Universidad de León
School of Industrial, Computer and Aerospace Engineering
Course on Distributed Systems

1.0. Example about TCP RTO Timer when no packet loss occurs

This example illustrates a successful transmission of several segments accomplished by sender host H_t to receiver host H_r. None of the sent segments is dropped, consequently, neither the RTO timer fires nor the 3-DUP mechanism is ever activated.

Host H_t requests a TCP connection with host H_r; the relevant handshake begins at time index 0. At time index 1, the handshake has finished and data interchanges can begin from either side; we'll assume that data transmission only takes place from host H_t to Host H_r so as to keep the example simple and at the same time significant. At this time, right after initializing the connection, TCP is in the Slow Start state, consequently, host H_t sends a single segment in the first round trip.

In the handshake, H_t and H_r have set their respective MSS to 1000 bytes, which is such an unusual value for MSS, however, we have chosen it for it's straightforward to calculate with at the same time helping us understand that 1000 is functionally acceptable as MSS. The first segment has SN = 1; again, keeping the examples simple leads us to use *relative Sequence Numbers,* like tcpdump and Wireshark do. The segment sent at index 1 has a length of 1000, in conformity with the MSS announced by H_r in the handshake. Segments like the former which size is the same as the MSS are conventionally known as *full segments*. The range of sequence numbers covered by this segment is: [SN, (SN + Len) – 1] = [1, (1 + 1000) - 1] = [1, 1000]. This segment, shortly after it is received by host H_r, causes it to send back an ACK for it; since the last byte covered by [1, 1000] is 1000, the ACK must have an ACK SN of 1001, following the TCP ACK semantics of "next in-sequence byte expected".

At index 1, right before the first segment is sent, an RTO (Retransmission TimeOut) timer is created and started so that it protects the transmission of the segment. The timer is free running at this time. Also, at index 1, the TCP transmitter snd.una state variable is set to 1 (The first unacknowledged byte)[1].

[1] The nomenclature used for TCP stack transmitter and receiver state variables follows the conventions set in RFC 793 under heading 3.3 "Sequence Numbers". The name of the **snd.una** variable in the Textbook by Peterson and Davie is LastByteAcked.

```
Send Sequence Space

                   1          2          3          4
              ----------|----------|----------|----------
                     SND.UNA    SND.NXT    SND.UNA
                                          +SND.WND

        1 - old sequence numbers which have been acknowledged
        2 - sequence numbers of unacknowledged data
        3 - sequence numbers allowed for new data transmission
        4 - future sequence numbers which are not yet allowed
```

Figure 0.1. Send Sequence Space from RFC 793 (*A verbatim copy thereof*).

ACK 1001 is received at time index 2; since it has advanced snd.una from 1 to 1001 (*It's an ACK that advances 1000*) the RTO timer is restarted so that the ensuing segment to be transmitted is protected. Sender host H_t is still in the Slow Start state, conequently, it will transmit a maximum of twice the number of bytes that were acknowledged in the received ACK, in the present case 1000 so that it can proceed with transmitting the next 2 x 1000 = 2000 bytes; since the MSS=1000, it can transmit a total of 2 full segments (A full MSS). With RTO timer started at 2, transmission proceeds with two segments which respectively have sequence numbers SN=1001 and SN=2001 respectively and both have a length of 1000 bytes (Again, a full MSS).

Assuming that the two back-to-back segments transmitted in time index 2 make it to the receiver, it reacts by sending a single Delayed Acknowledgement (DelAck) which cumulatively and positively ACKs the two latest received back-to-back segments. H_r generates *a single DelAck* because the two received segments have consecutive sequence numbers and the second one arrived at H_r before the *delack timer* elapsed (This timer is started by the receiver when it receives the first segment and may have a length of time of about 200ms).

At 4, the DelAck arrives, and since it moves snd.una to 3001 (An advancement of 2000 bytes from the former snd.una of 1001, *i.e.* equivalent to two full segments). Reception of the *advancing* ACK causes the RTO timer to be restarted. Since the ACK advanced snd.una by two MSS and being in the SS state, the transmitter can send at most (2 x 2 received MSS = 4 full segments). Right after time index 4, the transmitter proceeds to transmit 4 segments, the first of which carries SN = 3001.

Shortly after time index 4, the TCP at H_t initiated the transmission of the 4 segments, one by one which were successfully handed to receiver H_r. The first two of these segments arrived at H_r within less than the *delack timer* seconds, which made H_r to send back a single ACK for the two, *viz.* a delayed ACK or *delack*. The burst of 2 further packets arriving after the latter two also made H_r to send back a delack. The first of the latter two ACKs (A delack in this case) caused the RTO timer to be restarted, by contrast the second ACK did not cause a restart but a timer stop because no more data is waiting to be sent in H_t's send buffer.

All rights reserved © by José María Foces Morán and José María Foces Vivancos 2020-2021

2.0. Example about TCP RTO Timer-based retransmission where one packet gets dropped

Assume that the transmission of the segment with SN=2001 occurring shortly after time index 2 in Fig. 1 results in the segment being dropped. This packet loss is represented in Fig. 2, which is a continuation from Fig. 1 in which we assume the aforementioned segment loss (See time index 4).

Observe at time index 6 that we are assuming that the DelAck timer fires before another segment carrying a *contiguous* SN arrives, consequently, TCP at H_r sends back an ACK (SN 2001) for the data received in the preceding segment which SN=1001 (Recall that all the segment lengths considered in this example are of 1000 bytes in length, or a full segment). At time index 7, the ACK 2001 arrives at H_t; as is expected in TCP, that ACK causes the RTO timer to restart. Right after time index 7, the TCP at H_t sends a number of bytes that is twice as big as the advance of snd.una produced by the preceding segment which carried ACK 2001. Segments with SN=3001 and SN=4001 are transmitted, both of which have a payload length of 1000 bytes.

At time index 9 the two segments are received by H_r. Observe that the first segment (SN=3001) is an *out-of-order* segment, that is so because the maximum level of progress in H_r's buildup of the stream of data received from H_r is at 2000, that is, the next expected byte at H_r is 2001, not at 3001. The stipulated behavior of a receiver when receiving an *out-of-order* segment consists of sending back a QuickACK (A single ACK sent immediately, which, as usual carries an ACK SN representing the receiver's current value of rcv.nxt[2]). Following the prescription just explained, the delivery of a segment carrying data from SN=4001 at H_r causes that host to send back one more QuickAck (See time index 10).

The two QuickAcks are received by H_t; observe that these two ACKs are *duplicates* of the ACK sent at time index 6, *i.e., two duplicates* of it. *Not 3-DUP!* Conceptually, 2 duplicates won't attain retransmission of the segment at snd.una as 3-DUP does[3]. Faithfully complying with the specifications from RFC 5681 requires that 3 duplicates of an ACK segment be received for the sender to retransmit the segment at snd.una. The three duplicates must have the same ACK SN and the same AWS as the original ACK and it should carry no payload. In summary, no 3-DUP retransmission will be started at H_t.

[2] **rcv.nxt** on the receive side is equivalent to Peterson and Davie's **NextByteExpected**

[3] Actually, in a number of versions of the Linux TCP/IP stack, 2 duplicates, when received by the sender spur the retransmission of the segment at snd.una.

For completeness, we should observe that the two last ACKs with SN=2001 don't advance, therefore, neither can be used by H_t for sending further segments should there be any in the transmission buffer of H_t; ultimately, the H_t-to-H_r side of the TCP connection becomes idle since there remain no more ACKs pending to be sent back by H_r. Observe further that indeed there are pending ACKs, all of ACK SN=3001, SN=4001 and SN=5001, therefore, we cannot think that this direction of the connection idling is against the prescriptions of the Nagle's algorithm. Do you wonder how is this idling of the connection broken down? The key is in the non-advancing ACKs received, which won't restart the RTO timer, so, please check Fig. 2 where you'll readily identify a single restart of the RTO Timer (Time index 7); after that point in time, the RTO timer will free run until the countdown is exhausted at time index 11. That is the mechanism that will get the connection out of idling state: the retransmission of SN=1001 (Time index 12) after RTO timer fires at time index 11.

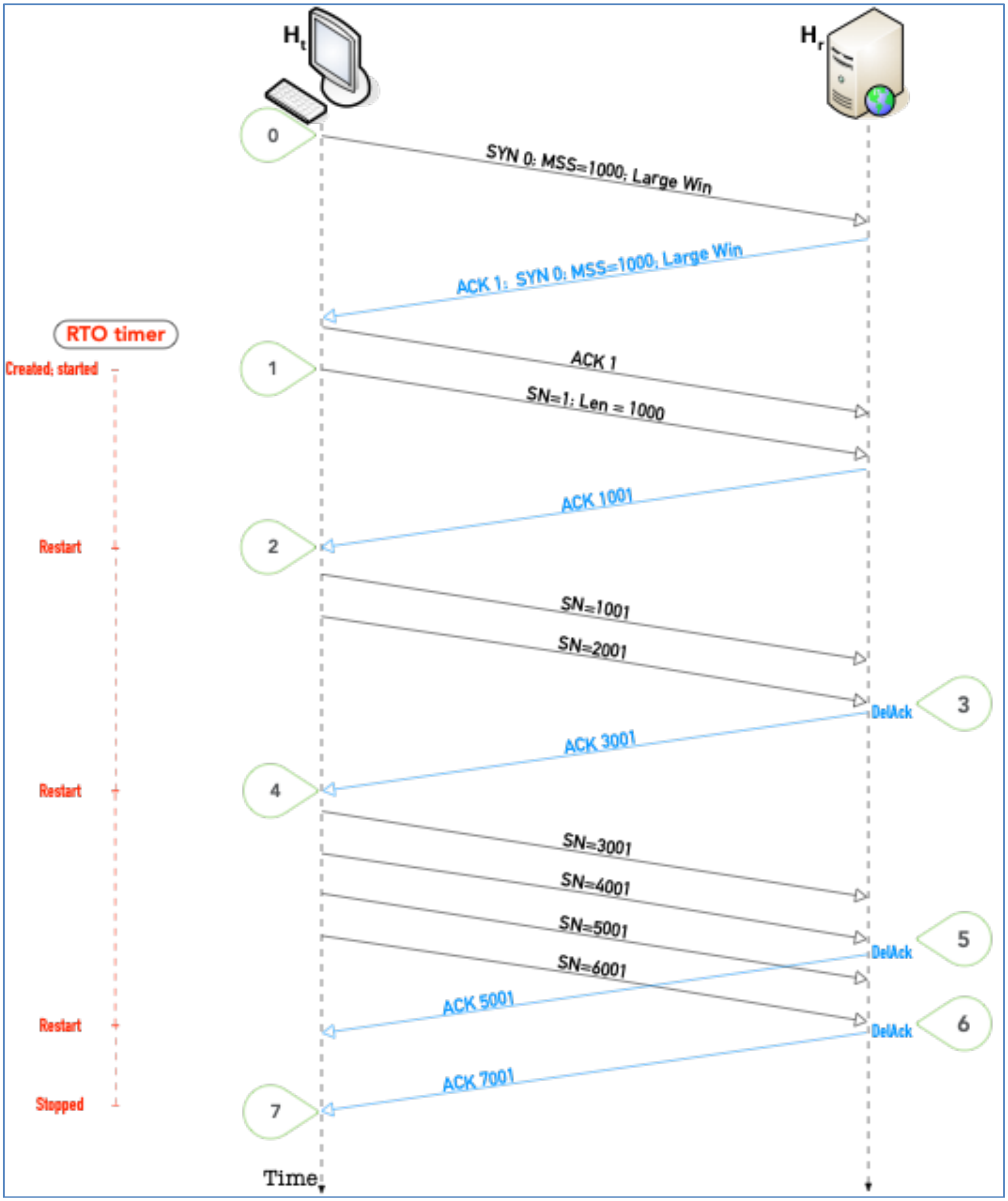

Figure 1. 3-way handshake and ensuing transmissions

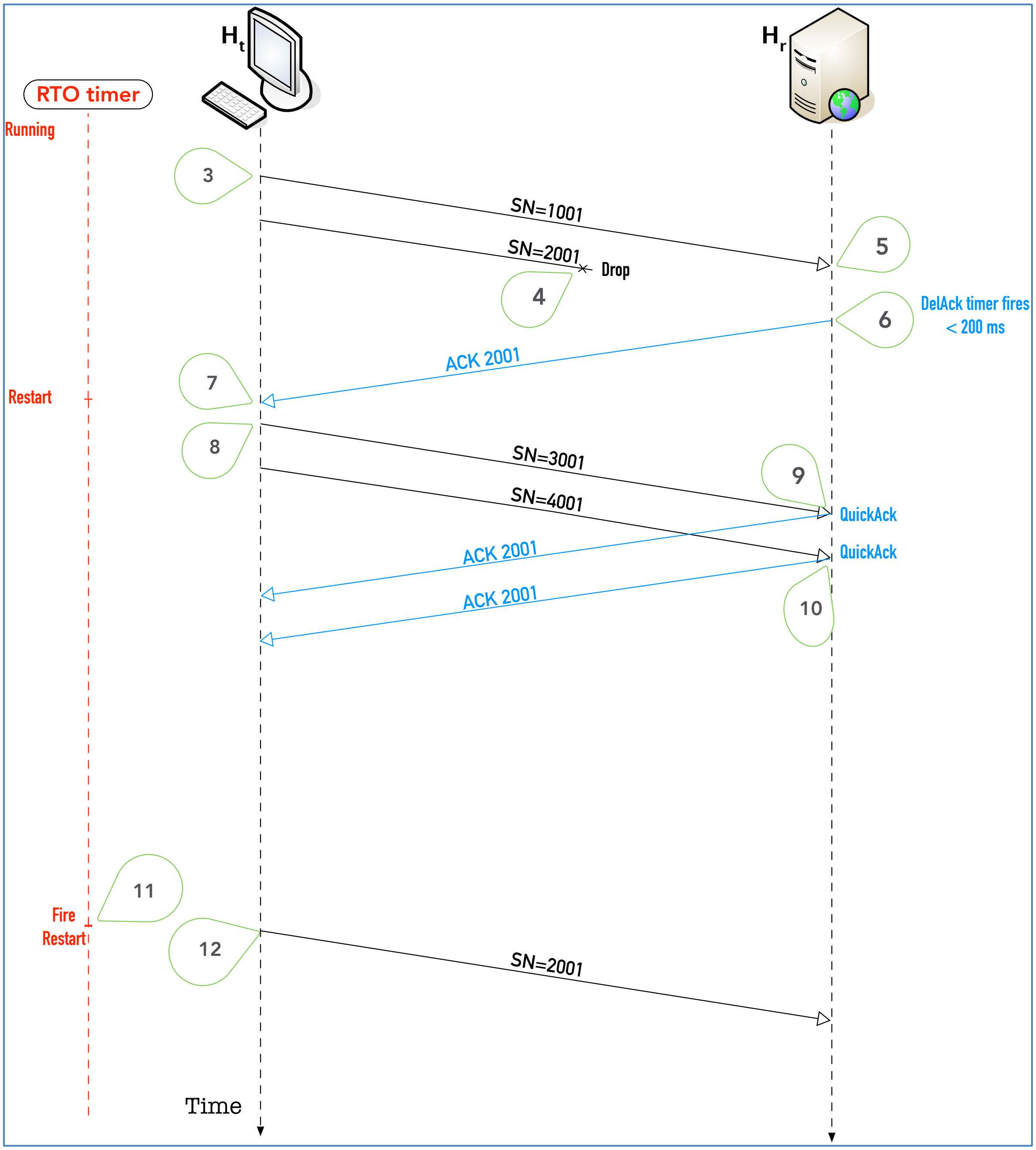

Figure 2. Continuation of the example in Fig. 1 assuming segment with SN=2001 gets dropped at time index 4.

