Network Working Group
Internet-Draft
Intended status: Informational
Expires: 17 September 2025

S. Cheshire
Apple Inc.

16 March 2025

Source Buffer Management
draft-cheshire-sbm-02

Abstract

In the past decade there has been growing awareness about the harmful effects of bufferbloat in the network, and there has been good work on developments like L4S to address that problem. However, bufferbloat on the sender itself remains a significant additional problem, which has not received similar attention. This document offers techniques and guidance for host networking software to avoid network traffic suffering unnecessary delays caused by excessive buffering at the sender. These improvements are broadly applicable across all datagram and transport protocols (UDP, TCP, QUIC, etc.) on all operating systems.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://StuartCheshire.github.io/draft-cheshire-sbm/draft-cheshire-sbm.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-cheshire-sbm/.

Discussion of this document takes place on the sbm Working Group mailing list (mailto:sbm@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/sbm/.

Source for this draft and an issue tracker can be found at https://github.com/StuartCheshire/draft-cheshire-sbm.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Copyright Notice

Table of Contents

1. Conventions and Definitions 3
2. Introduction 3
3. Source Buffer Backpressure 4
  3.1. Direct Backpressure 4
  3.2. Indirect Backpressure 6
4. Case Study -- TCP_NOTSENT_LOWAT 8
5. Shortcomings of TCP_NOTSENT_LOWAT 9
  5.1. Platform Differences 9
  5.2. Time versus Bytes 9
  5.3. Other Transport Protocols 11
6. TCP_REPLENISH_TIME 11
  6.1. Solicitation for Name Suggestions 12
7. Application Guidance 12
  7.1. Program Structure 12
  7.2. Selection of TCP_REPLENISH_TIME value 13
8. Applicability 14
  8.1. Physical Bottlenecks 15
  8.2. Algorithmic Bottlenecks 15
  8.3. Superiority of Direct Backpressure 16
  8.4. Application Programming Interface 18
  8.5. Relationship Between Throughput and Delay 18
  8.6. Bulk Transfer Protocols 18
9. Experimental Validation 19
10. Alternative Proposals 20
  10.1. Just use UDP 20
  10.2. Packet Expiration 20
  10.3. Traffic Priorities / Head of Line Blocking 21
11. Security Considerations 23

12. IANA Considerations 23
13. References 23
13.1. Normative References 23
13.2. Informative References 23
Acknowledgments 25
Author's Address 26

1. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Introduction

In 2010 Jim Gettys identified the problem of how excessive buffering in networks adversely affects delay-sensitive applications [Bloat1][Bloat2][Bloat3]. This important work identifying a non-obvious problem has led to valuable developments to improve this situation, like fq_codel [RFC8290], PIE [RFC8033], Cake [Cake] and L4S [RFC9330].

However, excessive buffering at the source -- in the sending devices themselves -- can equally contribute to degraded performance for delay-sensitive applications, and this problem has not yet received a similar level of attention.

This document describes the source buffering problem, steps that have been taken so far to address the problem, shortcomings with those existing solutions, and new mechanisms that work better.

To explain the problem and the solution, this document begins with some historical background about why computers have buffers in the first place, and why buffers are useful. This document explains the need for backpressure on senders that are able to exceed the network capacity, and separates backpressure mechanisms into direct backpressure and indirect backpressure.

The document describes the TCP_REPLENISH_TIME socket option for TCP connections using BSD Sockets, and its equivalent for other networking protocols and APIs.

The goal is to define a cross-platform and cross-protocol mechanism that informs application software when it is a good time to generate new data, and when the application software might want to refrain from generating new data, enabling the application software to write

chunks of data large enough to be efficient, without writing too many of them too quickly. This avoids the unfortunate situation where a delay-sensitive application inadvertently writes many blocks of data long before they will actually depart the source machine, such that by the time the enqueued data is actually sent, the application may have newer data that it would rather send instead. By deferring generating data until the networking code is actually ready to send it, the application retains more precise control over what data will be sent when the opportunity arises.

The document concludes by describing some alternative solutions that are often proposed, and explains why we feel they are less effective than simply implementing effective source buffer management.

3. Source Buffer Backpressure

Starting with the most basic principles, computers have always had to deal with the situation where software is able to generate output data faster than the physical medium can accept it. The software may be sending data to a paper tape punch, to an RS232 serial port (UART), or to a printer connected via a parallel port. The software may be writing data to a floppy disk or a spinning hard disk. It was self-evident to early computer designers that it would be unacceptable for data to be lost in these cases.

3.1. Direct Backpressure

The early solutions were simple. When an application wrote data to a file on a floppy disk, the file system “write” API would not return control to the caller until the data had actually been written to the floppy disk. This had the natural effect of slowing down the application so that it could not exceed the capacity of the medium to accept the data.

Soon it became clear that these simple synchronous APIs unreasonably limited the performance of the system. If, instead, the file system “write” API were to return to the caller immediately -- even though the actual write to the spinning hard disk had not yet completed -- then the application could get on with other useful work while the actual write to the spinning hard disk proceeded in parallel.

Some systems allowed a single asynchronous write to the spinning hard disk to proceed while the application software performed other processing. Other systems allowed multiple asynchronous writes to be enqueued, but even these systems generally imposed some upper bound on the amount of outstanding incomplete writes they would support. At some point, if the application software persisted in trying to write data faster than the medium could accept it, then the

application would be throttled in some way, either by making the API call a blocking call (simply not returning control to the application, removing its ability to do anything else) or by returning a Unix EWOULDBLOCK error or similar (to inform the application that its API call had been unsuccessful, and that it would need to take action to write its data again at a later time).

It is informative to observe a comparison with graphics cards. Most graphics cards support double-buffering. This allows one frame to be displayed while the CPU and GPU are working on generating the next frame. This concurrency allows for greater efficiency, by enabling two actions to be happening at the same time. But quintuple-buffering is not better than double-buffering. Having a pipeline five frames deep, or ten frames, or fifty frames, is not better than two frames. For a fast-paced video game, having a display pipeline fifty frames deep, where every frame is generated, then waits in the pipeline, and then is displayed fifty frames later, would not improve performance or efficiency, but would cause an unacceptable delay between a player performing an action and seeing the results of that action on the screen. It is beneficial for the video game to work on preparing the next frame while the previous frame is being displayed, but it is not beneficial for the video game to get multiple frames ahead of the frame currently being displayed.

Another reason that it is good not to permit an excessive amount of unsent data to be queued up is that once data is committed to a buffer, there are generally limited options for changing it. Some systems may provide a mechanism to flush the entire buffer and discard all the data, but mechanisms to selectively remove or re-order enqueued data are complicated and rare. While it could be possible to add such mechanisms, on balance it is simpler simply to avoid committing too much unsent data to the buffer in the first place. If the backlog of unsent data is kept reasonably low, that gives the source more flexibility decide what to put into the buffer next, when that opportunity arises.

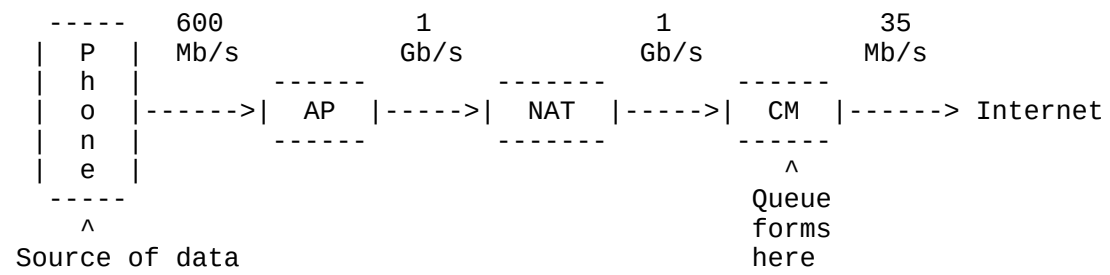
In summary, in order to give applications maximum flexibility, pending data should be kept as close to the application for as long as possible. Application buffers should be as large as needed for the application to do its work, and lower-layer buffers should be no larger than is necessary to provide efficient use of available network capacity and other resources like CPU time.

3.2. Indirect Backpressure

All of the situations described above using "direct backpressure" are one-hop communication where the CPU generating the data is connected more-or-less directly to the device consuming the data. In these cases it is relatively simple for the receiving device to exert direct backpressure to influence the rate at which the CPU sends data.

When we introduce multi-hop networking, the situation becomes more complicated. When a flow of packets travels 30 hops though a network, the bottleneck hop may be quite distant from the original source of the data stream.

For example, consider the case of a smartphone communicating via a Wi-Fi Access Point at 600 Mb/s, which is connected to a home NAT gateway via gigabit Ethernet, which is connected to a cable modem via gigabit Ethernet, which has an upstream output rate of 35Mb/s over the coaxial cable.



When the cable modem experiences an excessive flow of incoming packets arriving on its gigabit Ethernet interface, the cable modem has no direct way to cause the networking code on the smartphone to curtail the influx of data by pausing the sending application via blocking its write calls or delivering EWOULDBLOCK errors. The source of the excessive flood of data causing the problem (the smartphone) is three network hops away from the device experiencing the problem (the cable modem). When an incoming packet arrives, the cable modem's choices are limited to enqueueing the packet, discarding the packet, or enqueueing the packet and marking it with an ECN CE mark [RFC3168] [RFC9330]. The cable modem drops or marks an incoming packet in the expectation that this will, eventually, indirectly, cause the networking code and operating system on the sending device to take the necessary steps to curtail the sending application.

The reasons the cable modem's choices are so limited are because of security and packet size constraints.

Security and trust concerns revolve around preventing a malicious entity from performing a denial-of-service attack against a victim device, by sending fraudulent messages that would cause the victim to reduce its transmission rate. It is particularly important to guard against an off-path attacker being able to do this. This concern is addressed if queue size feedback generated in the network follows the same path already taken by the data packets and their subsequent acknowledgement packets. The logic is that any on-path device that is able to modify data packets (changing the ECN bits in the IP header) could equally well corrupt packets or discard them entirely. Thus, trusting ECN information from these devices does not increase security concerns, since these devices could already perform more damaging actions anyway. The sender already trusts the receiver to generate accurate acknowledgement packets, so also trusting it to report ECN information back to the sender does not increase the security risk.

A consequence of this security requirement is that it takes a full round trip time for the source to learn about queue state in the network. In many common cases this is not a significant deficiency. For example, if a user is receiving data from a well-connected server on the Internet, and the network bottleneck is the last hop on the path (e.g., the Wi-Fi hop to the user's smartphone in their home) then the location where the queue is building up (the Wi-Fi Access Point) is very close to the receiver, and having the receiver echo the queue state information back to the sender does not add significant delay.

Packet size constraints, particularly scarce bits available in the IP header, mean that for pragmatic reasons the ECN queue size feedback is limited to two states: "The source may try sending a little faster if desired," and, "The source should reduce its sending rate." Use of these increase/decrease indications in successive packets allows the sender to converge on the ideal transmission rate, and then to oscillate slightly around the ideal transmission rate as it continues to track changing network conditions.

Discarding or marking an incoming packet at some point within the network are what we refer to as indirect backpressure, with the assumption that these actions will eventually result in the sending application being throttled via having a write call blocked, returning an EWOULDBLOCK error, or some other form of backpressure that causes the source application to temporarily pause sending new data.

4. Case Study -- TCP_NOTSENT_LOWAT

In April 2011 the author was investigating sluggishness with Mac OS Screen Sharing, which uses the VNC Remote Framebuffer (RFB) protocol [RFC6143]. Initially it seemed like a classic case of network bufferbloat. However, deeper investigation revealed that in this case the network was not responsible for the excessive delay -- the excessive delay was being caused by excessive buffering on the sending device itself.

In this case the network connection was a relatively slow DSL line (running at about 500 kb/s) and the socket send buffer (SO_SNDBUF) was set to 128 kilobytes. With a 50 ms round-trip time, about 3 kilobytes (roughly two packets) was sufficient to fill the bandwidth-delay product of the path. The remaining 125 kilobytes available in the 128 kB socket send buffer were simply holding bytes that had not even been sent yet. At 500 kb/s throughput (62.5 kB/s), this meant that every byte written by the VNC RFB server spent two seconds sitting in the socket send buffer before it even left the source machine. Clearly, delaying every sent byte by two seconds resulted in a very sluggish screen sharing experience, and it did not yield any useful benefit like higher throughput or lower CPU utilization.

This led to the creation in May 2011 of a new socket option on Mac OS and iOS called “TCP_NOTSENT_LOWAT”. This new socket option provided the ability for sending software (like the VNC RFB server) to specify a low-water-mark threshold for the minimum amount of *unsent* data it would like to have waiting in the socket send buffer. Instead of inviting the application to fill the socket send buffer to its maximum capacity, the socket send buffer would hold just the data that had been sent but not yet acknowledged (enough to fully occupy the bandwidth-delay product of the network path and fully utilize the available capacity) plus some *small* amount of additional unsent data waiting to go out. Some *small* amount of unsent data waiting to go out is beneficial, so that the network stack has data ready to send when the opportunity arises (e.g., a TCP ACK arrives signalling that previous data has now been delivered). Too much unsent data waiting to go out -- in excess of what the network stack might soon be able to send -- is harmful for delay-sensitive applications because it increases delay without meaningfully increasing throughput or utilization.

Empirically it was found that setting an unsent data low-water-mark threshold of 16 kilobytes worked well for VNC RFB screen sharing. When the amount of unsent data fell below this low-water-mark threshold, kevent() would wake up the VNC RFB screen sharing application to begin work on preparing the next frame to send. Once the VNC RFB screen sharing application had prepared the next frame

and written it to the socket send buffer, it would again call kevent() to block and wait to be notified when it became time to begin work on the following frame. This allows the VNC RFB screen sharing server to stay just one frame ahead of the frame currently being sent over the network, and not inadvertently get multiple frames ahead. This provided enough unsent data waiting to go out to fully utilize the capacity of the path, without buffering so much unsent data that it adversely affected usability.

A live on-stage demo showing the benefits of using TCP_NOTSENT_LOWAT with VNC RFB screen sharing was shown at the Apple Worldwide Developer Conference in June 2015 [Demo].

5. Shortcomings of TCP_NOTSENT_LOWAT

While TCP_NOTSENT_LOWAT achieved its initial intended goal, later operational experience has revealed some shortcomings.

5.1. Platform Differences

The Linux network maintainers implemented a TCP socket option with the same name, but different behavior. While the Apple version of TCP_NOTSENT_LOWAT was focussed on reducing delay, the Linux version was focussed on reducing kernel memory usage. The Apple version of TCP_NOTSENT_LOWAT controls a low-water mark, below which the application is signalled that it is time to begin working on generating fresh data. The Linux version determines a high-water mark for unsent data, above which the application is *prevented* from writing any more, even if it has data prepared and ready to enqueue. Setting TCP_NOTSENT_LOWAT to 16 kilobytes works well on Apple systems, but can increase CPU load and severely limit throughput on Linux systems. This has led to confusion among developers and makes it difficult to write portable code that works on both platforms.

5.2. Time versus Bytes

The original thinking on TCP_NOTSENT_LOWAT focussed on the number of unsent bytes remaining, but it soon became clear that the relevant quantity was time, not bytes. The quantity of interest to the sending application was how much advance notice it would get of impending data exhaustion, so that it would have enough time to generate its next logical block of data. On low-rate paths (e.g., 250 kb/s and less) 16 kilobytes of unsent data could still result in a fairly significant unnecessary queueing delay. On high-rate paths (e.g., Gb/s and above) 16 kilobytes of unsent data could be consumed very quickly, leaving the sending application insufficient time to generate its next logical block of data before the unsent backlog ran out and available network capacity was left unused. It became clear

that it would be more useful for the sending application specify how much advance notice of data exhaustion it required (in milliseconds, or microseconds), depending on how much time the application anticipated needing to generate its next logical block of data.

The application could perform this calculation itself, calculating the estimated current data rate and multiplying that by its desired advance notice time, to compute the number of outstanding unsent bytes corresponding to that desired time. For example, if the current average data rate is 1 megabyte per second, and the application would like 0.1 seconds warning before the backlog of awaiting data runs out, then 1,000,000 x 0.1 gives us a TCP_NOTSENT_LOWAT value of 100,000 bytes.

However, the application would have to keep adjusting its TCP_NOTSENT_LOWAT value as the observed data rate changed. Since the transport protocol already knows the number of unacknowledged bytes in flight, and the current round-trip delay, the transport protocol is in a better position to perform this calculation.

In addition, the network stack knows if features like hardware offload, aggregation, and stretch acks are being used, which could impact the burstiness of consumption of unsent bytes.

Wi-Fi interfaces perform better when they send batches of packets aggregated together instead of sending individual packets one at a time. The amount of aggregation that is desirable depends on the current wireless conditions, so the Wi-Fi interface and its driver are in the best position to determine that.

If stretch acks are being used, then each ack packet could acknowledge 8 data segments, or about 12 kilobytes. If one such ack packet is lost, the following ack packet will cumulatively acknowledge 24 kilobytes, instantly consuming the entire 16 kilobyte unsent backlog, and giving the application no advance notice that the transport protocol is suddenly out of available data to send, and some network capacity becomes wasted.

Occasional failures to fully utilize the entire available network capacity are not a disaster, but we still would like to avoid this being a common occurrence. Therefore it is better to have the transport protocol, in cooperation with the other layers of the network stack, use all the information available to estimate when it expects to run out of data available to send, given the current network conditions and current amount of unsent data. When the estimated time remaining until exhaustion falls below the application' s specified threshold, the application is notified to begin working on generating more data.

5.3. Other Transport Protocols

TCP_NOTSENT_LOWAT was initially defined only for TCP, and only for the BSD Sockets programming interface. It would be useful to define equivalent delay management capabilities for other transport protocols, like QUIC [RFC9000][RFC9369], and for other network programming APIs.

6. TCP_REPLENISH_TIME

Because of these lessons learned, this document proposes a new BSD Socket option for TCP, TCP_REPLENISH_TIME.

The new TCP_REPLENISH_TIME socket option specifies the threshold for notifying an application of impending data exhaustion in terms of microseconds, not bytes. It is the job of the transport protocol to compute its best estimate of when the expected time-to-exhaustion falls below this threshold.

The new TCP_REPLENISH_TIME socket option should have the same semantics across all operating systems and network stack implementations.

Other transport protocols, like QUIC, and other network APIs not based on BSD Sockets, should provide equivalent time-based backlog-management mechanisms, as appropriate to their API design.

The time-based estimate does not need to be perfectly accurate, either on the part of the transport protocol estimating how much time remains before the backlog of unsent data is exhausted, or on the part of the application estimating how much time it will need generate its next logical block of data. If the network data rate increases significantly, or a group of delayed acknowledgments all arrive together, then the transport protocol could end up discovering that it has overestimated how much time remains before the data is exhausted. If the operating system scheduler is slow to schedule the application process, or the CPU is busy with other tasks, then the application may discover that it has underestimated how much time it will take to generate its next logical block of data. These situations are not considered to be serious problems, especially if they only occur infrequently. For a delay-sensitive application, having some reasonable mechanism to avoid an excessive backlog of unsent data is dramatically better than having no such mechanism at all. Occasional overestimates or underestimates do not negate the benefit of this capability.

The IETF Transport Services API specification [RFC9622] states that "Sent events allow an application to obtain an understanding of the amount of buffering it creates." TCP_REPLENISH_TIME goes beyond giving an application *visibility* into the amount of buffering it creates, by giving an application the ability to *specify* the amount of buffering it would *like* to create.

6.1. Solicitation for Name Suggestions

Author's note: The BSD socket option name "TCP_REPLENISH_TIME" is currently proposed as a working name for this new option for BSD Sockets. While the name does not affect the behavior of the code, the choice of name is important, because people often form their first impressions of a concept based on its name, and if they form incorrect first impressions then their thinking about the concept may be adversely affected.

For example, one suggested name was "TCP_EXHAUSTION_TIME". We view "TCP_REPLENISH_TIME" and "TCP_EXHAUSTION_TIME" as representing two interpretations of the same quantity. From the application's point of view, it is expressing how much time it will require to replenish the buffer. From the networking code's point of view, it is estimating how much time remains before it will need the buffer replenished. In an ideal world, REPLENISH_TIME == EXHAUSTION_TIME, so that the data is replenished at exactly the moment the networking code needs it. In a sense, they are two ways of saying the same thing. Since this API call is made by the application, we feel it should be expressed in terms of the application's requirement.

7. Application Guidance

7.1. Program Structure

For an application that wishes to achieve good throughput while also caring about the timeliness of its data, the recommendation is that the application use the "TCP_REPLENISH_TIME" socket option (or equivalent) to specify how much time it expects it will need to generate its next batch of data.

After setting TCP_REPLENISH_TIME for a connection, the application then uses a notification API like kevent() on Mac OS (or equivalents on other platforms) to block and wait until the networking code determines that it is time to generate new data for that connection. Immediately after the creation of a new connection, kevent() (or equivalent) will immediately report that it is ready for more data. Once the application has written enough data to build up a sufficient backlog of unsent data waiting on the source device, kevent() will stop indicating that it is inviting the application to write more

data. Once the backlog of unsent data drains to the point where the networking code expects it to be exhausted in less than the time specified by TCP_REPLENISH_TIME for that connection, kevent() again reports the socket as writable to invite the application to generate its next batch of data.

It is important to note that the kevent() signal indicating that it is time to generate new data is a hint to the application. The presence of the kevent() signal tells the application that this is a good time to generate new data; the absence of the kevent() signal is *not* a *prohibition* on the application writing more data. Even if kevent() is not signalling impending exhaustion of the data buffer, an application is still free to write as much data as is appropriate for that application (potentially limited by some other parameter, such as the SO_SNDBUF size in BSD Sockets).

Note that there is precendent for this kind of behavior in current programming APIs. For example, if a TCP connection on Linux has a socket send buffer of 1000 kilobytes, and a TCP ACK packet arrives acknowledging 3 kilobytes of data, leaving only 997 kilobytes of data remaining in the socket send buffer, then epoll() will not immediately wake up the process to replenish the data and fill the buffer back up to the full 1000 kilobytes. Instead Linux will wait until the socket send buffer occupancy has fallen to 50% before waking up the process to replenish the data. This allows the process to do a relatively small number of efficient 500-kilobyte writes instead of a huge number of little 3-kilobyte writes. [Author's note: I would appreciate a confirmation that this is correct, with a reference, or alternatively inform me if this is wrong and I will remove it.]

In this way the application is able to keep a reasonable amount of data waiting in the outgoing buffer, without building too much backlog resulting in excessive delay.

7.2. Selection of TCP_REPLENISH_TIME value

The selection of the appropriate TCP_REPLENISH_TIME value depends on the application's needs.

For example, a screen sharing server (or a video streaming source) sending data at 60 frames per second may require 17 milliseconds between when it grabs the frame from the screen (or camera), compresses it, and has the data ready for transmission. Such an application might specify a TCP_REPLENISH_TIME of 20 milliseconds, so give reasonable confidence that it will have the next frame prepared and ready before the transport protocol finishes sending the previous frame. If the video capture process is more pipelined, so that it

takes the application 17 milliseconds to capture the frame from the camera, and then a further 17 milliseconds to compress that frame, then it might specify a TCP_REPLENISH_TIME of 35 milliseconds.

For an application that cares most about achieving the highest possible video quality, and a little extra delay is not a serious problem, it may be appropriate to specify a slightly higher TCP_REPLENISH_TIME to ensure a slightly higher safety margin and reduce the risk of the transport protocol occasionally becoming starved of new data.

For an application that cares most about getting the lowest possible delay rather than achieving the highest utilization of available network capacity, it may be appropriate to specify a slightly lower TCP_REPLENISH_TIME to keep buffering delay to a minimum, at the risk of occasionally leaving some amount of network capacity unused.

Continuing the example of the video streaming application, if a given frame has a lot of movement relative to the previous frame, then the video compression algorithm can be set either to encode the frame at lower quality (yielding the same compressed data size) or at the same quality (yielding a larger compressed data size). In the latter case, if the compressed data size is three times larger than a typical compressed frame, the application can still write that larger block of data. The write is not prevented or blocked just because it exceeds the desired TCP_REPLENISH_TIME budget. After writing this larger block of data kevent() (or equivalent) will not signal that it is ready for more data until after the large block has drained, which may take more than one typical frame time. In this way the kevent() loop has the effect of automatically reducing the frame rate to stay within the available network capacity, instead of continuing to generate frames faster than the network can carry them and building up an increasing backlog (with a corresponding increasing delay). The application may accept this reduced frame rate, or it may choose to adjust its video compression algorithm to a lower quality so as to increase the frame rate. In either case, the source device buffering delay is kept under control.

In all cases, it is expected that application writers will experiment with different values of TCP_REPLENISH_TIME to determine empirically what works best for their application.

8. Applicability

This time-based backlog management is applicable anywhere that a queue of unsent data may build up on the sending device.

8.1. Physical Bottlenecks

A backlog may build up on the sending device if the source of the packets is simply generating them faster than the outgoing first-hop interface is able to send them. This will cause a queue to build up in the network hardware or its associated driver. In this case, to avoid packets suffering excessive queueing delay, the hardware or its driver needs to communicate backpressure to IP, which needs to communicate backpressure to the transport protocol (TCP or QUIC), which needs to communicate backpressure to the application that is the source of the data. We refer to this case as a physical bottleneck.

For an example of a physical bottleneck, consider the case where a user has symmetric 1Gb/s Internet service, and they are sending data from a device communicating via Wi-Fi at a lower rate, say 300 Mb/s. In this case (assuming the device is communicating with a well-connected server on the Internet) the limiting factor of the entire path is the first hop -- the sending device's Wi-Fi interface. If the device's Wi-Fi hardware, driver, and networking software does not produce appropriate backpressure, then outgoing network traffic will experience increasing delays. The Linux Byte Queue Limits mechanism [Hruby][THJ][Herbert] is one example of a technique to tune hardware buffers to an appropriate size so that they are large enough to avoid transmitter starvation without being so large that they unnecessarily increase delay.

Poor backpressure from first-hop physical bottlenecks can produce the ironic outcome that upgrading home Internet service from 100Mb/s to 1Gb/s can sometimes result in a customer getting a worse user experience, because the service upgrade causes the bottleneck hop to change location, from the Internet gateway (which may have good queue mangement using L4S [RFC9330]) to the source device's Wi-Fi interface, which may have very poor source buffer management.

8.2. Algorithmic Bottlenecks

In addition to physical bottlenecks, there are other reasons why software on the sending device may choose to refrain from sending data as fast as the outgoing first-hop interface can carry it. We refer to these as algorithmic bottlenecks.

In the case study in Section 4, the bottleneck was the transport protocol's rate management (congestion control) algorithm, not a physical constraint of the outgoing first-hop interface (which was gigabit Ethernet).

* If the TCP receive window is full, then the sending TCP implementation will voluntarily refrain from sending new data, even though the device's outgoing first-hop interface is easily capable of sending those packets. This is vital to avoid overrunning the receiver with data faster than it can process it.

* The transport protocol's rate management (congestion control) algorithm may determine that it should delay before sending more data, so as not to overflow a queue at some other bottleneck within the network. This is vital to avoid overrunning the capacity of the bottleneck network hop with data faster than it can forward it, resulting in massive packet loss, which would equate to a large wastage of resources at the sender, in the form of battery power and network capacity wasted by generating packets that will not make it to the receiver.

* When packet pacing is being used, the sending network implementation may choose voluntarily to moderate the rate at which it emits packets, so as to smooth the flow of packets into the network, even though the device's outgoing first-hop interface might be easily capable of sending at a much higher rate. When packet pacing is being used, a temporary backlog can build up at this layer if the source is generating data faster than the pacing rate.

Whether the source application is constrained by a physical bottleneck on the sending device, or by an algorithmic bottleneck on the sending device, it is still beneficial to avoid overcommitting data to the outgoing buffer.

As described in the introduction, the goal is for the application software to be able to write chunks of data large enough to be efficient, without writing too many of them too quickly, and causing unwanted self-inflicted delay.

8.3. Superiority of Direct Backpressure

Since multi-hop network protocols already implement indirect backpressure signalling in the form of discarding or marking packets, it can be tempting to use the same mechanism to generate backpressure for first-hop physical bottlenecks. Superficially there might seem to be some attractive elegance in having the first hop use the same drop/mark mechanism as the remaining hops on the path. However, this is not an ideal solution because indirect backpressure from the network is very crude compared to the much richer direct backpressure that is available within the sending device itself. Relying on indirect backpressure by discarding or marking a packet in the sending device is a crude rate-control signal, because it takes a

full network round-trip time before the effect of that drop or mark is observed at the receiver and echoed back to the sender, and it may take multiple such round trips before it finally results in an appropriate reduction in sending rate.

In contrast to queue buildup in the network, queue buildup at the sending device has different properties regarding (i) security, (ii) packet size constraints, and (iii) immediacy. This means that when it is the source device itself that is building up a backlog of unsent data, designers of networking software have more freedom about how to manage this.

(i) When the source of the data and the location of the backlog are the same physical device, network security and trust concerns do not apply.

(ii) When the mechanism we use to communicate about queue state is a software API instead of packets sent though a network, we do not have the constraint of having to work within limited IP packet header space.

(iii) When flow control is implemented via a local software API, the delivery of STOP/GO information to the source is immediate.

Furthermore, the situation where the bottleneck is the first hop of the path is a fairly common case, and it is the case where indirect backpressure is at its worst (it takes an entire network round trip to learn what is already known on the sending device), so it is worthwhile optimizing for this common case.

Direct backpressure can be achieved simply making an API call block, or by returning a Unix EWOULDBLOCK error, or using equivalent mechanisms in other APIs, and has the effect of immediately halting the flow of new data. Similarly, when the system becomes able to accept more data, unblocking an API call, indicating that a socket has become writable using select() or kevent(), or equivalent mechanisms in other APIs, has the effect of immediately allowing the production of more data.

Indirect backpressure is vastly inferior to direct backpressure. For rate adjustment signals generated within the network, indirect backpressure has to be used because in that situation better alternatives are not available. Direct backpressure is vastly superior, and where direct backpressure mechanisms are possible they should be preferred over indirect backpressure mechanisms.

8.4. Application Programming Interface

It is important to understand that these backpressure mechanisms at the API layer are not new. By necessity, backpressure has existed for as long as we have had networking APIs (or serial port APIs, or file system APIs, etc.). All applications have always had to be prevented from generating a sustained stream of data faster than the medium can consume it. The problem is not that backpressure mechanisms did not exist, but that historically these backpressure mechanisms were exercised far too late, after an excessive backlog had already built up.

The proposal in this Source Buffer Management document is not to define entirely new API mechanisms that did not previously exist, or to fundamentally change how networking applications are written; the proposal is to use existing networking API mechanisms more effectively. Depending on how a networking application is written, using kevent() or similar mechanisms to tell it when it is time to write to a socket, it may be that the only change the application needs is a single call using TCP_REPLENISH_TIME to indicate its expected time budget to generate a new block of data, and everything else in the application remains completely unchanged.

8.5. Relationship Between Throughput and Delay

Is is important to understand that Source Buffer Management using TCP_REPLENISH_TIME does not alter the overall long-term average throughput of a data transfer. Calculating the optimum rate to send data (so as not to exceed receiver's capacity, or the available network capacity) remains the responsibility of the transport protocol. Using TCP_REPLENISH_TIME does not alter the data rate; it controls the delay between the time when data is generated and the time when that data departs the sending device. Using the example from Section 4, in both cases the long-term average throughput was 500 kb/s. What changed was that originally the application was generating 500 kb/s with two seconds of outgoing delay; after using TCP_REPLENISH_TIME the application was generating 500 kb/s with 250 milliseconds of outgoing delay.

8.6. Bulk Transfer Protocols

It is frequently asserted that latency matters primarily for interactive applications like video conferencing and on-line games, and latency is relatively unimportant for most other applications.

We do not agree with this characterization.

Even for large bulk data transfers -- e.g., downloading a software update or uploading a video -- we believe latency affects performance.

For example, TCP Fast Retransmit [RFC5681] can immediately recover a single lost packet in a single round-trip time. TCP generally performs at its absolute best when the loss rate is no more than one loss per round-trip time. More than one loss per round-trip time requires more extensive use of TCP SACK blocks, which consume extra space in the packet header, and makes the work of the rate management (congestion control) algorithm harder. This can result in the transport protocol temporarily sending too fast, resulting in additional packet loss, or too slowly, resulting in underutilized network capacity. For a given fixed loss rate (in packets lost per second) a higher total network round-trip time (including the time spent in buffers in the sending network interface, below the transport protocol layer) equates to more lost packets per network round-trip time, causing error recovery to occur less quickly. A transport protocol cannot make rate adaptation changes to adjust to varying network conditions in less than one network round-trip time, so the higher the total network round-trip time is, the less agile the transport protocol is at adjusting to varying network conditions.

In short, a client running over a transport protocol like TCP may itself not be a real-time delay-sensitive application, but a transport protocol itself is most definitely a delay-sensitive application, responding in real time to changing network conditions. The application doing the large bulk data transfer may have no need to use TCP_REPLENISH_TIME to manage its own application-layer backlog, but the transport protocol it is using (e.g., TCP or QUIC) obtains significant benefit from receiving timely direct backpressure from the driver and hardware below to keep the network round-trip time low.

9. Experimental Validation

The mechanisms described in this document do not exist for purely ideological or philosophical reasons. Any work to improve source buffer management in end systems should be validated by confirming that real-world applications exhibit verifiably improved responsiveness, and by taking measurements using benchmark tools that measure application-layer round-trip times under realistic working conditions [RPM]. Using the ‘ping’ command to send one ICMP Echo packet [RFC792] per second on an otherwise idle network is not a good predictor of real-world application performance. Testing the scenario where the outgoing buffer is almost always completely empty due to lack of traffic does not reveal anything about how it will perform when a nontrivial amount of data is being sent and the buffer

is no longer empty. The quality of the source buffer management policy and the effectiveness of its backpressure mechanisms only become apparent when a source of the data is willing and able to exceed the available network capacity, and the backpressure mechanisms become operational to regulate the rate that data is being sent.

10. Alternative Proposals

10.1. Just use UDP

Because much of the discussion about network latency involves talking about the behavior of transport protocols like TCP, sometimes people conclude that TCP is the problem, and think that using UDP will solve the source buffering problem. It does no such thing. If an application sends UDP packets faster than the outgoing network interface can carry them, then a queue of packets will still build up, causing increasing delay for those packets, and eventual packet loss when the queue reaches its capacity.

Any protocol that runs over UDP (like QUIC [RFC9000][RFC9369]) must end up re-creating the same rate optimization behaviors that are already built into TCP, or it will fail to operate gracefully over a range of different network conditions.

Networking APIs for UDP cannot include capabilities like reliability, in-order delivery, and rate optimization, because the UDP header has no sequence number or similar fields that would make these capabilities possible. However, networking APIs for UDP SHOULD provide appropriate backpressure to the client software, so that software using UDP can avoid unnecessary self-inflicted delays when inadvertently attempting to send faster than the outgoing first-hop interface can carry it. This backpressure allows advanced protocols like QUIC to provide capabilities like reliability, in-order delivery, and rate optimization, while avoiding unwanted delay caused by on-device first-hop buffering.

10.2. Packet Expiration

One approach that is sometimes used, is to send packets tagged with an expiration time, and if they have spent too long waiting in the outgoing queue then they are automatically discarded without even being sent. This is counterproductive because the sending application does all the work to generate data, and then has to do more work to recover from the self-inflicted data loss caused by the expiration time.

If the outgoing queue is kept short, then the amount of unwanted delay is kept correspondingly short. In addition, if there is only a small amount of data in the outgoing queue, then the cost of sending a small amount of data that may arguably have become stale is also small -- usually smaller than the cost of having to recover missing state caused by intentional discard of that delayed data.

For example, in video conferencing applications it is frequently thought that if a frame is delayed past the point where it becomes too late to display it, then it becomes a waste of network capacity to send that frame at all. However, the fallacy in that argument is that modern video compression algorithms make extensive use of similarity between consecutive frames. A given video frame is not just encoded as a single frame in isolation, but as a collection of visual differences relative to the previous frame. The previous frame may have arrived too late for the time it was supposed to be displayed, but the data contained within it is still needed to decode and display the current frame. If the previous frame was intentionally discarded by the sender, then the subsequent frames are also impacted by that loss, and the cost of repairing the damage is frequently much higher than the cost would have been to simply send the delayed frame. Just because a frame arrives too late to be displayed does not mean that the data within that frame is not important. The data contained with a frame is used not only to display that frame, but also in the construction of subsequent frames.

10.3. Traffic Priorities / Head of Line Blocking

People are often very concerned about the problem of head-of-line-blocking, and propose to solve it using techniques such as packet priorities, the ability cancel unsent pending messages [MMADAPT], and out-of-order delivery on the receiving side. There is an unconscious unstated assumption baked into this line of reasoning, which is that having an excessively long outgoing queue is inevitable and unavoidable, and therefore we have to devote a lot of our energy into how to organize and prioritize and manage that obligatory excessive queue. In contrast, if we take steps to keep queues short, the problems of head-of-line-blocking largely go away. When the line is consistently short, being at the back of the line is no longer the serious problem that it used to be.

On the receiving device, if a single packet is lost, then subsequent data cannot be delivered to the receiving application in-order until the missing packet is retransmitted and arrives at the receiver to fill in the gap. Using techniques like TCP Fast Retransmit [RFC5681], this recovery can occur in a single network round-trip time, making the effective application-layer round-trip time for that

data twice the underlying network round-trip time. When using techniques like L4S [RFC9330] to minimize network losses and queueing delays, even twice the network round-trip time may be substantially better than today' s typical network round-trip times. For many applications the difference between one network round-trip time and two network round-trip times may have negligible effect on the user experience of that application, especially if such degradations are rare.

There is a small class of applications, like audio and video conferencing over long distances, where people may feel that a single network round-trip time provides adequate user experience but two network round-trip times would be unacceptable. This is the scenario where out-of-order delivery on the receiving side appears attractive. However, writing application code to take advantage of out-of-order delivery has proven to be surprising difficult. Many modern data types are not amenable to easy interpretation when parts of the data are missing. In compressed data, such as ZIP files, JPEG images, and modern video formats, correct interpretation of data depends on having the data that preceded it, making it very difficult to write software that will correctly handle gaps in the data. For example, in a compressed video stream where a frame is encoded as differences relative to the previous frame, there is no easy way to decode the current frame when the previous frame is missing. This scenario has many similarities to Packet Expiration (Section 10.2) except that when using Packet Expiration the data loss is intentional and self-inflicted, whereas out-of-order delivery encompasses both the case of intentional packet loss by the sender and inadvertent packet loss in the network.

In a network using L4S [RFC9330] the motivation for writing extremely complicated software to handle out-of-order delivery (i.e., data with gaps) is weak, especially when L4S makes actual packet loss exceedingly rare, and Fast Retransmit recovers from these rare losses in a single extra round-trip time, which is low when L4S is being used.

Note that the justification for scenarios where one network round-trip time is acceptable but two network round-trip times would be unacceptable only applies when the network round-trip time is large relative to the user-experience requirements of the application. For example, for distributing real-time audio within a home network, where round-trip delays over the local Ethernet or Wi-Fi network are just a few milliseconds, simply relying on Fast Retransmit to recover occasional lost packets within a few milliseconds [TCPFR] makes the application programming easier and is preferable to accepting received data out of order and then playing degraded audio due to gaps in the data stream. To give some calibration, the speed of

sound in air is roughly one foot per millisecond, so a 5 ms playback delay buffer to allow for loss recovery equates to the same delay as standing five feet further away from the speakers.

11. Security Considerations

No security concerns are anticipated resulting from reducing the amount of stale data sitting in buffers at the sender.

12. IANA Considerations

This document has no IANA actions.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

13.2. Informative References

[Bloat1] Gettys, J., "Whose house is of glasse, must not throw stones at another", December 2010, <https://gettys.wordpress.com/2010/12/06/whose-house-is-of-glasse-must-not-throw-stones-at-another/>.

[Bloat2] Gettys, J. and K. Nichols, "Bufferbloat: Dark Buffers in the Internet", ACM Queue, Volume 9, issue 11 , November 2011, <https://queue.acm.org/detail.cfm?id=2071893>.

[Bloat3] Gettys, J. and K. Nichols, "Bufferbloat: Dark Buffers in the Internet", Communications of the ACM, Volume 55, Number 1 , January 2012, <https://dl.acm.org/doi/10.1145/2063176.2063196>.

[Cake] Hiland-Jrgensen, T., Taht, D., and J. Morton, "Piece of CAKE: A Comprehensive Queue Management Solution for Home Gateways", 2018 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN) , June 2018, <https://ieeexplore.ieee.org/document/8475045>.

[Demo] Cheshire, S., "Your App and Next Generation Networks", Apple Worldwide Developer Conference , June 2015, <https://developer.apple.com/videos/play/wwdc2015/719/?time=2199>.

[Herbert] Tom Herbert, "Byte Queue Limits: the unauthorized biography", January 2025, <https://medium.com/@tom_84912/byte-queue-limits-the-unauthorized-biography-61adc5730b83>.

[Hruby] Tom Hrub, "Byte Queue Limits", August 2012, <https://blog.linuxplumbersconf.org/2012/wp-content/uploads/2012/08/bql_slide.pdf>.

[MMADAPT] Aiman Erbad and Charles Buck Krasic, "Sender-side buffers and the case for multimedia adaptation", ACM Queue, Volume 10, issue 10 , October 2012, <https://queue.acm.org/detail.cfm?id=2381998>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, DOI 10.17487/RFC5681, September 2009, <https://www.rfc-editor.org/info/rfc5681>.

[RFC6143] Richardson, T. and J. Levine, "The Remote Framebuffer Protocol", RFC 6143, DOI 10.17487/RFC6143, March 2011, <https://www.rfc-editor.org/info/rfc6143>.

[RFC792] Postel, J., "Internet Control Message Protocol", STD 5, RFC 792, DOI 10.17487/RFC0792, September 1981, <https://www.rfc-editor.org/info/rfc792>.

[RFC8033] Pan, R., Natarajan, P., Baker, F., and G. White, "Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem", RFC 8033, DOI 10.17487/RFC8033, February 2017, <https://www.rfc-editor.org/info/rfc8033>.

[RFC8290] Hoeiland-Joergensen, T., McKenney, P., Taht, D., Gettys, J., and E. Dumazet, "The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm", RFC 8290, DOI 10.17487/RFC8290, January 2018, <https://www.rfc-editor.org/info/rfc8290>.

[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>.

[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>.

[RFC9369] Duke, M., "QUIC Version 2", RFC 9369, DOI 10.17487/RFC9369, May 2023, <https://www.rfc-editor.org/info/rfc9369>.

[RFC9622] Trammell, B., Ed., Welzl, M., Ed., Enghardt, R., Fairhurst, G., Khlewind, M., Perkins, C. S., Tiesel, P.S., and T. Pauly, "An Abstract Application Programming Interface (API) for Transport Services", RFC 9622, DOI 10.17487/RFC9622, January 2025, <https://www.rfc-editor.org/info/rfc9622>.

[RPM] Paasch, C., Meyer, R., Cheshire, S., and W. Hawkins, "Responsiveness under Working Conditions", Work in Progress, Internet-Draft, draft-ietf-ippm-responsiveness-05, 21 October 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-ippm-responsiveness-05>.

[TCPFR] Cheshire, S., "Ruckus WiFi Evaluation", April 2006, <http://stuartcheshire.org/papers/Ruckus-WiFi-Evaluation.pdf>.

[THJ] Toke Hiland-Jrgensen, "The State of the Art in Bufferbloat Testing and Reduction on Linux", March 2013, <https://www.ietf.org/proceedings/86/slides/slides-86-iccrg-0.pdf>.

Acknowledgments

This document has benefited from input and suggestions from: Chris Box, Morten Brrup, Neal Cardwell, Yuchung Cheng, Eric Dumazet, Jonathan Lennox, Sebastian Moeller, Yoshifumi Nishida, Christoph Paasch, Kevin Smith, Ian Swett, Michael Welzl, all who joined the side meeting at IETF 121 in Dublin (November 2024), and others [please don’ t be shy about reminding me if I somehow missed your name].

Author’s Address

Stuart Cheshire
Apple Inc.
Email: cheshire@apple.com