Multi-Stream TCP Design

Nicola Zingirian
Department of Information Engineering
University of Padova
Via Gradenigo 6, 35131 Padova, IT
nicola.zingirian@unipd.it.

2023 IEEE 19th International Conference on Intelligent Computer Communication and Processing (ICCP) | 979-8-3503-7035-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCP60212.2023.10398622

***Abstract*— The proposed Multi-Stream TCP (MS-TCP) transport protocol challenges the traditional one-to-one relationship between the Transport Control Protocol (TCP) connections and full-duplex streams. This paper explores the feasibility and advantages of allowing a single TCP connection to support multiple streams through distinct sockets, with a focus on enhancing the efficiency of concurrent web transactions. The MS-TCP protocol addresses the increasing demand for efficient utilization of TCP network connections in today's network interoperation, dominated by HTTP-based transactions, offering a legacy-aware architecture alternative to the disruptive HTTP/3 QUIC solution. By relaxing the strict connection-stream relationship, MS-TCP introduces the concept of "channel", a single TCP-based connection that can accommodate multiple concurrent streams. The paper presents a detailed examination of the design, architecture, and implementation choices involved in realizing the requirements of MS-TCP. These requirements include adherence to POSIX Standard APIs, seamless integration with existing network middleboxes, mitigation of Head-of-Line (HoL) blocking issues, enabling zero roundtrip time, and skipping the slow-start state when activating new streams.**

Keywords— QUIC, HTTP/3, Head of Line Blocking

I. Introduction

Most network communications, either user-to-service or server-to-server, rely on the Hyper-Text Transfer Protocol (HTTP) [1] protocol in Web, Web Services [2], or REST [3][4] transactions. Hence, the efficient interaction between the HTTP and the underlying transport protocol has gained significant importance in ensuring optimal network service performance, particularly in the face of escalating bandwidth and limited improvements in latency reduction [5][6] in the network links. Despite the advancements in efficiency made throughout the history of the web, analyzed later in this paper, the demand for latency and Head of Line (HoL) blocking reduction has now reached such a level of criticality that the next HTTP/3 [7] protocol, in conjunction with the QUIC [8] protocol, started revolutionizing the OSI layering over the Internet. The revolution stands in the retirement of the Internet dominant stream-oriented transport protocol [9], i.e. the TCP [10], and the massive adoption of the unreliable message-oriented delivery protocol UDP [11], consequently moving the implementation of the stream management at the application level. While Google, in cooperation with the IETF, is rapidly pushing such a revolution, a minority literature shows relevant drawbacks of dismissing the TCP, i.e., protocol increased vulnerability to attacks [12], need for complex tools for troubleshooting [13], reduced performance for large stream and increased CPU overheads [14] , need to stay up to date in a diversity of implementations [15], and the upcoming need for doubled TCP and QUIC maintenance to support the TCP fallback [16].

On the other side, surprisingly limited initiatives aimed at adding some of attractive QUIC features to the legacy TCP protocol [17][18][19][20] instead of obsoleting it, although its extensibility[21]. This paper intend to fill this gap by contributing a TCP variant, called the Multi-Stream (MS)-TCP that effectively handles a set of concurrent independent streams, like QUIC protocol, preserving the full compatibility with both Posix [22] Socket APIs used by applications and with the existing network middle boxes [23], that can regard the Multi-Stream Protocol as a regular TCP connection.

The paper is organized as follows. Section II analyzes the evolution of the HTTP utilization of the transport layer services up to the latest 3.0 version and motivates the MS-TCP approach. Section III shows the MS-TCP requirements and architecture. Section IV presents the MS-TCP design in terms of variations of the existing TCP. Section V shows discusses the scenarios opened by the MS-TCP. Section VI concludes the paper addressing the future steps in the project.

II. Background and Motivation

A short survey of the interactions between the Web Application and Transport protocols is helpful to recognize the rationale of the new HTTP/3 protocol and to precisely motivate the MS-TCP proposal.

A. Mapping of the HTTP Transactions on the Transport

Since its origin, the HTTP protocol has been relying on the TCP protocol, as stream-oriented service offered by TCP matches the "entity-body" files transfer.

1) One transaction over one TCP connection

In the earliest HTTP versions (the so-called 0.9 and the 1.0 [24]), each transaction corresponded to exactly one TCP connection, so every transaction opened a connection just before starting the HTTP request and closed it just after completing the HTTP response. This approach, today referred to as "Connection close" [25] modality, requires web clients to establish and terminate individual connections for each web page embedded object [26] (e.g., HTML/text, icons, multimedia, frames, JSON records, JavaScript code etc.). Thus, web clients follow the strategy of opening concurrent TCP connections, one for each transaction to overlap network and application latencies. However, to avoid overwhelming the web server system with an excessive number of simultaneous open TCP connections, clients keep a limited number of concurrent connections opened, typically a pool of six connections [27] consequently limiting the number of concurrent transactions.

2) Serialized Transactions over one TCP connection

To improve the performance of each TCP connection in the pool, HTTP/1.1 [28] has first introduced the "Connection keep-alive" [25] modality as a default, which allows several transactions to run back-to-back over the same TCP connection. Thanks to this modality, apart from the first transaction, which opens a new connection, the following ones do not experience the inefficiency of the connection establishment handshaking latency and of the TCP slow-start phase, as they immediately enter the congestion avoidance phase [29]. Consecutive transactions serialized over a single TCP however cannot remove the Head of Line (HoL) blocking at the application level[30], due to the application

This work has been supported by Click & Find s.r.l

latencies. Streams that are slow to produce or render, on the server or client sides respectively, reduce the connection throughput thus increasing the queueing time of the connection waiting for acquiring an available TCP connection from the pool. This happens as an effect of the transport-level "flow control" even when no congestion takes place.

3) Concurrent Transactions over one TCP Connection

To reduce this bottleneck, the HTTP/2 protocol allows several transactions to use the same TCP connection concurrently instead of consecutively. Under HTTP/2, the application segments the stream into binary frames and sends them over a TCP stream. Therefore, the application packetizes the stream and interleaves the packets of different streams serializing them into a TCP stream. The TCP splits the stream again into IP datagrams (and, by the irony of fate, the PPP [31] reserializes again over mobile links). Unfortunately, this complex architecture could not remove the HoL Blocking at the transport level [32], as an occasional loss or out-of-order delivery of a stream packet, causes a slow down to the other streams sharing the connection.

4) Concurrent Transactions over UDP

To mitigate the transport-layer Head-of-Line (HoL) blocking, the QUIC community has suggested dismissing, rather than updating, the Transmission Control Protocol (TCP), and laying the latest HTTP/3 [7] over the User Datagram Protocol (UDP). This choice appoints the application for segmenting the stream, inserting one or more segments in each UDP datagram, sequencing, reordering, and retransmitting the datagrams, even preventing network congestion and peer's buffer overflow. The top consequence of this choice is to move the stream support from the transport layer to the application layer. QUIC proponents argue that this choice is acceptable or even desirable, to contrast the "ossification" [33] of the transport layer emphasizing that, unlike the application layer, which operates in the user space, the transport layer works within the kernel space, resulting in limited flexibility and upgradability.

TABLE I. TCP CONGESTION CONTROL IN LINUX KERNEL

TCP congestion control	*1st Version*	*Issue date*
Vegas TCP	2.2	1/1999
Binary Increase Congestion control (BIC)	2.6.8	8/2004
Cubic Binary Increase Congestion control (CUBIC)	2.6.19	11/2006
Westwood TCP	2.6.30	6/2009
Proportional Rate Reduction (PRR) and Data Center TCP (DCTCP)	3.2	1/2012
BottleneckBandwidth and Roundtrip (BBR)	4.9	12/2016

B. Motivation of the MS-TCP

As the HTTP protocol is widely used across various applications, we assume, in continuity with the well-established mapping of OSI layers over the host architectures, that an architecture in which every application should re-implement or embed libraries for the stream support, sequencing, flow, and congestion controls, appears intrinsically weak. The main concerns are that the exploding number of combinations of possible interacting peer implementations (and versions) make the performance and security assessment and guarantees extremely hard [34] to control. We also regard the discussion around "ossification" as not well substantiated and deriving from a cultural barrier between application and operating system developers. Objectively speaking, the kernel modules have not counted less TCP congestion control updates (see TABLE I.) than the HTTP stream management evolution steps reported in Section II.A. Lastly, we note that the removal of the stream management from the transport layer makes most IP traffic, previously devoted to the TCP, move to UDP. This potentially opens huge compatibility issues with all network middleboxes, e.g., firewalls, NATs, traffic shapers, and load balancers, transparent proxies, which are integral components of the network infrastructure that must be regarded with caution to ensure interoperability (see e.g., [35]).

Under such motivations, the paper shows that an alternative transport-level support, and specifically the proposed Multi-Stream TCP (MS-TCP), can embed the same QUIC stream handling strategies, while allowing the application to use conventional stream-oriented sockets, and what is of utmost importance, using the legacy POSIX Socket system call APIs. As an example of the practical consequences, the MS-TCP allows even the oldest HTTP/1.0 client/server applications experience QUIC-like stream management with no need of updating any line of code or any HTTP headers.

III. MULTI-STREAM TCP ARCHITECTURE

The conventional TCP protocol operates under the assumption that there is a direct correspondence between each connection and each full-duplex stream. In other words, opening or closing a stream is synonymous with opening or closing a connection. Conversely, the MS-TCP enables a single TCP connection to accommodate multiple streams, with a specific emphasis on improving the efficiency of concurrent web transactions.

A. Channel vs. Stream TCP mechanisms

By decoupling streams from connections, certain TCP mechanisms appear more appropriately mapped to either the connection itself or individual streams. To better stress this distinction, we introduce the term "channel" to designate a TCP connection supporting multiple stream sessions. TABLE II. shows the analysis results.

TABLE II. TCP MECHANISM SCOPES

Mechanism	*Scope*
Congestion Control	Channel
Flow Control	Stream
Socket	Stream
Addresses and Ports	Channel

TCP's congestion control mechanism monitors the network path performance and operates at the channel level rather than the individual stream level. When congestion occurs, it is crucial to address it at the channel level to ensure quick reaction and inter-stream fairness. By taking simultaneous action across all streams, congestion can be promptly mitigated. By contrast, the flow control mechanism regulates the processing speed differences between stream producers and consumers at the stream level, without directly dealing with the channel or network path. This differentiation

of the congestion and flow control scopes is the architectural key to achieve efficient overlapping of web transactions on a single TCP connection at the transport layer, without involving the application layer, as done in the solutions presented in Sections II.A.3) and II.A.4). The Local and Remote pairs (ports, IP addresses) are channel-specific, being the coordinates to address the peer communicating processes running on remote nodes. Finally, the socket scope is the stream, being the file descriptor of each separate stream.

B. Socket API

TABLE II. leads to propose a model for which many sockets, each handling their own distinct stream, can refer to one connection ("channel") and share the same TCP state variables (namely the TCP Control Block, TCB [36]), such as ports and addresses, TCP FSM state, congestion window, retransmission counters, and roundtrip time estimation variables. Conversely, each socket refers to distinct pair of receiving and transmission buffers with its own flow control handling.

When the *connect* system call operates on a socket, the MS-TCP, before establishing a new TCP connection, attempts linking that socket to an already-established peer-equivalent connection, managed by another socket present in the same process file descriptor table. If an existing connection is found and linked, the open request is conveyed via that connection to the remote peer. The correspondent *accept* system call creates the new passively opened socket, directly linked to the already opened connection. This mechanism of linking sockets to an existing connection eliminates further connection handshaking and slow-start procedures, enabling the socket stream to plug in a fully operating connection.

It is worth noting that also TCP binds many sockets to the same local port and address whenever passively accepting connections over a listening socket. However, those sockets still refer to distinct connections having different remote ports or addresses. Instead, MS-TCP can manage many passively or actively opened sockets over one TCP connection thus supporting multiple streams over a single channel.

IV. MS-TCP DESIGN

A. Requirements of MS-TCP

The MS-TCP addresses the same performance requirements of the QUIC protocol [37], i.e., contrasting the Head-of-Line (HoL) blocking among different concurrent streams, enable the activation of a new stream with zero roundtrip time, and skip the TCP slow start phase.
In addition, the MS-TCP addresses other legacy compliance requirements not covered by QUIC, i.e., keeping the POSIX Standard Socket APIs for stream-oriented sockets unaltered, interacting with all network middleboxes as a regular TCP connection, keeping multiple stream overheads smaller or equal to the ones of the equivalent separate TCP connections, both for network and host systems. The Transport Security Layer [38] interaction with the MS-TCP, i.e., how it works at the channel level, to remove the security overhead at the stream level, is beyond the scope of this paper.

B. MS-TCP Segment Format

The MS-TCP segment is a regular TCP segment, with a few fields slightly altered in semantics in the TCP header or a few new ones added in the options. This modification aims at keeping new pieces of information for the single stream management, with no changes in the well-consolidated TCP sequencing and acknowledgment standard mechanisms.

1) Multi-Stream Option

MS-TCP incorporates stream information as a new option field in each segment, supporting the stream multiplexing within the connection. This option includes:

- Option Kind: 1-byte size, value 253 [39]
- Option length: 1-byte size, value 4
- Last Stream Segment (LSS) flag: 1-bit size
- Stream Identifier (SID): 5-bit size
- Stream Segment Number (SSN): 10-bits size

The legacy TCP Sequence field, at the channel level, ensures that all the streams conveyed are sequenced correctly, with no need for additional per-stream indexes. However, the Stream Segment Number (SSN) adds a separate segment numbering for each stream to support an inter-stream Head of Line Blocking mitigation in combination with the Selective Acknowledgment (SACK) option as shown later. It is worth noticing that the SID and the SSN pertain to the payload data and the advertised window, but not to the acknowledgment information, that is at the channel level. The SSN width in conjunction with the TCP sequence number is robust against up to 1024 consecutive segments losses in a stream. The SID width supports up to distinct 32 streams during the channel life. No stream id reuse is allowed. The option space will be subtracted from the selective acknowledgement options.

2) Advertised Window

The advertised window field indicates the available receiver buffer space only for the stream identified by the SID.

3) Dummy Payload Flag

Some MS-TCP operations, such as stream start or end, or overflow management, require the transmission of out-of-band information even when there is no payload data to transmit or to fit in the receiving buffer. However, without an actual payload, the TCP channel sequence number could not advance to ensure reliable acknowledgment delivery. Therefore, MS-TCP enables the use of "dummy payload" segments that carry a dummy payload, allowing for the assignment of a unique sequence number. When the receiver transfers the in-order segment payload to the stream buffer, it skips over the payload of any segments marked as "dummy payload". The flag is saved in the TCP header reserved flag field [40] and applies only in presence of the MS-TCP option.

4) TCP Segment Payload

Each MS-TCP segment is specifically assigned to carry a consecutive portion of one stream only, so that more distinct streams cannot share the payload space within an MS-TCP segment.

C. Stream Opening

When a new socket attempts to establish a connection, the *connect* service examines the file descriptor table of the calling process to locate another existing socket, bound to the same local address, and connected to the same remote port, and remote address through a MS-TCP connection. If no match is found then Case 1 applies, otherwise Case 2. The two cases are detailed in the following.

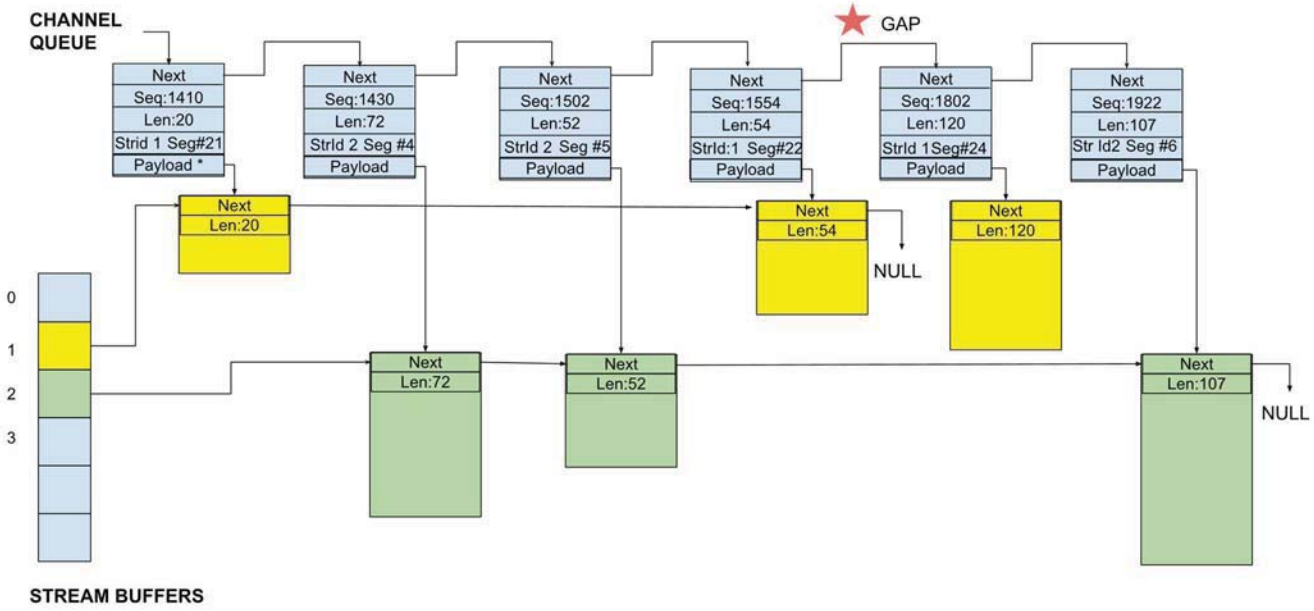

Fig. 1. Receiving buffer data structure

1) Case 1: Opening the first stream in a new multi-stream channel with compatibility check for Legacy TCP.

The active opener sends a SYN-flagged segment including the multi-Stream option. If the remote peer TCP supports the MS-TCP too, it recognizes and confirms the multi-Stream option in the SYN+ACK-flagged segment, marks the TCB as multi-stream, and allocates the MS-TCP-specific structures. Otherwise, it replies with no multi-Stream option and both peers establish a conventional TCP connection, ignoring the multi-stream option. In both situations, the socket becomes connected at the end of the regular handshake.

2) Case 2: Opening a new stream within a multi-stream channel already established.

In this case, the new socket links to the TCB of the existing connection, with the state "Established", without further handshaking, as the channel is available. In this case, there are two possible scenarios.

Scenario 1 - the socket has upstream data to transmit immediately, which is the most common scenario as it corresponds to the HTTP use case. In this scenario, the data is directly sent with the next available SID in the multi-stream option, with zero round trip time wait.

Scenario 2 - the active opener application immediately waits for downstream data. Albeit a rare scenario, if no upstream data is sent after a given timeout time based on the connection estimated roundtrip time [41], a dummy-flagged segment is sent, with the proper multi-stream option. The dummy segment carries a one-byte payload that is not part of the stream, i.e., it is sequenced in the channel but contains only out-of-band information.

In both scenarios, the passive opener recognizes the new stream id, allocates a new pair of transmission and receiving buffer, and extends the Transmission Control Block (TCB) with the new stream adding an entry to the listening queue containing the link to that TCB and the new stream id. The *accept* service consumes it by returning the new connected socket linked to that TCB and stream id. In the scenario 2, if the segment has a dummy-flagged payload, the payload is dropped and not transferred into the stream buffer. While a stream is in the listening queue, any received data is stored in the newly allocated buffer up to its capacity. This buffered data is ready to be consumed once the stream is accepted. In case of listening queue overflow, the passive opener sends an in immediate dummy payload with SSN=0 and the Last Stream Segment Flag set, causing a broken pipe exception in write attempts or an end of file in read attempts, at the active opener side.

D. Receiving Buffer

The receiving buffer is organized into two main inter-related components: the channel queue, and the stream buffers (see Fig. 1). The channel queue is designed as a linked list of "segment info" nodes, each reporting the segment sequence number, the payload size, the SID, and the SSN. Each sequence node is linked to the following node of the list, according to the TCP sequence number ordering. Each segment node also links to one payload node, containing the payload length and the payload data of the corresponding segment. Each payload node links to the payload node of the subsequent (i.e., SSN+1) segment of the same stream, if present, otherwise to NULL. At each new incoming segment, the handler scans the ordered list to insert the corresponding segment information node in the proper order, according to the TCP sequence. During the scanning process, the receiver keeps track of the last scanned node that shares the same SID as the received segments. If the SSN of the last scanned node directly precedes that of the new segment, the receiver links the payload node of the last scanned node to the newly inserted payload node.

The stream buffer vector is the entry point of the payload data for each stream, i.e., the vector element *i* links to the first unconsumed payload node of the corresponding stream SID = *i*. When a new segment is inserted, if there is no other payload node having the same SID, the stream buffer vector element links the payload node. The linked list of payload nodes reflects the continuous stream portion to be consumed. An additional stream state variable records the offset first character to be consumed in the first linked payload node.

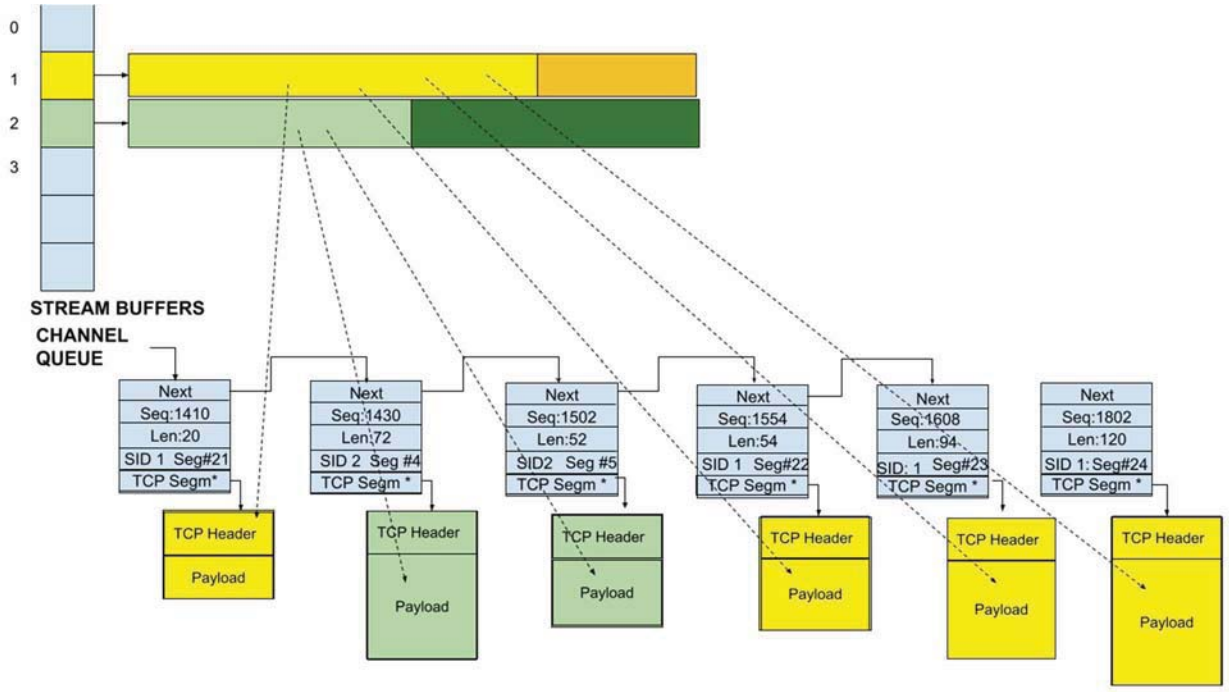

Fig. 2. Transmission buffer data structure

The *read* system call selects the stream buffer vector element corresponding to the socket stream identifier and consumes the linked payloads up to the NULL or to the maximum byte counts requested by the caller. When a payload node is completely consumed is erased, and the corresponding segment info node is maintained, changing the payload pointer to NULL. The segment info node is erased only when a cumulative acknowledge has a value greater or equal to the segment info node sequence plus the payload size. It is worth noticing that there can be payload nodes surviving over their corresponding segment nodes and vice-versa, allowing the channel sequence number to advance even in case of slow consumers, or allowing the buffer consumption to advance beyond sequence gaps affecting only other streams.

E. Transmission buffer

The transmission buffer is organized with both the channel queue and the stream buffers (Fig. 2). The stream buffers, which are circular buffers implementing linear byte buffers, are fed by the *write* system call. A scheduler is assigned the task of consuming data from the stream buffers and arranging the segments to ensure that the channel queue is filled up to the congestion window size. During the creation of segments, the scheduler prioritizes consuming data from the stream having the smallest "flight size", which corresponds to the fewest bytes sent and not yet acknowledged [42], to guarantee the inter-stream fairness. The segments are created with the complete header and additional MS-TCP option fields (SID, SSN). When a transmission request or retransmission timeout occurs, the transmission routine retrieves the pre-prepared header segments from the channel queue. The routine then updates the ACK field, Window Field, and checksum before passing them to the IP layer. Re-segmentation is not allowed, as the sequence numbers are a channel property that must be kept coherent among all the streams.

F. Selective Acknowledgment

The Selective Acknowledgment (SACK) in the context of a per-stream flow control independence from channel congestion control, plays a crucial role in mitigating the inter-stream HoL Blocking interference.

1) SACK in regular TCP

The SACK was originally designed [43] [44] as an option to reduce unnecessary segment retransmissions and not as an alternative to the cumulative acknowledgment mechanism. Therefore, the regular TCP sender, although suspending the retransmission of the selectively acknowledged ("sacked") segments, does not remove them from the transmission buffer considering that the receiver could retract (or "renege") the selected acknowledgment. The receiver has the chance of removing out-of-order segments from the receiving buffer even if sacked.

2) SACK in MS-TCP

In the event of lost segments, SACK information in MS-TCP can reveal the sender which streams have encountered gaps in the sequence and which ones have not. MS-TCP regards sacked segments, which are ordered for a single stream, but out of order for the TCP channel, as ultimately acknowledged since they can be promptly consumed by the application. This differs from TCP's SACK policy, whereby the sender keeps such segments in the transmission queue. Hence, the MS-TCP sender removes the segments and utilizes the available buffer space to advance the corresponding stream(s), in compliance with the TCP channel's congestion window size and with the latest per-stream advertised windows received. As a result, inefficiencies in packet loss recovery only affect the streams experiencing packet loss, thus eliminating the HoL blocking issue.

Although it is well known that SACK can be uncomplete when multiple sparse holes occur in the sequence, we observe that, when packet losses are frequent and sparse, the congestion appears to be severe, so as the congestion control should intervene at the channel level, across all the streams. Although further research will define proper strategies to decide when to extend the congestion prevention, whether to a part or to all the streams, the general underlying trade-off is

favorable as the SACK support is more robust when it is needed the most, i.e., for single or occasional packet losses. This trade-off also mitigates the drawback of subtracting option space from the SACK option field to support the Multi-Stream option.

G. MS-TCP Flow Control

Like in TCP, the advertised window serves as the primary means for managing flow control in MS-TCP. We analyze how the advertised window is managed in three distinct cases.

1) In-order incoming segment

The first case involves receiving a segment in the correct sequence order without any preceding channel gap, i.e., with no missing segments. In this instance, the acknowledgment number and advertised window size are transmitted in accordance with standard protocol rules. The MS-TCP advertised window size refers to the amount of free space available in the stream buffer, which is specified in the option fields. To prevent the delayed acknowledgment mechanism from restricting the advertised window refresh, the delayed acknowledgment timeout [10] is triggered on a per-stream basis. As a result, acknowledgments are typically sent every other segment of the same stream rather than every other segment of the channel. This rule keeps the acknowledgments rate remains consistent with that of equivalent streams running over separate connections.

If a receiver fails to receive acknowledgment segments that may contain important advertised window refresh information, it can push a window update for a specific stream, particularly if the latest received advertised window size was zero. To achieve this, a dummy segment can be utilized, as the MS-TCP sender is assumed never to surpass the stream buffer space through zero-window probing segments[10].

2) Out-of-order incoming segments

In the second case, an out-of-order incoming segment keeps the acknowledgment unvaried. However, a stream, receiving an out-of-order segment, may detect that there are no stream gaps even in not yet acknowledged segments. In that case, when acknowledging that segment, it sends the unvaried ACK, the regular SACK option, and the advertised window for the SID of that segment. The sacked segments are ready to be consumed accessing the corresponding linked payload nodes. Even when there are no gaps in the sequence for that stream, the MS-TCP advertised window still pertains to the cumulative acknowledgment number, similarly to the regular TCP, according to the following expression:

$$Adwin_i = B_i - (Offs_{ACKi} - Offs_{CONSi}) \qquad (1)$$

where the B_i, $Offs_{ACKi}$ and $Offs_{CONSi}$ indicate, for stream i, the total buffer size, the highest cumulatively acknowledged stream byte position and the first unconsumed byte positions, respectively. This implies that although the receiving buffer of a stream verifies the continuity of a stream portion beyond the acknowledged sequence, the MS-TCP does not include this portion in the stream buffer occupancy to compute *Adwin*. However, when the stream portion is consumed it contributes to the growth of the advertised window. Therefore, in regular TCP, $Offs_{CONS} \leq Offs_{ACK}$ whereas, in MS-TCP, $Offs_{CONSi}$ can be greater than $Offs_{ACKi}$ so that $Adwin_i$ can be greater than B, as shown in the example of Fig. 3. Interleaved segments from streams A and B operate over a TCP channel until stream A loses a segment with a sequence number X. Since the Acknowledgment in TCP is cumulative, it cannot exceed sequence number X, even though subsequent segments with a sequence greater than X from both streams are received. While the stream B segments beyond X are continuous, on the contrary, the stream A segments are not.

The gap in A makes $Adwin_A < B_A$, as no data after the sequence number X can be consumed. In contrast, the data of stream B including the sacked segment data, can be consumed. Consequently, the advertised window for stream B reflects a space that can exceed the buffer size by the total amount of the sacked segment sizes.

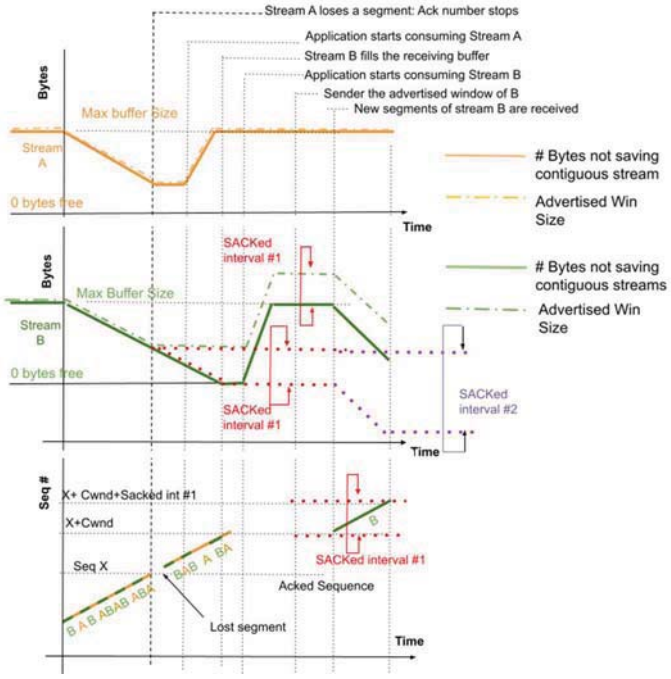

Fig. 3. Advertised Windows and buffer occuparion in double stream MS-TCP connection

For a better readability, the example assumes that the stream reception and consumption happen in separate moments. In normal cases the advertised window computed by expression (1) normally balances the consumption and the reception also when happening concurrently.

On the transmitting side, when the SACK is received, the transmission buffer removes the sacked payload node data, but not the corresponding segment info nodes, and computes the transmission window boundaries as $X + cwnd + S$, where *cwnd* is the congestion window size in bytes and S is the total size of sacked segments in bytes. Although the advertised window is still locked to the sequence number X, however the flow control is resilient and allows stream B to proceed to avoid the Head of Line Blocking caused by stream A packet loss. The segment info nodes persist as pure sequence placeholders until the cumulative acknowledgment overpass X and removes the info nodes of the acknowledged segments. We finally remark that $Offs_{ACK}$ and $Offs_{CONS}$ in single stream connections are regular TCP sequence data in the TCB, whereas in $Offs_{ACKi}$ and $Offs_{CONSi}$ are stream state variables bookkept in the MS-TCP whenever a cumulative acknowledgment removes the segment info nodes of the corresponding stream i and a *read* for the stream i consumes new payload node data, respectively.

3) Late incoming segments

In the third case, a late segment, either retransmitted or not, restores the sequence continuity by filling a gap and the acknowledgment number grows by two or more segment sizes. In this case the MS-TCP needs updating the advertised window size for each stream whose segments are cumulatively acknowledged and removed from the receiving buffer. During the segment info nodes removal, the MS-TCP builds a list of advertised windows for each stream involved in the cumulative acknowledgment and keeps track of the highest TCP (channel) sequence number for that stream. After the removing the segment info, a separate acknowledgment is sent, with or without payload, for each element of the list reporting the SID, the advertised window and the acknowledgment number corresponding to the highest sequence number saved.

H. Receiver buffer Overflow Exception

The event of buffer overflow is an exception for which the MS-TCP cannot simply ignore the segment as the channel sequence cannot be blocked. The overflow handling in MS-TCP works as follows. 1) When an incoming segment's payload would exceed the stream buffer limit, a segment info node is created with no payload node linked, and with an internal overflow flag set. 2) When an overflow-flagged segment info node is to acknowledge because it is in order, the ack number is set equal to that segment sequence number, the corresponding stream ID (SID), and an advertised window set to zero. This acknowledgment indicates that the segment is not acknowledged and that there is no room for any payload at that sequence number. 3) Upon receiving this flow control information, the transmitter retransmits the segment with the "Dummy" flag set, to nullify the payload. The transmission buffer also unlinks the payload node from the segment info node and keeps it as a pending payload node. 4) When the retransmitted dummy segment is received, the overflow flag is cleared, and that segment, along with possibly other subsequent segments, can be cumulatively acknowledged. This allows the channel sequence to progress. 5) When the transmitter receives the next notification of a non-zero advertised window with sufficient space, it retransmits the pending payload node, linking it to a new channel sequence number. It is worth noticing that latencies of this exception handling do not prevent the SACK from making the other streams advance.

I. Stream Close

Like other TCP flags, the FIN flag is associated with the channel sequence number, and thus, it is utilized to terminate the entire channel. Typically, it is employed after closing all the streams within the channel. As opposed, to close an individual stream, the transmitter informs the receiver that a stream segment sequence sent is the last one, using the Last Stream Sequence flag in the MS-TCP options (see Section B.1) and the receiver acknowledges that last segment sequence to the peer transmitter either via the channel cumulative acknowledgment or the SACK. If the last data segment is empty, the dummy payload flag is set. The full-duplex stream closes after both last segment acknowledgments are received in both directions, which can occur in distinct time spans, as in regular TCP close. When the last segment is acknowledged for both directions, the stream socket is closed the stream buffers are freed. In the current design the SIDs are not reused, so that when the only active stream is closed the TCP connection is closed consequently.

V. Discussion

The MS-TCP design presented in this paper shows that a backward compatible moderate revision of the TCP header, an appropriate design of receiving and transmission buffers, a few punctual sematic changes in the Advertised Window and in the Selective Acknowledgment, make the TCP sequencing and acknowledging legacy a still valid framework to set up a reliable, fully compatible, and congestion-managed channel, ready to carry multiple streams each having its own independent and flow control, robust to HoL blocking against other concurrent streams in case of occasional out-of-order deliveries or packet losses.

Previous experiences of transport layer protocols having some commonalities with the MS-TCP have been the TCP Fast Open [17], aiming at exploiting the handshake packets to promptly include stream data, creating a session cookie but maintaining one stream per connection, the SCTCP [18] an UDP-based alternative to carry multiple concurrent streams on different network paths, with no attention, like QUIC, to any TCP compatibility, and multipath TCP extensions [19] to support high performance streams sending packet through different network paths. We finally mention here a work [20] proposing a misleadingly homonymic “multi-stream TCP”, an upper-level mechanism that split an individual stream into multiple regular TCP connections (flows) each routed across different network paths by a Software-Defined Network.

We believe that the absence of a TCP multi-stream support in literature, is due to the past perception that such a solution would have been biased towards a specific application protocol, namely the web, and thus deemed unworthy of influencing the transport layer. This motivated the HTTP-sided adaptations surveyed in Section II.A. However, HTTP today appears as a transport protocol sublayer encompassing an extra-dominant portion of the network data transmissions. As a result, the influence of the HTTP over the transport protocol now appears fully consistent. In this perspective, we notice that such critical aspects as congestion management and network compatibility, which impact network stability, require meticulous software versioning and testing, that is in our opinion the good side of the “ossification”, and strongly suggests their retention within the hands of kernel developers rather than being delegated to application developers. We believe that the risk of inappropriate task mapping is occurring due to the inadequate harmonization between the application and the network operating system functionalities, that is precisely what MS-TCP aims to facilitate.

VI. Conclusion and Future Work

The design of the MS-TCP including multi-stream sockets, has been proved using an experimental user-space implementation over raw sockets to run on real networks. The future work addresses i) Extensive measurement and comparisons with QUIC, ii) Transport Layer Security (TLS) adaptation to the MS-TCP to reuse the existing session keys to add new stream to an existing channel with zero overheads, iii) research on the stream fairness policies in the transmission scheduler and in the rules for maximum allowed sacked data size per stream, also capitalizing the QUIC literature, iv) MS-TCP porting to the Linux Kernel for performance tuning.

References

[1] I. Grigorik “HTTP protocols”, O'Reilly, 2017.

[2] T.M. Chester, "Cross-platform integration with XML and SOAP," IT Professional, vol. 3, no. 5, pp. 26-34, 2001.

[3] M. Masse, "REST API design rulebook: designing consistent RESTful web service interfaces," O'Reilly Media, Inc., 2011..

[4] F. Halili and E. Ramadani, "Web services: a comparison of soap and rest services," Modern Applied Science, vol. 12, no. 3, pp. 175, 2018.

[5] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger, "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <https://www.rfc-editor.org/info/rfc7323>

[6] A. Briones, A. Mallorquí, A. Zaballos, and R. Martin de Pozuelo, "Adaptive and aggressive transport protocol to provide QoS in cloud data exchange over Long Fat Networks," Future Generation Computer Systems, vol. 115, pp. 34-44, 2021.

[7] M. Bishop, "HTTP/3", RFC 9114, DOI 10.17487/RFC9114, June 2022, <https://www.rfc-editor.org/info/rfc9114>.

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

[9] M. Seufert, R. Schatz, N. Wehner, B. Gardlo, and P. Casas, "Is QUIC becoming the New TCP? On the Potential Impact of a New Protocol on Networked Multimedia QoE," 2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX), Berlin, Germany, 2019, pp. 1-6, doi: 10.1109/QoMEX.2019.8743223.

[10] W. Eddy, "Transmission Control Protocol (TCP)", STD 7, RFC 9293, DOI 10.17487/RFC9293, August 2022, <https://www.rfc-editor.org/info/rfc9293>.

[11] J. Postel, "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[12] E. Hatzoglou, V. Kouliaridis, G. Karopoulos, et al., "Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study," Int. J. Inf. Secur., vol. 22, pp. 347-365, 2023. https://doi.org/10.1007/s10207-022-00630-6

[13] R. Marx, M. Piraux, P. Quax, and W. Lamotte, "Debugging QUIC and HTTP/3 with qlog and qvis," Proceedings of the Applied Networking Research Workshop, pp. 58-66, 2020.

[14] T. Shreedhar, R. Panda, S. Podanev and V. Bajpai, "Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads," in IEEE Transactions on Network and Service Management, vol. 19, no. 2, pp. 1366-1381, June 2022, doi: 10.1109/TNSM.2021.3134562.

[15] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same Standards, Different Decisions, 'A Study of QUIC and HTTP/3 Implementation Diversity'," Proceedings of the ACM Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ '20), pp. 14-20, 2020. https://doi.org/10.1145/3405796.3405828

[16] M. Kahlewind and B. Trammell, "Manageability of the QUIC Transport Protocol", RFC 9312, DOI 10.17487/RFC9312, September 2022, <https://www.rfc-editor.org/info/rfc9312>.

[17] Y. Cheng, J. Chu, S. Radhakrishnan and A. Jain, “TCP fast open”, Internet Engineering Task Force, RFC7413, 2014

[18] R. Stewart, M. Tüxen, and K. Nielsen, "Stream Control Transmission Protocol", Internet Engineering Task Force, RFC9260, 2022.

[19] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <https://www.rfc-editor.org/info/rfc6824>.

[20] N. Farrugia, V. Buttigieg and J. A. Briffa, "Multi-Stream TCP: Leveraging the Performance of a Per-Packet Multipath Routing Algorithm When Using TCP and SDN," 2019 IEEE 44th Conference on Local Computer Networks (LCN), Osnabrueck, Germany, 2019, pp. 218-221, doi: 10.1109/LCN44214.2019.8990704.

[21] M. Honda et al., "Is it still possible to extend TCP?," Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference, 2011.

[22] The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008),

[23] B. Carpenter and S. Brim, "Middleboxes: Taxonomy and Issues", RFC 3234, DOI 10.17487/RFC3234, February 2002, https://www.rfc-editor.org/info/rfc3234.

[24] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, DOI 10.17487/RFC1945, May 1996, https://www.rfc-editor.org/info/rfc1945

[25] R. Fielding, M. Nottingham, and J. Reschke, "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, https://www.rfc-editor.org/info/rfc9112, Section 9.3.

[26] P. Lubbers, B. Albers, F. Salim, and T. Pye, "Pro HTML5 programming," Apress, 2011.

[27] Diffusion Data™, "Diffusion Cloud 6.9.0 User Manual, Sect. 4.2," https://docs.diffusiondata.com/cloud/latest/manual/html/designguide/solution/support/connection_limitations.html.

[28] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <https://www.rfc-editor.org/info/rfc2616>.

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

[30] I. Grigorik, "Making the web faster with HTTP 2.0," Communications of the ACM, vol. 56, no. 12, pp. 42-49, 2013.

[31] W. Simpson, "The Point-to-Point Protocol (PPP)", STD 51, RFC 1661, DOI 10.17487/RFC1661, July 1994, <https://www.rfc-editor.org/info/rfc1661>.

[32] M. Scharf and S. Kiesel, "NXG03-5: Head-of-line Blocking in TCP and SCTP: Analysis and Measurements," IEEE Globecom 2006, San Francisco, CA, USA, 2006, pp. 1-5, doi: 10.1109/GLOCOM.2006.333

[33] G. Papastergiou et al., "De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives", IEEE Commun. Surveys & Tutorials, vol. 19, no. 1, 2017.

[34] S. Cook, B. Mathieu, P. Truong and I. Hamchaoui, "QUIC: Better for what and for whom?," 2017 IEEE International Conference on Communications (ICC), Paris, France, 2017, pp. 1-6, doi: 10.1109/ICC.2017.7997281

[35] S. Huang, F. Cuadrado and S. Uhlig, "Middleboxes in the Internet: A HTTP perspective," 2017 Network Traffic Measurement and Analysis Conference (TMA), Dublin, Ireland, 2017, pp. 1-9, doi: 10.23919/TMA.2017.8002906.

[36] J. Touch, M. Welzl, and S. Islam, "TCP Control Block Interdependence", RFC 9040, DOI 10.17487/RFC9040, July 2021, <https://www.rfc-editor.org/info/rfc9040>.

[37] A. Langley et al., "The quic transport protocol: Design and internet-scale deployment," Proceedings of the conference of the ACM special interest group on data communication, pp. 183-196.

[38] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[39] J. Touch, "Shared Use of Experimental TCP Options", RFC 6994, DOI 10.17487/RFC6994, August 2013, <https://www.rfc-editor.org/info/rfc6994>.

[40] M. West and S. McCann, "TCP/IP Field Behavior", RFC 4413, DOI 10.17487/RFC4413, March 2006, <https://www.rfc-editor.org/info/rfc4413>.

[41] V. Paxson, M. Allman, J. Chu, and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

[42] Allman, M., Paxson, V., and W. Stevens, "TCP Congestion Control", RFC 2581, DOI 10.17487/RFC2581, April 1999, <https://www.rfc-editor.org/info/rfc2581>.

[43] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options", RFC 2018, DOI 10.17487/RFC2018, October 1996, <https://www.rfc-editor.org/info/rfc2018>.

[44] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky, "An Extension to the Selective Acknowledgement (SACK) Option for TCP", RFC 2883, DOI 10.17487/RFC2883, July 2000, <https://www.rfc-editor.org/info/rfc2883>.