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 (On the Ht to Hr side) is in the Slow Start state, assuming that the application at H_t has plenty of bytes to send across the connection, consequently, host H_t sends a very small number of MSS-sized segments, or a single MSS-sized segment, *i.e.* 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 because 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 latter 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-order 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.

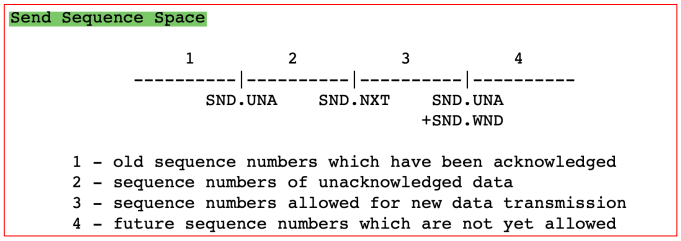

Figure 0.1. Send Sequence Space from RFC 793 (*A verbatim copy thereof*).

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 bytes forward in the stream. Technically, it is referred to as an ACK that advances*) the RTO timer is restarted so that the ensuing segment to be transmitted is protected. Sender host H_t's TCP is still in the Slow Start state, consequently, it will transmit a maximum of twice the number of bytes that were acknowledged in the received, advancing ACK which acked 1000 bytes (A full MSS-sized segment) so that it can proceed to transmitting the next 2 x MSS = 2 x 1000 = 2000 bytes; since the MSS=1000, it can transmit a total of 2 full segments (One full MSS, each). 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, each a full MSS).

Linux TCP receivers avail of two acking modes: QuickAck and DelAck. In QuickAck mode, Linux TCP returns an ACK immediately, right after it receives some types of TCP segments, for example: a segment that fills a receive buffer gap, a segment which SN is not contiguous to the latest received in-order segment. There exist other kinds of TCP segments that when received cause the receiver to send back a QuickAck. The default Linux acking mode is DelAck, however. In DelAck mode, TCP will wait an additional 200 ms after receiving a TCP segment. That time, 200 ms is known as the DelAck timer, and it allows TCP to wait for a would-be next in-order segment, so that, if that next segment actually arrives, TCP has sent a single ACK for the two in-order segments, thereby saving some network bandwidth. The DelAck mode in Linux can be configured to wait for up a specific number of milliseconds for the next in-order segment, thereby achieving even greater bandwidth savings. The DelAck timer can be set to 10 ms with the following command:

```
# echo 10 > /proc/sys/net/ipv4/tcp_delack_min
```

The default acking mode when initializing a connection in the SS (Slow Start) state is QuickAck which will allow TCP to determine the *congestion window* quickly.

However, despite the default acking mode is QuickAck in the SS state, in this example we will assume DelAck as the used acking mode. Keep this on mind!

Continuing with the chronogram, 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 latest two 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 (It is an "*Ack that advances*", the advancement being of a length of 2000 bytes from the former snd.una of 1001, *i.e.* equivalent to two full segments assuming the example's MSS of 1000 bytes). 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 (Slow Start), the transmitter can send at most 2 x 2 received MSS = 4 full segments. Right after time index 4, the transmitter proceeds to transmitting 4 segments, the first of which carries SN = 3001. This segment is the next to be transmitted within the current window: Note SNs through 3000 have already been *transmitted*, consequently it's SN=3001 the next. Notice it's only the received advancing Acks that spur the transmitter to continue the transmission of the segments that comprise the current transmission window (Pointed to by send buffer variable snd.nxt and which end is at is snd.una + snd.win). In the present example the transmission window size (snd.win), in TCP's SS state is twice the number of bytes cumulatively Acked (Those for which a corresponding advancing Ack has been received).

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 an RTO restart but an RTO timer stop because no more data are pending to be transmitted in H_t's send buffer.

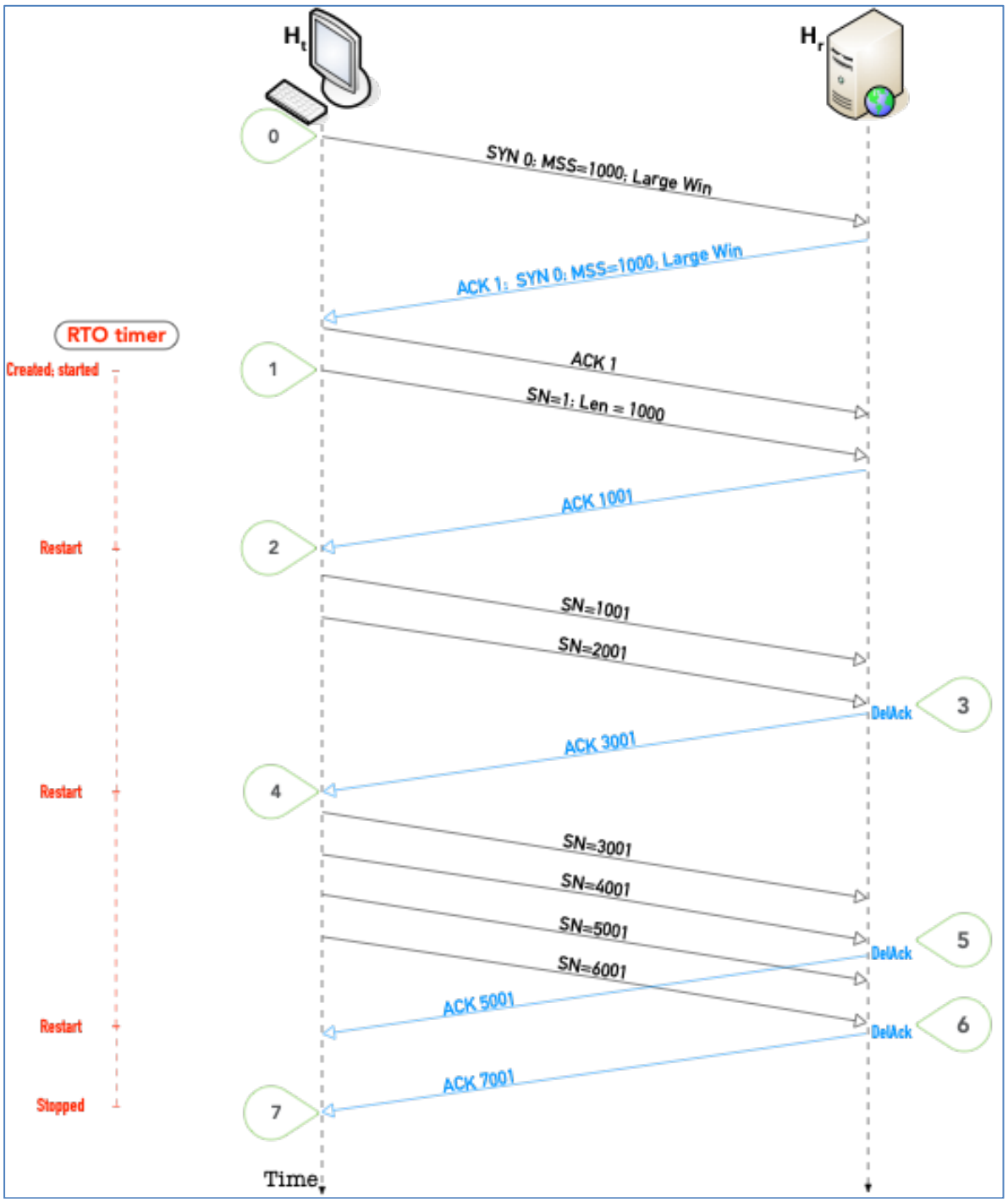

Figure 1. 3-way handshake and ensuing data transfers.

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

Here, we seek to illustrate the retransmission made by TCP when a packet is lost. The retransmission based on 3-DUP is what we illustrated in the past example. Now, we are interested in an example where a packet is lost, but, the number of packets transmitted after it is not sufficient to cause a 3-DUP. The TCP mechanism that ensures that the lost packet is retransmitted is based on the RTO timer.

In this example (See fig. 2), the 3-way handshake from the past example (Fig. 1) is reused and also, we assume that the transmission of the first segment (SN 1) and its ACK (1001) have both taken place. Then, the example of TCP transmission continues with the transmitter (H_t) sending two segments, the first having SN 1001 and the one following it having SN 2001. The latter of the two segments is lost amid its the path to its destination host H_r (See time index 4 in fig. 2). Like in the past example, segment with SN 1001 is successfully delivered at the receiver which starts the DelAck timer (≈ 200 ms) even though TCP is in the SS state and *Linux TCP never applies the DelAck timer when TCP is in the SS state*.

Observe at time index 6 that we assume that the DelAck timer fires before another segment carrying data arrives that carries an SN *contiguous* to 1001 + 1000 (The MSS). 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 MSS bytes in length, or 1000 bytes in this case, or a full segment). At time index 7, the ACK 2001 arrives at H_t; as TCP prescribes, that ACK that *advances* snd.una causes the RTO timer to restart (It is not stopped because further segments are waiting to be sent on the transmit buffer, at the present time). The TCP transmitter's state, after receiving the ACK is:

snd.una = 2001
snd.nxt = 3001

At time index 8, the sending TCP (H_t) sends a number of bytes that is twice as big as the advance of snd.una produced by the preceding received ACK (snd.una was advanced from 1001 to 2001, an advance worth 1000 bytes, or a full MSS). The equivalent to 2 x MSS bytes can be sent now. Variable snd.nxt points to 3001, at this time, *after* the transmission of SN 1001 (Len 1000) and SN 2001 (Len 1000) in the preceding Rtt; accordingly, segments with SN=3001 and SN=4001 are transmitted.

At time index 9 the two full segments are received by H_r. Observe that the first segment (SN 3001; Len 1000) is an *out-of-order* segment according to the state of the receiving TCP; that is so because the maximum level of progress in H_r's buildup of the in-order stream of data received from H_r is at 2000 at this moment. In other

words, the next expected *in-order* byte at H_r is 2001, *not at 3001*. The stipulated behavior of a TCP receiver that receives an *out-of-order* segment consists of sending back an immediate ack, or 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 TCP prescription just explained, the delivery of a segment carrying data from SN=4001 at H_r causes that host to send back a QuickAck (See time index 10) again carrying an ACK SN of 2001 (Its current value of **rcv.nxt**), as usual in TCP, representing the maximum progress made by the receiver in building the *stream* of in-order data received from the sender.

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 should do[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 data. In summary, no 3-DUP retransmission will be started at H_t.

For completeness, we should observe that the two last ACKs received 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 (Which is not the case); ultimately, the H_t-to-H_r side of the TCP connection becomes *idle*:

- No data segments are pending to be transmitted in the transmission buffer of H_t
- No ACKs are pending to be sent back from H_r
- The connection is *idle*.

Observe further that there are indeed pending ACKs at H_r: all of ACK SN=3001, SN=4001 and SN=5001, or an accumulated ACK worth all of them. Do you wonder how this idling of the connection can be broken down, now? Observe that there is still one segment that must be retransmitted after being lost. How is this connection work finished?

The key is in the received non-advancing ACKs. Non-advancing ACKs won't cause the RTO timer to restart. Check Fig. 2 where you'll readily identify only one restart of the RTO Timer (Time index 7); after that point in time, the RTO timer will forever free run down *until* the countdown elapses at time index 11. At that point

[2] **rcv.nxt** on the receive side is equivalent to Peterson and Davie's **NextByteExpected.** Receiver variable **rcv.nxt** points to the next in-order byte expected. Whenever the receiver sends back an ACK, the ACK SN encapsulated is a copy of the value of variable **rcv.nxt**.
[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.

in time the RTO timer fires, causing the connection to leave the idle state by having the segment pointed by snd.una (2001) retransmitted (See time index 12) and all of the ensuing segments up to, and not including snd.nxt.

Resent segment having SN 2001 arrives at the receiver at time index 13. That segment *fills the gap* created when that segment was dropped, the first time it was transmitted. H_r reacts by sending back a quickack *updating the max level of stream completion,* which at this time is at 5001 -note that segments with SN 3001 and 4001 were successfully received by H_r and were as well stored in H_r's receive buffer, hence the sudden jump in the ACK SN sent back from 3001 all the way through 5001.

On the sender side, the RTO timer is stopped after ACK 5001 is received. Transmitter variables snd.una and snd.nxt are equal, which means the transmission buffer is empty at this time, consequently no RTO timer is started. The connection is still alive though it is kept in the idle state until the moment the sending application decides to send more bytes through it.

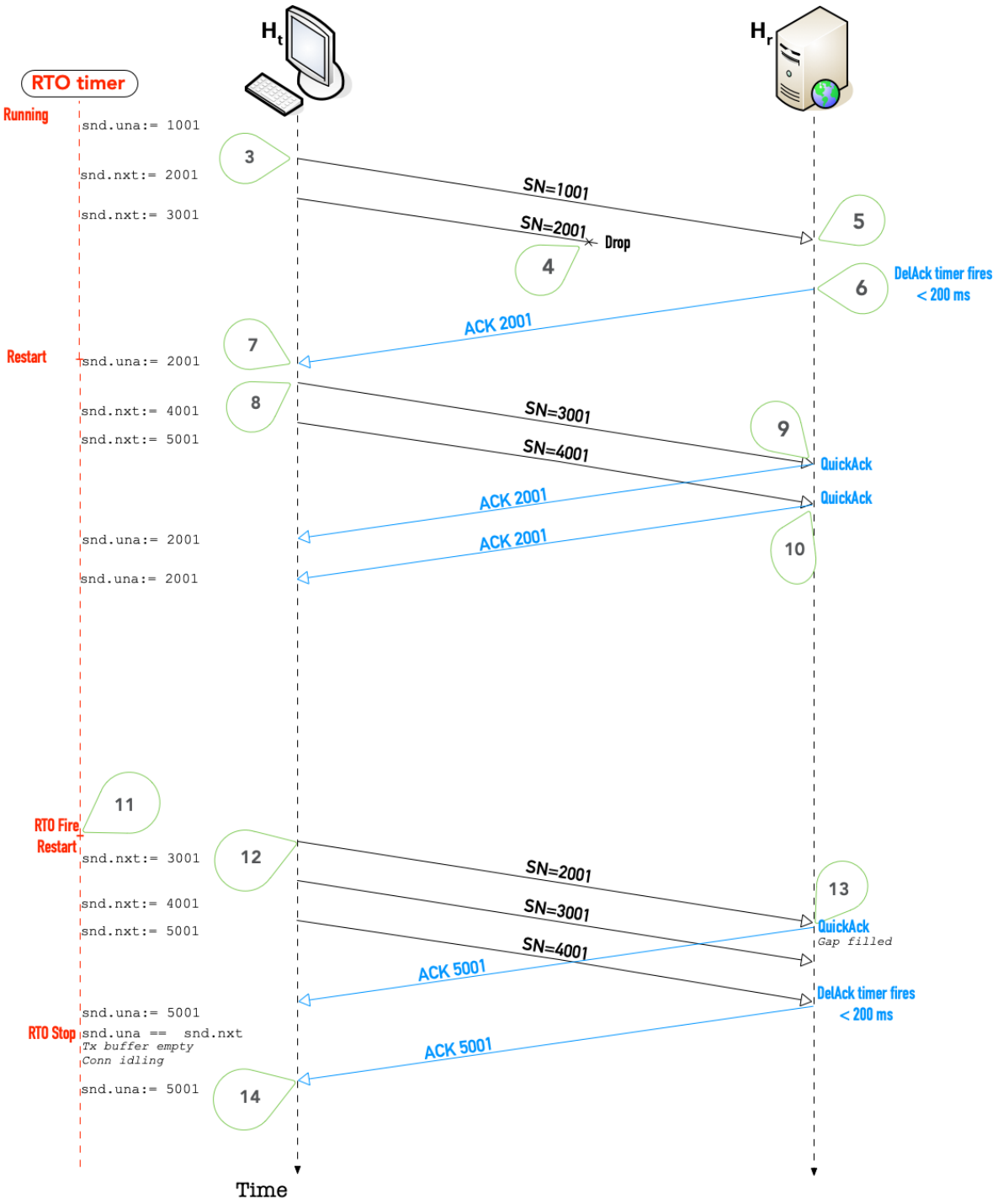

Figure 2. Continuation of the example in Fig. 1 assuming segment with SN=2001 gets dropped at time index 4.

