TCP Flow Control and Congestion

Ali Bahja

21th november 2024

1 Objective

The purpose of this lab is to analyze the mechanisms of **TCP flow control and congestion control** using Wireshark captures. The lab investigates how TCP adapts to receiver limitations, how it reacts to congestion, and how its mechanisms compare with Go-Back-N and Selective Repeat protocols.

2 Topology and Setup

The experiment uses two clients downloading data from a server. Wireshark captures TCP segments, ACKs, retransmissions, and congestion window dynamics. TCP graphs are analyzed to identify flow control and congestion control phases.

3 Procedure and Results

3.1 Flow Control (Client 2)

For *Client 2*, 424 packets are downloaded at an average speed of 273 KB/s. Wireshark shows reduced window sizes advertised by the receiver, which indicate that the receiver communicates its limitations to the sender. This activates the TCP flow control mechanism.

The high throughput and relatively low number of exchanged packets confirm that transmission is limited by the receiver's advertised window capacity rather than by network congestion.

3.2 Congestion Control (Client 1)

For *Client 1*, 531 packets are downloaded at an average speed of 59 KB/s. Wireshark shows multiple timeouts followed by drastic reductions of the congestion threshold. These events correspond to packet losses and reductions in the congestion window size.

When a lost packet is retransmitted, the throughput gradually increases thanks to TCP's AIMD mechanism. The congestion window grows incrementally after each successful RTT cycle.

The TCP throughput graph clearly shows :

— Slow start phase : rapid initial increase.

— Congestion avoidance : linear growth afterwards.

— Multiplicative decrease : window reductions after losses.

3.3 ACK Number Calculation

Packet 466 is the last correctly sent segment before Wireshark reports `[TCP Previous segment not captured]`.

Packet 490 is the ACK of the retransmitted segment after the loss.

No.	Time	Source	Destination	Protocol	Length	Info
461	0.118387	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=411510 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
462	0.118390	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=412963 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
463	0.118408	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [PSH, ACK] Seq=414416 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
464	0.118412	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=415869 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
465	0.118528	195.11.17.5	195.11.16.5	TCP	54	55992 → 80 [ACK] Seq=132 Ack=417322 Win=13077 Len=0
466	0.120856	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=417322 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
467	0.120865	195.11.16.5	195.11.17.5	TCP	1507	[TCP Previous segment not captured] 80 → 55992 [ACK] Seq=420228 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled…
468	0.120873	195.11.17.5	195.11.16.5	TCP	54	55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
469	0.120882	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [PSH, ACK] Seq=421681 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
470	0.120886	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#1] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
471	0.120891	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=423134 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
472	0.120894	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#2] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
473	0.120902	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=424587 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
474	0.120908	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#3] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
475	0.120916	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=426040 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
476	0.120922	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#4] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
477	0.120929	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=427493 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
478	0.120935	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#5] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
479	0.120945	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [PSH, ACK] Seq=428946 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
480	0.120950	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#6] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
481	0.123314	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=430399 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
482	0.123326	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#7] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
483	0.123336	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=431852 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
484	0.123339	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#8] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
485	0.123345	195.11.16.5	195.11.17.5	TCP	1507	[TCP Window Full] [TCP Previous segment not captured] 80 → 55992 [PSH, ACK] Seq=434758 Ack=132 Win=64109 Len=1453 [TCP s…
486	0.123348	195.11.17.5	195.11.16.5	TCP	54	[TCP Dup ACK 468#9] 55992 → 80 [ACK] Seq=132 Ack=418775 Win=17436 Len=0
487	0.123355	195.11.16.5	195.11.17.5	TCP	1507	[TCP Fast Retransmission] 80 → 55992 [ACK] Seq=418775 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]
488	0.123363	195.11.17.5	195.11.16.5	TCP	54	55992 → 80 [ACK] Seq=132 Ack=433305 Win=10171 Len=0
489	0.125751	195.11.16.5	195.11.17.5	TCP	1507	[TCP Retransmission] 80 → 55992 [ACK] Seq=433305 Ack=132 Win=64109 Len=1453
490	0.125765	195.11.17.5	195.11.16.5	TCP	54	55992 → 80 [ACK] Seq=132 Ack=436211 Win=17436 Len=0
491	0.125775	195.11.16.5	195.11.17.5	TCP	1507	80 → 55992 [ACK] Seq=436211 Ack=132 Win=64109 Len=1453 [TCP segment of a reassembled PDU]

FIGURE 1 – Wireshark Report

The final ACK number is computed as :

$$ACK_{final} = SEQ_{last} + (13 \times MSS)$$

where the last correct SEQ is 417322, MSS = 1453, and 13 packets follow.

$$ACK_{final} = 417322 + (1453 \times 13) = 417322 + 18889 = 436211$$

This confirms that all retransmitted segments are acknowledged and that transmission resumes normally.

3.4 Comparison with Go-Back-N and Selective Repeat

TCP shares similarities and differences with classical ARQ protocols :

Similarities :

— Like Go-Back-N, TCP uses a sliding window and retransmits lost segments after timeouts.

— Like Selective Repeat, TCP retransmits only lost segments, minimizing unnecessary retransmissions.

Differences :

— Unlike Go-Back-N, TCP does not retransmit segments already received correctly, thanks to Fast Retransmit triggered by duplicate ACKs.

— Unlike Selective Repeat, which uses per-packet ACKs, TCP uses cumulative ACKs, simplifying implementation but making multiple loss detection less precise.

— TCP integrates congestion control (AIMD, slow start), which neither Go-Back-N nor Selective Repeat provide.

4 Analysis

The experiments confirm that flow control limits throughput according to receiver capacity, while congestion control dynamically adapts to network conditions. TCP effectively combines reliability, retransmission, and congestion avoidance. Compared to ARQ protocols, TCP is more efficient for real-world networks since it integrates congestion control in addition to error recovery.

5 Conclusion

This lab demonstrates how TCP implements both flow and congestion control. Client 2 highlights receiver-limited throughput (flow control), while Client 1 illustrates congestion avoidance with slow start, AIMD growth, and retransmissions. The ACK analysis confirms correct delivery after losses. Finally, the comparison with Go-Back-N and Selective Repeat shows how TCP borrows from ARQ strategies but extends them with congestion management.