MEASURING PERFORMANCE OF WEB PROTOCOL WITH UPDATED TRANSPORT LAYER TECHNIQUES FOR FASTER WEB BROWSING

Ziaul Hossain[1] & Gorry Fairhurst[2]

[1] Department of Computing, University of Fraser Valley, Abbotsford, BC, Canada
[2] School of Engineering, University of Aberdeen, UK

ABSTRACT

Popular Internet applications such as web browsing, web video download or variable-rate voice suffer from standard Transport Control Protocol (TCP) behaviour because their transmission rate and pattern are different from conventional bulk transfer applications. Previous works have analysed the interaction of these applications with the congestion control algorithms in TCP and proposed Congestion Window Validation (CWV) as a solution. However, this method was incomplete and has been shown to present drawbacks. This paper focuses on the 'newCWV' which was proposed to address these drawbacks. newCWV depicts a practical mechanism to estimate the available path capacity and suggests a more appropriate congestion control behaviour. These new modifications benefit variable-rate applications that are bursty in nature, with shorter transfer durations. In this paper, this algorithm was implemented in the Linux TCP/IP stack and tested by experiments, where results indicate that, with newCWV, the browsing can get 50% faster in an uncongested network.

KEYWORDS

Network Protocols, HTTP, TCP, Congestion Control, newCWV, Bursty TCP traffic

1. INTRODUCTION

With the development of the Internet, many applications have gained enormous popularity. Email, VoIP applications, File sharing etc., each have taken a share of the total Internet traffic, but the largest share is currently Web browsing applications with almost 70% of the total traffic across the Internet [1]. Web traffic uses TCP and HTTP [2] [3] protocols for request and delivery of the web page content. There had already been numerous developments across these protocols with a view to improve the performance without proposing any replacement of these standards.
Many of these updates to TCP focus on the congestion control mechanism as this technique define how much data can be transferred from the sender to receiver for an application flow. *cwnd* ensures that the sending rate of a flow is comparatively safe for the other flows that share the same bottleneck along the path between the sender and the receiver. But the focus in TCP improvements was primarily for bulk file transfers only. These modifications are not suitable for HTTP like traffic, which is 'bursty' (variable rate traffic with irregular intervals) in nature. This problem has been reported earlier and several attempts had also been made to realise a solution [4][5] . Unfortunately, these solutions were still conservative and lack proper measurement of the available path capacity to set the congestion window (*cwnd*) – the most important parameter of

David C. Wyld et al. (Eds): CCNET, AIMLA, CICS, IOTBS, NLTM, COIT - 2023
pp. 01-16, 2023. CS & IT - CSCP 2023 DOI: 10.5121/csit.2023.130401

the congestion control mechanism. This shortcoming limits the performance of bursty applications like HTTP.

A newer method has been developed termed as 'newCWV' [6]. When sending bursty or rate-limited traffic, this new method allows a sender to estimate the path capacity more accurately and set the *cwnd* to an appropriate value accordingly. The rationale for newCWV is presented briefly and the algorithm is explained elaborately in [6]. But there is a void in validating the arguments and also in measuring the expected application performance improvement with this proposal.
This paper aims to explain the motivation behind developing newCWV in detail and then analyse the web traffic transfer durations in order to measure improvements. Particular focus of this paper is on the implementation and integration of newCWV into the Linux and run experiments to support the theory. Through experiments, this paper shows that, when HTTP-like traffic uses 'newCWV', there is significant gain in performance compared to conventional TCP. Web browsing can proceed in approximately 50% faster rate in an uncongested network with the newCWV.

Section 2 of this paper explains the bursty behaviour of the HTTP traffic, the basics of TCP congestion control and the state of the art to set the background. Then, section 3 explains the modification specified in [6]. Section 4 summarises the experiment and presents the results with discussion. Finally, section 5 concludes the findings.

2. BACKGROUND

To understand the problem of transporting HTTP-like traffic with unmodified TCP, the behaviour of these protocols needs to be examined. This section explores the bursty behaviour of the HTTP protocol, the conventional congestion control of TCP and explains the interactions when these are used together.

2.1. Nature of HTTP traffic

The HTTP web traffic is naturally bursty in nature. Burstiness could be termed as a property of an application where the traffic is generated in a random manner at different rates over its running time. This could be characterised as periods of inactivity separated by periods when the chunks of data are downloaded. [7] showed that popular HTTP applications such as Web video (YouTube), Maps (Google Maps), Remote Control (LogMeIn) all send data in the downstream at variable rate with spikes up to 400KB/s, separated by periods with no activity. This burstiness is caused by the HTTP request pattern in the client/user application. Besides application behaviour, small-scale burstiness can also be caused by TCP.

[8] showed that TCP self-clocking, combined with network queuing delay (due to packets of the same flow or cross traffic) can shape the packet inter-arrivals of a TCP flow resulting in an ON-OFF pattern. With a view to modelling the inactivity (OFF) periods of the web clients, [9] showed that the OFF duration could range from a few seconds to many tens of seconds, with a probability of 80% and 10% respectively, which causes burstiness of a TCP connection when requesting content from the server.

The cited papers all agree that burstiness has become a common pattern for HTTP traffic. A simple experiment was run that captured packets while accessing a webpage from a browser to capture this bursty behaviour.

Figure 1 shows the resulting burstiness. In this capture, the chunks of data are the results of HTTP GET requests made by the client. It is visible that there are considerable inactive periods between consecutive bursts.

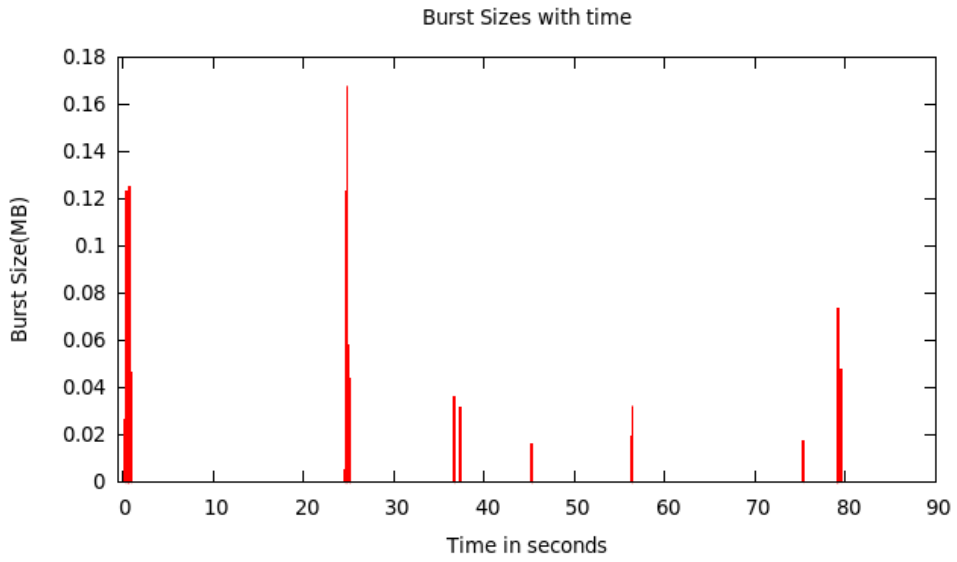

Figure 1. Bursty traffic pattern of HTTP web for a single web page

2.2. TCP Congestion Control Mechanisms

After being first standardized in 1981 by the Internet Engineering Task Force (IETF), TCP was enriched by a series of developments to face numerous challenges occurring in the underlying network. [10] provided a roadmap that described many of these changes.

A basic operating procedure of TCP is explained in the remainder of this subsection.
A TCP sender uses a parameter called the congestion window, or '*cwnd*'. This is initialised to the Initial Window (IW) size. It determines the amount of data that can be sent to the receiver while before receiving an acknowledgement from the receiver. The value of the cwnd is important, as it ultimately dictates the transfer rate and eventually the response time for an HTTP connection. TCP uses four congestion control algorithms to set the value of this ''cwnd that were specified by RFC2001, RFC2581 and RFC5681 [11][12][13]. They are Slow Start, Congestion Avoidance, Fast Retransmit and Fast Recovery.

Slow Start: In the Slow Start phase, a TCP sender sends data limited by the cwnd value and waits for Acknowledgement (ACK) packet from the receiver. Upon receiving an ACK, the value of the *cwnd* is increased by one segment. So, if a sender sent 4 segments at first (because *cwnd* = 4), and then receives 4 ACKs for these segments, then after increasing *cwnd* for each segment, the final *cwnd* value will be 6, and 6 segments can be sent. As a result of this cumulative increase, the cwnd increases using an exponential function. This continues until it reaches the Slow Start threshold (ssthresh) or the sender discovers congestion or encounters a loss.

Congestion Avoidance (CA): When the *cwnd* reaches the ssthresh, a limit is imposed on the increase of the *cwnd*. After this point, the size of the *cwnd* is only increased by one segment in one RTT. For example, if 8 segments are sent altogether, then when the 8 segments are acknowledged, the *cwnd* becomes 4 only. This corresponds to a slower linear growth of *cwnd*.

Fast Retransmit: When a packet is lost, the subsequent packets are received out of order at the receiver. When this happens, the receiver sends duplicate ACK packets when each segment is received. All the ACK packets acknowledge the same sequence number. Upon receiving the first duplicate ACK, the sender does not immediately take action, but waits to see if this is a re-ordering issue or a packet loss. When it receives a series of duplicate ACKs equal to the DupACK threshold (3 as currently standardised), the sender TCP retransmits the segment, and resets the congestion state.

Fast Recovery: When a segment is lost, rather than setting the *cwnd* to the lowest value and then send packets in sequence, it is assumed that a better approach would be to start from an intermediate value so that the flow is not badly affected. So, after a lost segment has been transmitted, CA is performed instead of Slow Start. The *cwnd* is set to (ssthresh + DupACK) segments. This is to virtually inflate the network. Since DupACKs packets have been received, this means these packets have left the network (i.e. had been received successfully). With each further DupACK, the *cwnd* is incremented by one segment. When a new ACK is received, the *cwnd* is reset to ssthresh and CA is resumed.

Selective ACK (SACK): SACK acknowledges reception of out of sequence packets. This helps avoid retransmission of already received packets. Using SACK, the receiver appends a TCP option in the DupACK header that contains a range of non-contiguous data that have been received. This allows the sender to resend only the packets that were missing from the flow. Support for SACK is negotiated at the beginning of a TCP connection; it can be used if both ends support the SACK option. [14] showed that NewReno with SACK enabled, requires fewer packet transmissions in the First Recovery phase, reduces unnecessary duplicate transmission and avoids waiting time.

2.3. TCP Variants

Different variants of TCP have evolved using combinations of these algorithms and with modifications to control the data flow and to improve response to network congestion. When a loss is detected, the TCP sender takes measures to control the flow of further packets by reducing the sending rate. Different TCP variants such as Tahoe, Reno, NewReno, which act differently in response to detected congestion.

Tahoe used Slow Start, Congestion Avoidance and Fast Retransmit. A problem with Tahoe is that restarted from the initial *cwnd* value after each packet loss. This resulted in lower throughput. To deal with this, Reno implemented Fast Recovery. This effectively recovered a of single packet loss within a window. If two or more packets were dropped in the same window, the sender was forced to timeout and restart in Slow Start. To overcome this problem, NewReno uses a modified Fast Retransmission phase based on the research [15][16]. This starts when a packet is lost and ends when a Full ACK is received, which means that all the packets transmitted between the lost packet and the last packet have been successfully received. However, if there are multiple packet-drops, then the sender will acknowledge a packet that has a lower sequence number than the last transmitted packet. This is a Partial ACK, and in this case, the lost packet is retransmitted immediately without waiting for receiving duplicate ACKs. This avoids a possible timeout. This ensures better performance than Reno, but may need to restart after a timeout if many packets are dropped from the same window.

When there are multiple losses, SACK provides better performance by enabling the receiver to inform the sender when there are multiple packet losses. A SACK block indicates a contiguous block of data that has been successfully received. The segment just before the first block and the

gap between any two consecutive blocks denote lost segments (more accurately, these are segments for which there is no acknowledgment). When the sender receives a SACK option, it can find out which segments may be lost and retransmits them. SACK is widely implemented in the current Internet, usually in combination with NewReno [10]. Therefore NewReno with SACK is considered as the standard congestion control for TCP.

Figure 2 shows the cwnd evolution for different variants with a 50ms path delay. It can be noticed in this figure that after some variation during the early stage (about 3s), all the variants reach a steady state with a similar behaviour. At time 1s, all variants start in the Slow Start phase and increase the cwnd exponentially until ssthresh is reached or after it suffers a loss. When this happens, Tahoe reduces its *cwnd* to an initial value (IW) and resumes in Slow Start. Once it reaches ssthresh, the sender enters the CA phase (linear increase of cwnd). Other variants enter Fast Retransmission and Fast Recovery phase, where the *cwnd* progresses almost linearly.

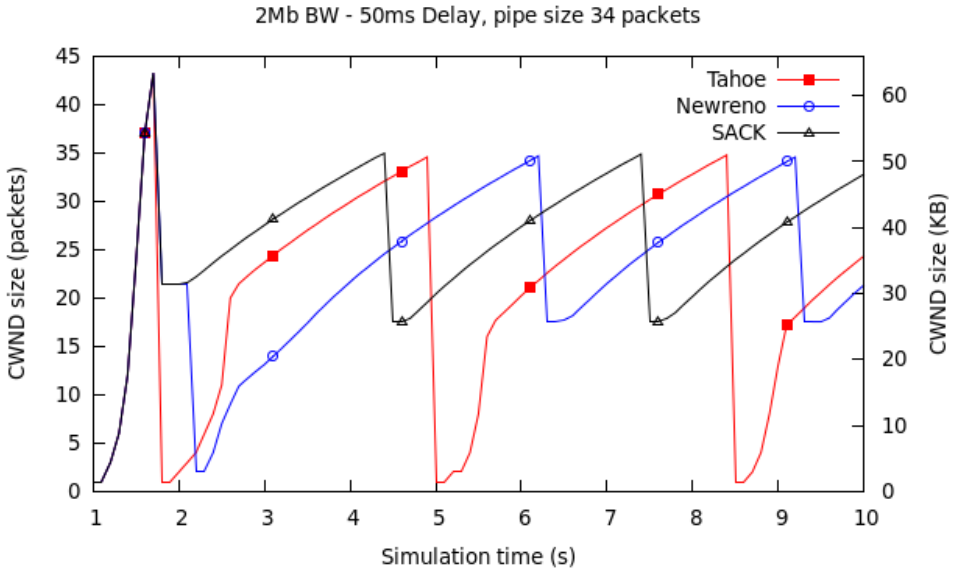

Figure 2. Congestion Window evolution with time during the simulation

Whenever packet loss occurs (at 1.8s, 5s and 8.5s for Tahoe) Tahoe enters the Fast Retransmit phase and retransmits the lost packet. The *cwnd* is set to the restart window (RW) (normally 1 segment) and continues in the Slow Start phase.

NewReno and SACK enter the Fast Recovery stage (at 1.8s for the first loss) where the ssthresh is set to a new value that is half the size of the unacknowledged data and then *cwnd* is set to ssthresh. At 2s, to more losses cause NewReno to fail to recover and eventually the sender is forced to enter the Slow Start phase. SACK was successful in recovering and eventually followed the CA until it faced another loss. Later, during the simulation, NewReno successfully recovered using Fast Retransmit and Fast Recovery. Then it moves to the CA phase.

2.4. TCP CWV

Standard TCP congestion control required that when an application is idle for a period greater than the Retransmission Timeout (RTO), the *cwnd* is reset to a small value. So, the next burst of data requires the sender to re-enter the Slow Start phase from this small value. Several RTTs may

be consumed before the previous sending rate is again achieved. This approach is too conservative, in that it fails to use available capacity. For a bursty application, this scenario is quite common where each burst is separated by an idle period. As a result, the application performance suffers from this conservative behaviour of TCP.

Figure 3 explains the situation as a diagram. After RTO, the cwnd (red bold solid line) drops to the RW. Then it takes long time to grow back for the next burst. So, the net burst is unnecessarily delayed while the path capacity might have been enough to transmit the burst in shorter RTTs.

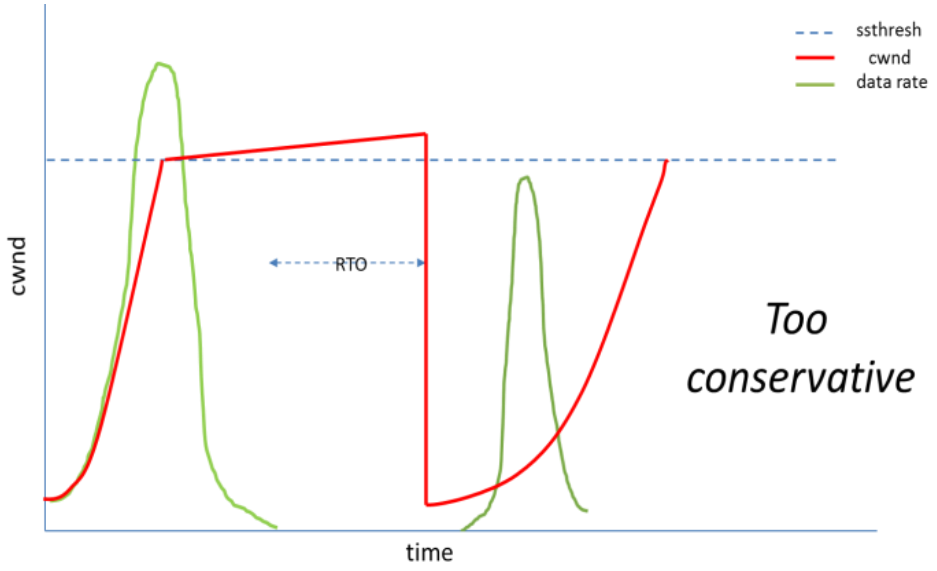

Figure 3. Reducing the cwnd to a low value of RW makes it overly conservative for idle period

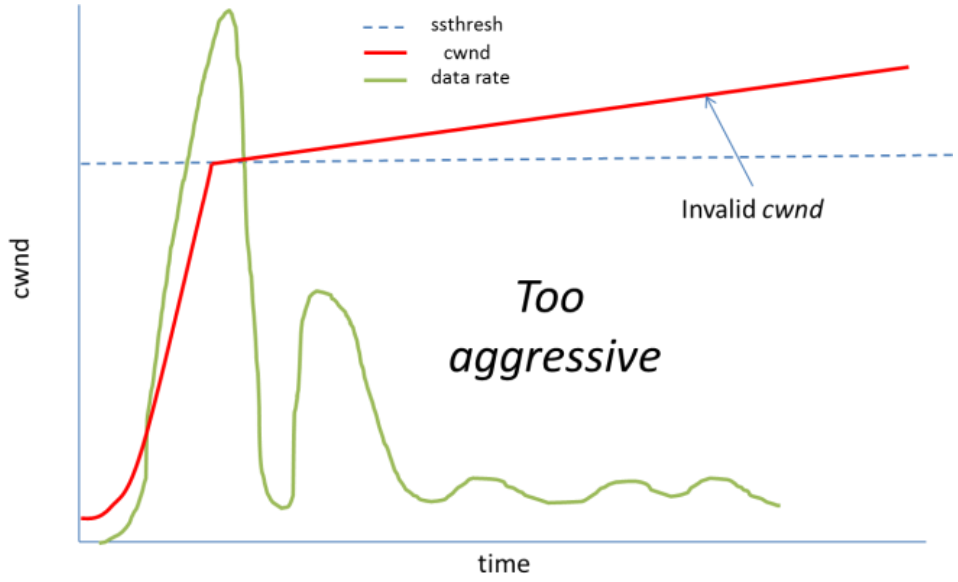

Figure 4. Increasing the cwnd during the application limited period makes it invalid

On the other hand, during an application-limited period, a Standard TCP sender continues to grow the *cwnd* for every received acknowledged packet (ACK), allowing the cwnd to reach an

arbitrarily large value. However, in this case the packet probes along the transmission path are sent at a lower rate than permitted by *cwnd*, so the reception of an ACK does not actually provide evidence that the network path was able to sustain the transmission rate reflected by the current cwnd. The *cwnd* is called 'invalid'.

Figure 4 explains such a scenario. The actual path capacity which may be significantly lower than the *cwnd*, can be mistaken.

If an application with an invalid cwnd were to suddenly increase its transmission rate, the sender would be allowed to immediately inject a significant volume of additional traffic into the network. This could lead to severe network congestion, potentially harming other flows that share a common bottleneck.

TCP Congestion Window Validation (TCP-CWV), was first specified in RFC 2861 [4], was proposed as an experimental standard by the IETF. The intention was to find a remedy for the problems imposed by TCP when used by a bursty application. TCP-CWV changed how cwnd is updated and is to be used during an idle or application-limited period.

TCP-CWV modified the congestion control algorithm of standard TCP during an application-limited period when the cwnd had not been fully utilised for a period larger than an RTO.
During an idle period, which is greater than one RTO, TCP-CWV reduced cwnd by half for every RTO period. This is equivalent to exponentially decaying cwnd during the idle period compared to reducing the cwnd in a single step with standard TCP. This is common traffic pattern for bursty applications to have an idle period in the order of seconds – which could be larger than a few RTOs worth of time. As a result, TCP-CWV ultimately reduces to RW and causes problem like standard TCP.

Another recommendation of CWV was to set the *cwnd* according to (*w_used*+*cwnd*)/2 for each RTO period that does not utilise the full *cwnd*, where *w_used* is the maximum amount that has been used since the sender was last network-limited/cwnd-limited. This avoids a growth of cwnd to an invalid value; it can cause the cwnd to reduce to a value that is close to the current application rate.

This results in two problems:
First, the *cwnd* should reflect the network capacity for a flow and control the amount of data that the network could sustain. However, CWV tends to set the cwnd according to the traffic pattern and application rate - only seeking to be conservative in use of network capacity. As a result, the *cwnd* is set to a lower value that is more conservative than when using standard TCP, which would have allowed larger bursts.

Secondly, CWV used w_used, the amount of data that has been sent by the application, but not yet acknowledged. In an application-limited period where the application is not using the allowed path capacity, w_used is does not reveal the available capacity. According to this approach, the *cwnd* is set to a value that is determined by the application's sending rate in the last RTO period (last few RTTs), rather than the network capacity. This impacts the application performance where the subsequent bursts are to be rate-limited and would take longer to complete. So, this should not be regarded as the available path capacity for the TCP flow that is recoded in the *cwnd*.

In summary, when TCP-CWV was specified in 2000, it identified a need to change the way TCP responded for bursty applications but failed to offer a complete solution.

3. TCP NEWCWV: MODIFICATION FOR HTTP-LIKE TRAFFIC

When newCWV was standardised in 2015, it introduced a variable called 'pipeACK' that was used to measure the acknowledged size of the network pipe. The pipeACK variable is considered as a safe bound for the capacity available to the sender since this represents the actual amount of data that was successfully transmitted in an RTT from the sender to the receiver. This variable can be computed by measuring the volume of data that have been acknowledged by the receiver within the last RTT.

The pipeACK is used to determine if the sender has validated the cwnd. The sender enters the non-validated phase when:

$$pipeACK < \frac{1}{2} \times cwnd$$

newCWV also defined a new phase. A sender was allowed to use the cwnd for a period (5 minutes), called the Non-Validated Period (NVP). During the NVP, the *cwnd* is preserved. The reason for storing the *cwnd* for several minutes because it is the default server timeout for TCP connection.

In summary, newCWV brought stability for both phases of rate-limiting period and application limiting period for HTTP like traffic. An algorithm was proposed and implemented in the Linux Kernel module, which was used to verify the effectiveness of this modification in the next section.

4. EXPERIMENT, RESULTS & DISCUSSION

This section first describes the network emulation used to explore the behaviour of newCWV. Then presents the findings in different scenarios with possible explanations for such behaviour.

4.1. Experimental Setup

A network emulation method was chosen to conduct the experiments because this enables a real implementation of network protocols to be tested in a controlled environment. The test bed used a dumbbell topology representing a single network path bottleneck (refer to Figure 5).

Client 1 and Server 1 were used to benchmark the newCWV behaviour for the main traffic (either HTTP web or HTTP streaming content). Another server (client 2, server 2) was used to inject cross traffic (in this case a large file transfer using FTP) into a shared network bottleneck. All servers ran Linux kernel versions 3.12, and the clients were running 3.8 or greater.

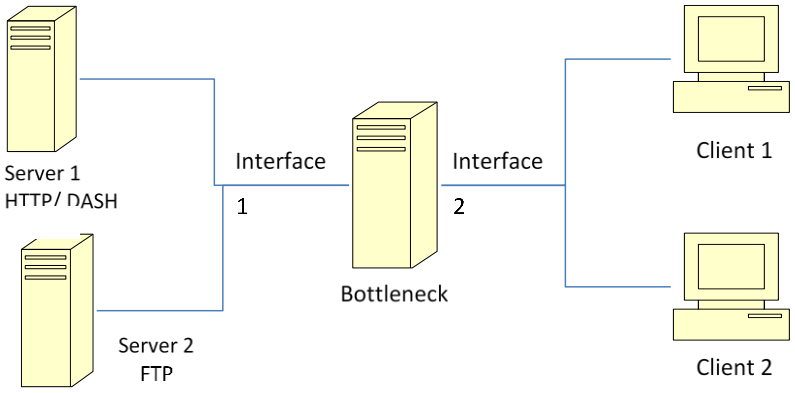

Figure 5. Experiment topology: the bottleneck router imposed a fixed bandwidth and delay between the client and server

Server 1 acts as a web server or streaming server that used standard TCP (NewReno with SACK). It was installed with the newCWV Loadable Kernel Modules (LKM) for Linux, the traffic generators and iproute2 utilities (to enabling pacing when required) allowing this to be chosen at the start of each experiment.

The experiments are run across a range of time intervals that represent values that range between HTTP response bursts (idle periods) and for HTTP response sizes larger than a particular value (Burst size after idle). The results obtained from multiple iterations of these experiments are averaged to measure the completion time of the HTTP/TCP connections for different combinations of idle periods and burst sizes.

The comparison plots, shown in the results section, present the ***improvement in burst transfer time*** (less time required for transmission) when newCWV is used compared to using a standard TCP (NewReno with SACK). The performance gain in transfer time (% improvement) is calculated by taking an average of the transfer gain over all bursts. The transfer gain was calculated by the following formula, where time taken in NewReno/SACK is ***Tr*** and time taken for a burst with newCWV is ***Tc***:

$$\text{Gain (in percentage)} = (Tr - Tc) / Tr \times 100$$

The gain can be positive where the burst is transmitted faster or negative when a particular HTTP response takes longer to transmit due to loss. A positive average of all these values indicates an overall gain in performance – the higher the value, the better.

The table below (Table 1) summarizes the experiment parameters:

Table 1: Experiment Parameters

Parameter	Value
TCP Initial Window (IW)	3 Segments
Ssthresh sharing	NO
Bottleneck Bandwidth	2 Mbps
Delay / RTT	200 ms
HTTP Generator	Tmix tool
Linux Kernel	3.12
No of HTTP connections	3151
Total Data analysed	7.68 GB
Average Transfer rate	700 kbps
Iterations with same parameters	5

4.2. Comparing performance over an uncongested path

To understand the effect of newCWV in a non-congested scenario, experiments were run with no bandwidth limit at the bottleneck router; only a link delay of 200 ms was applied. There was no cross-traffic and no rate limit was applied. Figure 6 below presents the performance improvement of newCWV compared to NewReno, plotted burst sizes vs. different idle periods.

The improvement is visible in this figure (Figure 6). A newCWV sender transfers a burst in 37-62% less time than NewReno. Larger improvements are achieved for the higher burst sizes, as expected; about 10% more improvement is achieved for bursts of 80KB (60%) than 5KB bursts (50%). While standard TCP reduces its cwnd after an idle period, newCWV retains a larger cwnd and is able to transfer the burst in less time, saving several RTTs – an approximate average of 50% improvement suggests newCWV requires half the RTTs compared to NewReno.

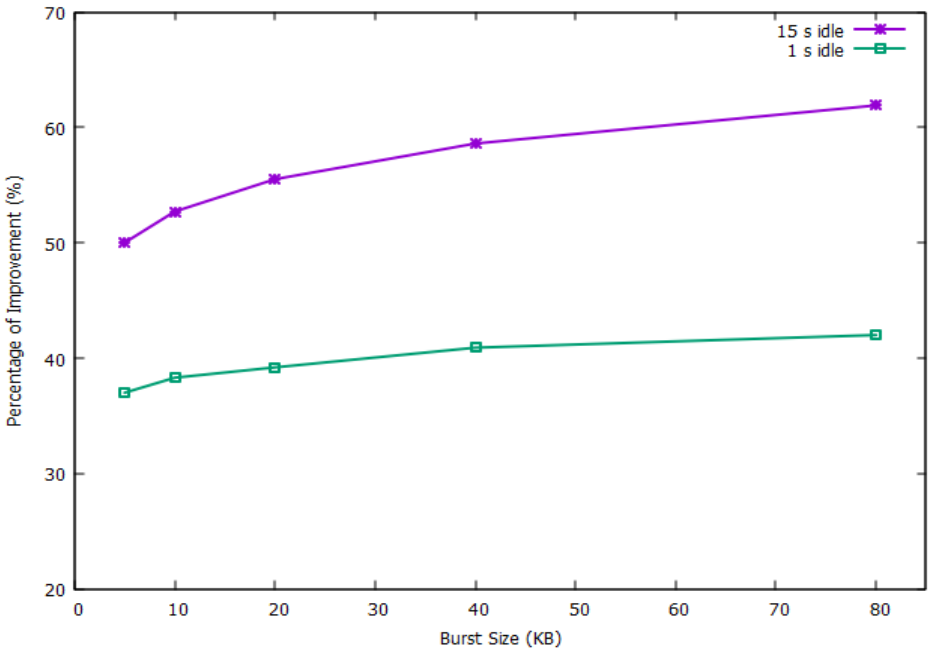

Figure 6. Performance Improvement of HTTP traffic shown when newCWV is used instead of NewReno over different burst sizes and idle periods.

Another interesting fact is that for a same size of burst, for example a 40 KB burst, would encounter almost 20% more improvement in performance when the idle period is larger. So, it shows that for real-life web browsing traffic, even if the idle period is high newCWV will support more traffic than the conventional TCP.

In short, for an uncongested scenario (as may be expected in a LAN), newCWV shows improved performance over standard TCP.

4.3. Comparing performance in a congested path

To test the effectiveness of newCWV in an Internet context, a bottleneck of 2 Mbps was set with a finite router buffer (30 KB). The path MTU was 1500 B, which ensured a maximum of 20 segments to be queued in the buffer. The newCWV protocol still shows improvement over standard TCP, which now varies from 10-35% over the idle period – burst size domain (shown in Figure 7).

While in the previous scenario, there were no other traffic, the performance improved more for higher burst sizes. However, in this congested scenario, the trend is somewhat opposite: Higher burst sizes offer less improvement. In the case of the idle period comparison, the similarity remains, where a larger idle period increases the improvement as in the previous non-congested scenario.

For bursts larger than 5 KB (larger than an IW of 4 KB), it takes about 25-33% less time on average for a transfer with an idle period. A larger improvement is demonstrated around 35% with newCWV, but the advantage diminishes for larger burst sizes (for 40 KB or 80 KB), because it encounters higher loss. The newCWV sender is prone to a higher loss rate for larger bursts. These bursts can appear either at the beginning of the TCP connection or after an idle period.

Figure 8 confirms that the number of losses is higher when using newCWV.

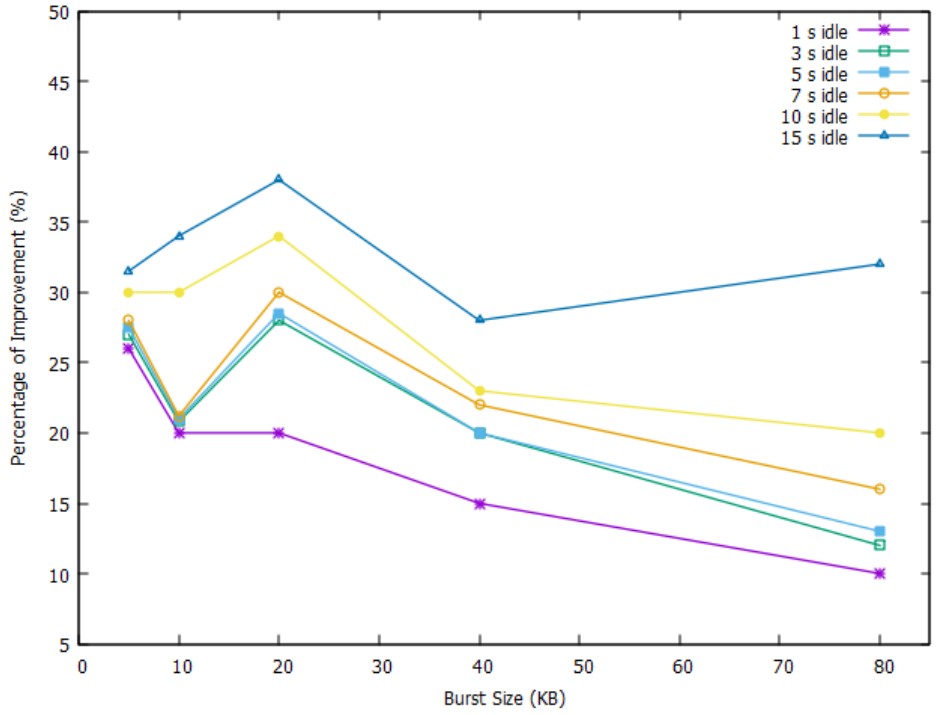

Figure 7. HTTP performance improvement in percentage is shown in a congested scenario when newCWV is used instead of NewReno over different burst sizes and idle periods.

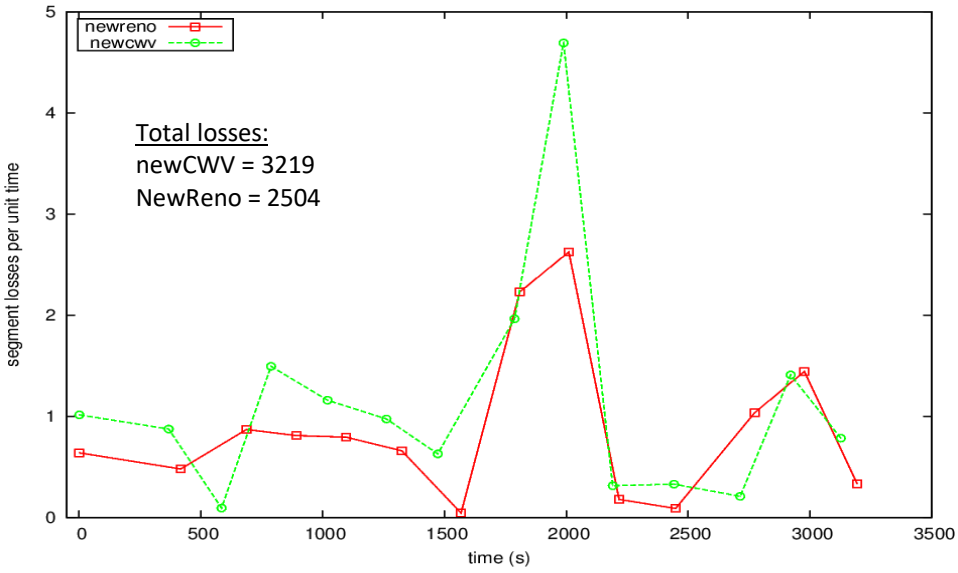

Figure 8. Loss plot comparing NewReno and newCWV; newCWV suffers more loss on average than standard TCP.

A newCWV sender allows larger bursts into the network for a large HTTP response. With a finite network buffer this will increase the probability of (burst) loss and queuing delay for this flow and other flows that share a common bottleneck (e.g., higher packet loss and jitter for concurrent real-time applications).

Although newCWV continues to show better performance in delivering bursts faster in a congested scenario, higher burst losses for larger bursts may degrade the overall average improvement in burst transfer times that could have been possible for HTTP flows.

4.4. Effect of other application on HTTP with newCWV

At first, the HTTP workload was run with newCWV without pacing. Table 2 shows that, for HTTP responses with a size of 5KB or more, there was an improvement in transfer times of 18-20%, although it is low compared to the previous cases without any cross traffic.

For large responses, such as 80 KB or more, the performance of newCWV reduces compared to standard TCP. The newCWV sender was observed to take about 10-15% more time to complete the bursts than NewReno. The large level of packet loss (and therefore delay) caused by large bursts being injected into the network eliminated the benefit of newCWV. This indicates that some burst mitigation technique is desirable.

Table 2. Performance Improvement in Percentages when HTTP runs with newCWV against NewReno. A negative value means performance degradation.

Burst Size (KB)	**Idle Periods**					
	1 s	**3 s**	**5 s**	**7 s**	**10 s**	**15 s**
5	17.9 %	18.3 %	18.4 %	18.9 %	19.1 %	19.5 %
10	10.2 %	10.6 %	10.7 %	11.2 %	11.4 %	11.7 %
20	6.7 %	6.7 %	6.9 %	7.1 %	7.3 %	7.3 %
40	4.2 %	4.4 %	4.4 %	4.6 %	4.9 %	5.2 %
80	-10.2 %	- 12.5 %	- 13.4%	- 13.9 %	- 15.1 %	- 15.3 %

In summary, when using a very congested bottleneck shared with other applications, newCWV needs to be combined with pacing – sending the burst in regular intervals – to ensure a performance improvement. Otherwise, it can lead to significant loss and induce delay to the applications using the bottleneck.

To assess the effect of newCWV, an FTP application (running NewReno) shared the bottleneck with a HTTP workload using different algorithms: NewReno, newCWV and paced newCWV.

Figure 9, shows that for the whole period of the experiment (about an hour), the FTP application competed with the HTTP traffic for a share of the capacity of the 2 Mbps bottleneck. Fluctuations in FTP throughput were observed as it shared the bottleneck with the variable rate HTTP web traffic. FTP did not suffer from starvation when the other TCP was using newCWV.
The curves for newCWV follow the curve for NewReno with hardly any differences. Though newCWV seems to be more aggressive after an idle period than standard TCP (NewReno), which helps a bursty sender application, it was reacting to congestion appropriately sharing the bottleneck with another long-lived TCP flow. This depicts the friendliness of newCWV with other TCP application like FTP.

In summary, newCWV demonstrated improved performance for HTTP traffic in both a congested and uncongested scenario. It is recommended that newCWV is used in combination with pacing, to smooth out the burst and hence also to reduce losses. newCWV is also fair in a sense that it does not poses significant threat (aggressiveness or starvation) to other co-existing TCP flows.

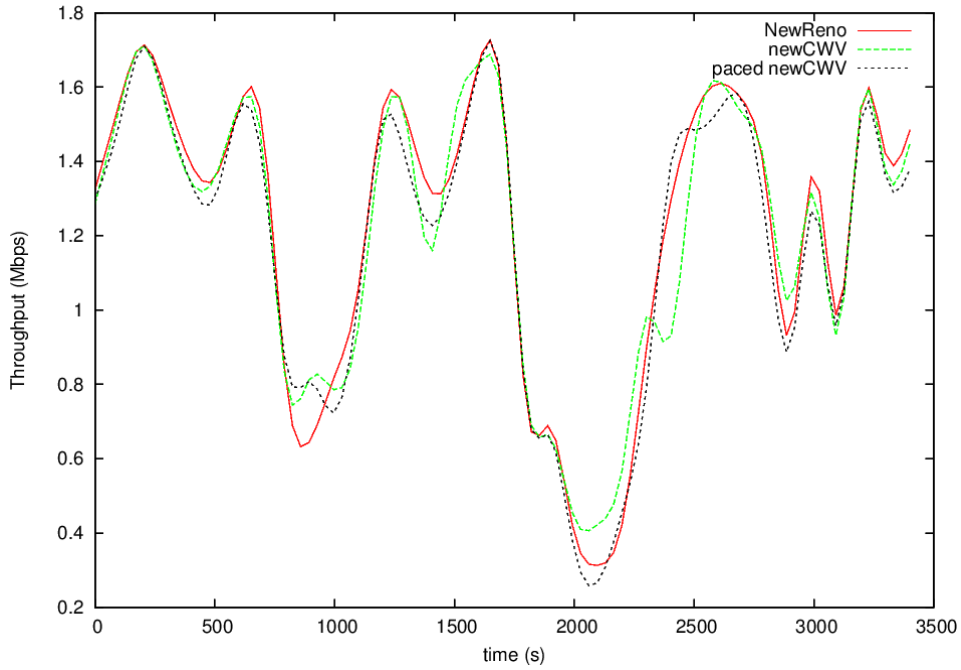

Figure 9. FTP cross-traffic throughput; no significant differences in background application performance while the HTTP traffic was running different algorithms.

4.5. Discussion

Since newCWV can avoid suboptimal performance, by defining a new way to use the *cwnd* and *ssthresh* during a rate-limited interval and specifies how to update these parameters after congestion has been detected. The mechanism defined in RFC 7661 is considered safe to use even when *cwnd* is greater than the receive window [17], because it validates the *cwnd* based on the amount of data acknowledged by the network in an RTT, which implicitly accounts for the allowed receive window.

The paper evaluated a working version of this algorithm in Linux. Since newCWV was published as an experimental specification in the RFC-series as RFC 7661, it has been implemented in some production endpoint TCP stacks. It is referenced in the latest IETF QUIC [18] transport specification: QUIC Loss Detection and Congestion Control, (RFC 9002). It is also referenced in a range of other IETF specifications, that includes Self-Clocked Rate Adaptation for Multimedia (RFC 8298), Model-Based Metrics for Bulk Transport Capacity (RFC 8337), TCP Control Block Interdependence (RFC 9040) and Operational Considerations for Streaming Media (RFC 9317).

5. CONCLUSION

Web-based traffic is the dominant type of traffic in today's Internet. As web uses HTTP/2, that uses TCP as underlying protocol, it is very important to study the transport behaviour to ensure the browsing can be made faster. A set of problems have been identified by earlier research works when bursty HTTP application use traditional TCP congestion control. Although some solutions had been proposed, they were limited and did not properly address the key requirements. newCWV seeks to address the congestion control problems and is implementable.
This paper found that the newCWV mechanism is useful for applications with variable rates in both rate-limited periods and idle periods. newCWV can lead HTTP based traffic to completion in a 50% faster manner, which means web browsing will be much more faster, web based video

streaming would be some more smoother etc. Moreover, it does not induce any harm to other network traffic sharing a common bottleneck.

The great impact of using newCWV is that application designers do not have to worry about the underling transport support for bursty applications, since the transport can accommodate a wide range of traffic variation. This gives application developers more freedom when developing new applications and can encourage the development of next generation Internet applications. For future work, it would be interesting to see the performance comparison with current TCP and QUIC implementations and to consider a variety of other network conditions.

ACKNOWLEDGEMENTS

The author acknowledges the Electronics Research Group of University of Aberdeen, UK, for all the support in conducting these experiments. This research was completed as a part of the University of Aberdeen, dot.rural project. (EP/G066051/1).

ACRONYMS

ACK	Acknowledgement
cwnd	congestion window
CA	Congestion Avoidance
CWV	Congestion Window Validation
DASH	Dynamic Adaptive Streaming over HTTP
FTP	File Transfer Protocol
HTTP	Hyper-Text Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IW	Initial Window
LKM	Loadable Kernel Module
MTU	Maximum Transfer Unit
NVP	Non-Validated Period
TCP	Transmission Control Protocol
RFC	Request for Comments
RTO	Retransmission Time Out
RTT	Round Trip Time
RW	Restart Window
SACK	Selective ACKnowledgement
ssthresh	Slow Start Threshold

REFERENCES

[1] Sandvine Resources, (2022), “Global Internet Phenomena Report”, Whitepaper, Sandvine Corporations.
[2] Fielding R. *et al* (1999), “Hypertext Transfer Protocol -- HTTP/1.1”, RFC 2616, IETF
[3] ISI (1981), “Transmission Control Protocol”, RFC 793, IETF.
[4] Handley, M., Padhye, J. and Floyd, S. (2000), “TCP Congestion window Validation”, RFC 2861, IETF.
[5] Biswas, I. (2011), “Internet congestion control for variable rate TCP traffic”, PhD Thesis, School of Engineering, University of Aberdeen.
[6] Fairhusrt, G., Sathiaseelan, A. & Secchi, R. (2015), “Updating TCP to Support Rate-Limited Traffic”, RFC 7661, IETF.
[7] Augustin, B. and Mellouk, A. (2011), “On Traffic Patterns of HTTP Applications”, Proceedings of IEEE Globecomm, Houston, USA.

[8] Jiang, H. and Dovrolis, C. (2205), “Why is the internet traffic bursty in short time scales?”, Proceedings of the ACM SIGMETRICS international conference on Measurement and Modeling of Computer Systems, Banff, Canada.
[9] Casilari, E. et. al. (2004), “Modelling of Individual and Aggregate Web Traffic”, Proceedings of 7th IEEE conference in High Speed Networks and Multimedia Communications, Toulouse, France.
[10] Duke, M., et al (2015) A Roadmap for Transmission Control Protocol (TCP) Specification Documents, RFC 7414, IETF
[11] Stevens, W. (1997), "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC 2001, IETF.
[12] Allman, M., Paxson, V. and Stevens, W. (1999), "TCP Congestion Control", RFC 2581, IETF.
[13] Allman, M., Paxson, V. and Blanton E. (2009), “TCP Congestion Control”, RFC 5681, IETF.
[14] Fall, K. and Floyd, S. (1996), "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", ACM SIGCOMM Computer Communication Review, vol. 26, no. 3, pp. 5-21.
[15] Hoe, J. (1996), “Improving the Start-up Behavior of a Congestion Control Scheme for TCP”, Proceedings of the ACM SIGCOMM.
[16] Floyd, S., Henderson, T. and Gurtov, A. (2004), “The NewReno Modification to TCP's Fast Recovery Algorithm”, RFC 3782, IETF.
[17] Xu, L., et al, CUBIC for Fast and Long-Distance Networks, draft-ietf-tcpm-rfc8312bis, Work-In-Progress, TCPM Working Group, IETF.
[18] Iyenger, J. & Thomson, M. “QUIC: A UDP-Based Multiplexed and Secure Transport”, RFC 9000, IETF.

AUTHORS

Dr Ziaul Hossain is a Network Researcher, and his interest lies in making the web faster. He has achieved BSc in Computer Science from BUET (Bangladesh) and then pursued his MSc degree at the Queen Mary, University of London (UK). He received PhD in Engineering from University of Aberdeen (UK) where his research focused on performance of different applications over the satellite platform. He has taught at several universities and currently working as a Lecturer at the University of Fraser Valley, British Columbia, Canada.

Dr Gorry Fairhurst is a Professor in the School of Engineering at the University of Aberdeen, Scotland, UK, His research is in Internet Engineering, and broadband service provision via satellite. He is an active member of the Internet Engineering Task Force (IETF) where he chairs the Transport Area working group (TSVWG) and contributes to the IETF Internet area. He has authored more than 25 RFCs with over 150 citations within the RFC-series, and currently is contributing to groups including: QUIC, 6MAN, TSVWG, and MAPRG.