topology used in our simulations is shown in Figure 8. We show results for when source nodes are spread uniformly and randomly over an area of 20×20 m². We chose the number of sources from the set

³<https://www.nsnam.org/>

⁴https://www.nsnam.org/doxygen/classns3_1_1_yans_wifi_phy.html

²Recall our assumption that every new segment contains a fresh update.

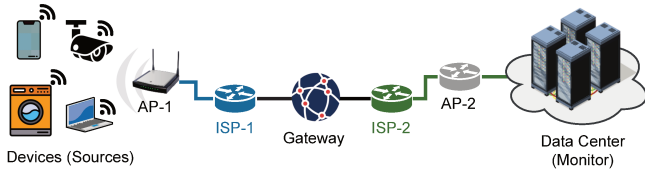


Figure 8: Simulation Setup

$\{1, 6, 12, 24, 48\}$. The channel between the source and AP-1 was log-normally distributed with a standard deviation of 12 and a path loss exponent of 3. The WiFi link rate was set to 12 Mbps and that of the P2P links was set to 6 Mbps.

We compare the performance of ACP+ to *Lazy*, which as mentioned before, is a conservative status updating mechanism that sends one update per $\overline{RTT}$ and maintains an average backlog of 1 update in the network for a given source.

Figure 9a shows that ACP+ achieves a smaller age per source than *Lazy*. The improvements are especially significant when a large number of sources share the access to AP-1. That ACP+ is able to achieve smaller ages can be understood via the average backlog per source when using ACP+ and *Lazy*, which is shown in Figure 9b. When we have just one source, ACP+ tries to fill each queue in the network with an update. This results in a larger backlog and a lower age in comparison to *Lazy*, which achieves a backlog of just 1 update. However, as the number of sources increases, while *Lazy* continues to maintain a backlog of 1 per source, ACP+ reduces it. The backlogs obtained are 3.23, 1.39, 0.91, 0.57, 0.34, respectively, for 1, 6, 12, 24, 48 sources.

The ACP+ backlogs when we have a large number of sources are not only much smaller than *Lazy*, it turns out that they are not too far from an ideal scheduling mechanism that schedules updates from the sources in a round-robin manner. For simplicity, ignore the difference in the link rates of the WiFi and P2P links. Also, assume that no packets are dropped due to channel errors over WiFi. A round-robin scheduler would keep six updates in transit of the source when we have just one source. This would result in a backlog of 6. It would schedule six sources one after the other in a manner such that a round of scheduling would lead to six packets in the six queues from the six different sources, resulting in an average backlog of 1 per source. Similarly, for when we have 12, 24, 48 sources, we would see backlogs per source of $1/2, 1/4, 1/8$, respectively. ACP+ sees larger backlogs than these, at least partly because of packet collisions over the WiFi access, which results in larger delays in the WiFi hop.

ACP+'s good adaptation to an increase in the number of sources is also seen in the fact that the RTT doesn't increase much as the number of sources increase. This is unlike *Lazy* which sees big increases in RTT. While ACP+ results in RTT of 5.5, 7.1, 7.9, 9, 10.4 ms, respectively, for 1, 6, 12, 24, 48 sources, *Lazy* sees RTT of 5.5, 6.3, 11.8, 26.3, 61.1 ms.

ACP+ is *age fair* with Jain's fairness index [6] reducing from about .99 to .89 as we increase source density from 6 to 48 nodes. As regards ACP, in such settings it had earlier been evaluated to be as bad as *Lazy* at age control.

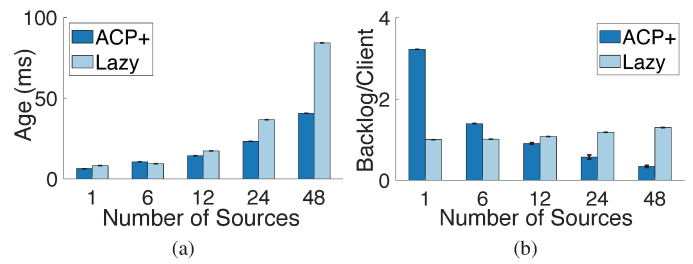


Figure 9: Average (a) source age and (b) source backlog

V. CONCLUSIONS

In this paper, we detail the design of ACP+, an improved version of the transport layer age control protocol ACP. We evaluated ACP+ and a mix of loss-based, delay-based and hybrid TCP congestion control algorithms over an inter-continental path connecting AWS data-centers in Mumbai, India and Frankfurt, Germany. We showed that ACP+ achieves a small age at a very low throughput. We also evaluated ACP+ in a setting where multiple sources send their updates using ACP+ over a shared multiaccess. We showed that the different ACP+ flows coexist well with each other and the sources see ages much lower than the old ACP and *Lazy* updating.

REFERENCES

- [1] ACP+: Improved Age Control over the Internet. <https://github.com/tanyashreedhar/AgeControlProtocolPlus>, 2021.
- [2] A. Baiocchi, A. P. Castellani, and F. Vacirca. YeAH-TCP: Yet Another Highspeed TCP. In *Proceedings of PFLDnet*, volume 7, pages 37–42, 2007.
- [3] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.
- [4] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. BBR: Congestion-based congestion control. *Queue*, 14(5):20–53, 2016.
- [5] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.
- [6] R. K. Jain, D.-M. W. Chiu, W. R. Hawe, et al. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, 1984.
- [7] I. Kadota, M. S. Rahman, and E. Modiano. WiFresh: Age-of-Information from Theory to Implementation, 2020.
- [8] S. Kaul, R. Yates, and M. Gruteser. Real-Time Status: How Often Should One Update? In *Proc. IEEE INFOCOM Mini Conference*, 2012.
- [9] L. Kleinrock. Internet congestion control using the power metric: Keep the pipe just full, but no fuller. *Ad hoc networks*, 80:142–157, 2018.
- [10] A. Kosta, N. Pappas, and V. Angelakis. Age of information: A new concept, metric, and tool. *Foundations and Trends in Networking*, 12(3):162–259, 2017.
- [11] Network Working Group. RFC 5681: TCP Congestion Control. <https://tools.ietf.org/html/rfc5681>, 2009.
- [12] T. Shreedhar, S. K. Kaul, and R. D. Yates. Poster: ACP: Age Control Protocol for Minimizing Age of Information over the Internet. In *MOBICOM*, pages 699–701. ACM, 2018.
- [13] T. Shreedhar, S. K. Kaul, and R. D. Yates. ACP: An End-to-End Transport Protocol for Delivering Fresh Updates in the Internet-of-Things. *CoRR*, abs/1811.03353, 2019.
- [14] T. Shreedhar, S. K. Kaul, and R. D. Yates. An Age Control Transport Protocol for Delivering Fresh Updates in the Internet-of-Things. In *2019 IEEE 20th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 1–7, 2019.
- [15] C. Sönmez, S. Baghaee, A. Ergişi, and E. Uysal-Biyikoglu. Age-of-Information in Practice: Status Age Measured Over TCP/IP Connections Through WiFi, Ethernet and LTE. In *2018 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, 2018.
- [16] R. D. Yates, Y. Sun, D. R. B. III, S. K. Kaul, E. Modiano, and S. Ulukus. Age of Information: An Introduction and Survey. *CoRR*, abs/2007.08564, 2020.