TCP Retransmission

The TCP retransmission means resending the packets over the network that have been either lost or damaged. Here, retransmission is a mechanism used by protocols such as [TCP](https://www.javatpoint.com/tcp) to provide reliable communication. Here, reliable communication means that the protocol guarantees packet's delivery even if the data packet has been lost or damaged.

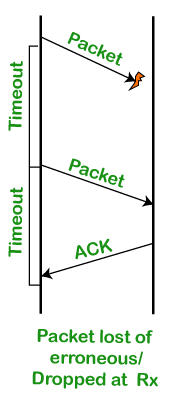
The networks are unreliable and do not guarantee the delay or the retransmission of the lost or damaged packets. The network which uses a combination of acknowledgment and retransmission of damaged or lost packets offers reliability.

Retransmission mechanism

Here, retransmission means the data packets have been lost, which leads to a lack of acknowledgment. This lack of acknowledgment triggers a timer to timeout, which leads to the retransmission of data packets. Here, the timer means that if no acknowledgment is received before the timer expires, the data packet is retransmitted.

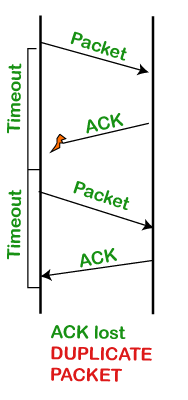
**Let's consider the following scenarios of retransmission.**

**Scenario 1: When the data packet is lost or erroneous.**



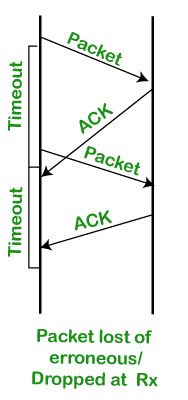
In this scenario, the packet is sent to the receiver, but no acknowledgment is received within that timeout period. When the timeout period expires, then the packet is resent again. When the packet is retransmitted, the acknowledgment is received. Once the acknowledgment is received, retransmission will not occur again.

**Scenario 2: When the packet is received but the acknowledgment is lost.**



In this scenario, the packet is received on the other side, but the acknowledgment is lost, i.e., the ACK is not received on the sender side. Once the timeout period expires, the packet is resent. There are two copies of the packets on the other side; though the packet is received correctly, the acknowledgment is not received, so the sender retransmits the packet. In this case, the retransmission could have been avoided, but due to the loss of the ACK, the packet is retransmitted.

**Scenario 3: When the early timeout occurs.**



In this scenario, the packet is sent, but due to the delay in acknowledgment or timeout has occurred before the actual timeout, the packet is retransmitted. In this case, the packet has been sent again unnecessarily due to the delay in acknowledgment or the timeout has been set earlier than the actual timeout.

In the above scenarios, the first scenario cannot be avoided, but the other two scenarios can be avoided. Let's see how we can avoid these situations.

**How long should the sender wait?**

**The sender sets the timeout period for an ACK. The timeout period can be of two types:**

* **Too short:** If the timeout period is too short, then the retransmissions will be wasted.
* **Too long:** If the timeout period is too long, then there will be an excessive delay when the packet is lost.

In order to overcome the above two situations, TCP sets the timeout as a function of the RTT (round trip time) where round trip time is the time required for the packet to travel from the source to the destination and then come back again.

**How can we obtain the RTT?**

The RTT can vary depending upon the network's characteristics, i.e., if the network is congested, it means that the RTT is very high. We can estimate the RTT by simply watching the ACKs.

**Let's see how we can measure the RTT.**

We will use the **original algorithm** to measure the RTT.

**Step 1:** First, we measure the **SampleRTT** for each segment or ACK pair. When the sender sends the packet, then we know the timer at which the packet is sent, and also, we know the timer at which acknowledgment is received. Calculate the time between these two, and that becomes the **SampleRTT**.

**Step 2:** We will not take only one sample. We will keep on taking different samples and calculate the weighted average of these samples, and this becomes the EstRTT (Estimated RTT).

**where, α+ β = 1**

**α lies between 0.8 and 0.9**

**β lies between 0.1 and 0.2**

**Step 3:** The timeout is set based on EstRTT.

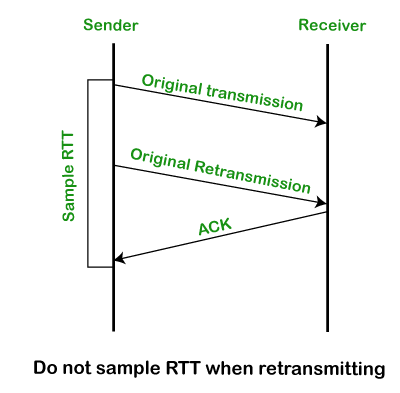
**timeout = 2 \* EstRTT.**

The timeout is set to be twice the estimated RTT. This is how the actual timeout factor is calculated.

**A Flaw in this approach**

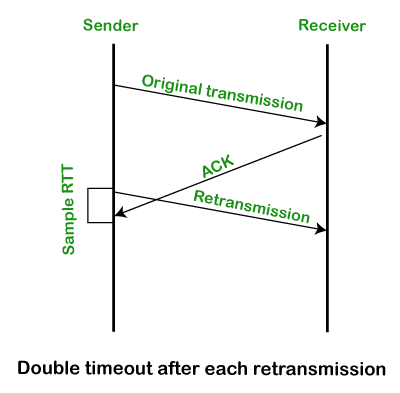
**There is a flaw in the original algorithm. Let's consider two scenarios.**

**Scenario 1.**



The above diagram shows that the sender sends the data, which is said to be an original transmission. Within the timeout period, no acknowledgment is received. So, the sender retransmits the data. After retransmitting the data, the acknowledgment is received. Let's assume that acknowledgment is received for the original transmission, not for the retransmission. Since we get the acknowledgment of the original transmission, so **SampleRTT** is calculated between the time of the original transmission and the time at which the acknowledgment is received. But actually, the **SampleRTT** should have been between the time of the retransmission and time of the acknowledgment.

**Scenario 2.**



The above diagram shows that the sender sends the original data packet for which we get the acknowledgment also. But the acknowledgment is received after retransmitting the data. If we assume that acknowledgment belongs to the retransmission, then **SampleRTT** is calculated between the time of the retransmission and the time of the acknowledgment.

In the above both the scenarios, there is an ambiguity of not knowing whether the acknowledgment is for the original transmission or for the retransmission.

**Conclusion of the above algorithm.**

* Here, ACK does not mean to acknowledge a transmission, but actually, it acknowledges a receipt of the data.
* If we consider the first scenario, the retransmission is done for the lost packet. In this case, we are assuming that ACK belongs to the original transmission due to which the SampleRTT is coming out to be very large.
* If we consider the second scenario, two same packets are sent so duplicity occurs in this case. In this case, we are assuming that ACK belongs to the retransmission due to which the SampleRTT is coming to be very small.

To overcome the above problems, a simple solution is given by the Karn/Partridge algorithm. This algorithm gave a simple solution that collects the samples sent at one time and does not consider the samples at the retransmission time for calculating the estimated RTT.

Karn/Partridge Algorithm

In the above two scenarios, retransmission occurs, and we have considered the Sample RTT. But this algorithm does not consider the Sample RTT when retransmitting. Since the retransmission has occurred, which means that something happens in this round-trip time or some congestion may occur in a network. To overcome this problem, this algorithm doubles the timeout after each retransmission. This algorithm is implemented in the TCP network.

**Limitation**

**It does not consider the variance in RTT.**

* **If the variance is small, the EstimatedRTT comes out to be accurate. If the variance is large, the EstimatedRTT is not accurate.**

To overcome the above limitation, the Jacobson/Karels algorithm was developed that introduces the variance factor in RTT.

**Jacobson/Karels Algorithm**

This algorithm was developed to overcome the limitation of the **Karn/Partridge** algorithm. It computes the difference between the SampleRTT and EstimatedRTT, and boost the RTT based on the difference.

Calculations for average RTT

* First, we calculate the difference factor.

**Diff = SampleRTT - EstimatedRTT**

* Now, we calculate the EstimatedRTT, which will be calculated in the same way as we did above.

**EstRTT = EstRTT + (δ\*Diff)**

* Now, we calculate the average of the difference factor.

**Dev = Dev + δ ( |Diff| - Dev)**

Here, Dev is a deviation factor, and δ is a factor between 0 and 1. The **Dev** is an estimate of the variance from the **EstRTT**.

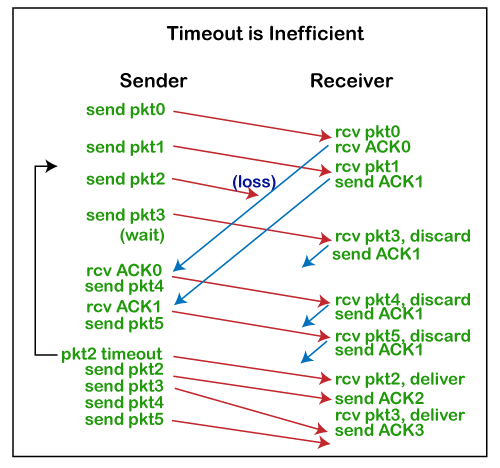
* We will consider the variance while calculating the timeout value.

Timeout = µ \* EstRTT + ɸ \* Dev

Where, **µ =1 and ɸ =4**

Fast Retransmission

The timeout-based strategy for retransmission is inefficient. TCP is a sliding-window kind of protocol, so whenever the retransmission occurs, it starts sending it from the lost packet onward.



Suppose I transmit the packets 0, 1, 2, and 3. Since packet 0 and packet 1 are received on the other side, packet 2 is lost in a network. I have received the acknowledgment of packet 0 and packet 1, so I send two more packets, i.e., packet 4 and packet 5. When packets 3, 4, and 5 are sent, then I will get the acknowledgment of packet 1 as TCP acknowledgments are cumulative, so it acknowledges up to the packet that it has received in order. I have not received the acknowledgment of packet 2, 3,4, and 5 within the timeout period, so I retransmit the packets 2, 3, 4, and 5. Since packet 2 is lost, but other packets, i.e., 3, 4,5 are received on the other side, they are still retransmitted because of this timeout mechanism.

**How can this timeout inefficiency be removed?**

The better solution under a sliding window:

Suppose n packet has been lost, but still, the packets n+1, n+2, and so on have been received. The receiver is continuously receiving the packets and sending the ACK packets saying that the receiver is still awaiting the nth packet. The receiver is sending repeated or duplicate acknowledgments. In the above case, ACK of packet 1 is sent three-time as packet 2 has been lost. This duplicate ACK packet is an indication that the nth packet is missing, but the later packets are received.

The above situation can be solved in the following ways:

* The sender can take the "duplicate ACKs" as an early hint that the nth packet has been lost so that the sender can do the retransmission as early as possible, i.e., the sender should not wait until the timeout occurs.
* The sender can implement a fast transmission strategy in TCP. In a fast transmission strategy, the sender should consider the triple duplicate ACKs as a trigger and retransmit it.

TCP uses three duplicate ACKs as a trigger and then performs retransmission. In the above case, when three ACKs of packet 1 are received, then the sender should send the lost packet, i.e., packet 2, without waiting for the timeout period to occur.