

## TCP Retransmission Time Out (RTO) and Round-Trip Time (RTT) in MQX RTCS

No.	Time	Source	Destination	Protocol	Length	Info
70	16.763424	192.168.1.202	192.168.1.17	TELNET	814	Telnet Data ...
71	16.792363	192.168.1.202	192.168.1.17	TELNET	754	Telnet Data ...
72	16.792469	192.168.1.202	192.168.1.17	TCP	814	[TCP Out-Of-Order] 23 → 63192 [PSH, ACK] Seq=29270 Ack=16 Win=1445 Len=760
73	16.792481	192.168.1.202	192.168.1.17	TCP	754	[TCP Retransmission] 23 → 63192 [PSH, ACK] Seq=30030 Ack=16 Win=1445 Len=700
74	16.792495	192.168.1.17	192.168.1.202	TCP	54	63192 → 23 [ACK] Seq=16 Ack=30730 Win=64240 Len=0
75	16.792508	192.168.1.17	192.168.1.202	TCP	54	[TCP Dup ACK 74#1] 63192 → 23 [ACK] Seq=16 Ack=30730 Win=64240 Len=0
76	16.792513	192.168.1.17	192.168.1.202	TCP	54	[TCP Dup ACK 74#2] 63192 → 23 [ACK] Seq=16 Ack=30730 Win=64240 Len=0
77	16.794403	192.168.1.202	192.168.1.17	TELNET	854	Telnet Data ...
78	16.822340	192.168.1.202	192.168.1.17	TELNET	714	Telnet Data ...
79	16.822440	192.168.1.17	192.168.1.202	TCP	54	63192 → 23 [ACK] Seq=16 Ack=32190 Win=64240 Len=0
80	16.823419	192.168.1.202	192.168.1.17	TELNET	894	Telnet Data ...
81	16.847449	192.168.1.202	192.168.1.17	TCP	894	[TCP Retransmission] 23 → 63192 [PSH, ACK] Seq=32190 Ack=16 Win=1445 Len=840
82	16.847511	192.168.1.17	192.168.1.202	TCP	54	63192 → 23 [ACK] Seq=16 Ack=33030 Win=63400 Len=0
83	16.852668	192.168.1.202	192.168.1.17	TELNET	1514	Telnet Data ...
84	16.897340	192.168.1.202	192.168.1.17	TCP	1514	[TCP Retransmission] 23 → 63192 [PSH, ACK] Seq=33030 Ack=16 Win=1445 Len=1460
85	16.897419	192.168.1.17	192.168.1.202	TCP	54	63192 → 23 [ACK] Seq=16 Ack=34490 Win=64240 Len=0

If you have noticed many TCP *Retransmission*, *Out Of Order* or *Duplicate Acknowledge* packets while running a TCP/IP application based on MQX RTCS then you may be interested in this article. After some investigation it comes to be that the default value of global variable `_TCP_rto_min` may cause this congestion depending on the application. Finally, this problem was solved by setting a new value to this variable after calling function `RTCS_create()` as shown in the snippet below. This article explains what is behind this behavior and how `_TCP_rto_min` affects the performance in an application using RTCS.

```

/* Initialize RTCS */
error = RTCS_create();
if (error != RTCS_OK)
{
    fputs("Fatal Error: RTCS initialization failed.", stderr);
    _task_block();
}
_TCP_rto_min = 250;

```

Network congestion occurs when a network node is carrying more data than it can handle. This can cause queuing delay, packet loss or the blocking of new connections. Network protocols that use aggressive retransmissions to compensate for packet loss due to congestion can increase congestion, even if the initial load has been reduced to a level that would not normally have induced network congestion.

Connection-oriented protocols such as TCP, watch for packet errors, losses, or delays to adjust the transmit speed; this is done using the **Round-Trip Time (RTT)** also called round-trip delay. RTT is the time required for a signal pulse to travel from a specific source to a specific destination and back again. In this context it is the time it takes for an outgoing TCP client packet to be acknowledged by the server.


The round trip time is an important factor when determining application performance because on each transmission the packets have to travel back and forth adding some delay. Initial RTT is the round trip time that is determined by looking at the TCP Three Way Handshake. Next you can see an example captured with WireShark sniffer.

No.	Time	Source	Destination	Protocol	Length	Info
5	15.984545	192.168.1.17	192.168.1.202	TCP	66	63192 → 23 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
6	15.985712	192.168.1.202	192.168.1.17	TCP	60	23 → 63192 [SYN, ACK] Seq=0 Ack=1 Win=1460 Len=0 MSS=1460
7	15.985847	192.168.1.17	192.168.1.202	TCP	54	63192 → 23 [ACK] Seq=1 Ack=1 Win=64240 Len=0

As you can see in the capture above device 'A' (with IP 192.168.1.17) sends a TCP SYN packet to device 'B' (with IP 192.168.1.202), this is where RTT timer begins. Then device 'B' sends a TCP SYN-ACK back to 'A', this is where RTT timer ends. Finally device 'A' sends a TCP ACK packet and at this point connection is established.

Below you can see the detailed information about the last ACK packet shown in the above figure; here you can find Initial RTT. In WireShark captures you can see the packet details by double clicking on the packet you want to analyze.

```
▶ Frame 7: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
▶ Ethernet II, Src: Dell_5a:68:a1 (d4:be:d9:5a:68:a1), Dst: 00:a7:c5:f1:11:90 (00:a7:c5:f1:11:90)
▶ Internet Protocol Version 4, Src: 192.168.1.17, Dst: 192.168.1.202
▲ Transmission Control Protocol, Src Port: 63192 (63192), Dst Port: 23 (23), Seq: 1, Ack: 1, Len: 0
  Source Port: 63192
  Destination Port: 23
  [Stream index: 0]
  [TCP Segment Len: 0]
  Sequence number: 1 (relative sequence number)
  Acknowledgment number: 1 (relative ack number)
  Header Length: 20 bytes
▶ Flags: 0x010 (ACK)
  Window size value: 64240
  [Calculated window size: 64240]
  [Window size scaling factor: -2 (no window scaling used)]
▶ Checksum: 0x8446 [validation disabled]
  Urgent pointer: 0
▲ [SEQ/ACK analysis]
  [This is an ACK to the segment in frame: 6]
  [The RTT to ACK the segment was: 0.000135000 seconds]
  [iRTT: 0.001302000 seconds]
```



TCP protocol provides a mechanism for ensuring that packets are received, it uses a timer called **Retransmission Timeout (RTO)** that has an initial value of three seconds. This value is a first, best guess of the Round-Trip Time for a stream socket packet. RTCS attempts to resend the packet, if it does not receive an acknowledgment in this time. After each retransmission the value of the RTO is doubled up to three times. This means that if the sender does not receive the acknowledgement after three seconds it will retransmit the packet, this time the sender will wait for six seconds to get the acknowledgement, and if it still does not receive the acknowledgement, it will retransmit the packet for a third time and will wait for 12 seconds. If sender does not receive acknowledgement at this time it will give up. In RTCS the RTO is defined in `tcp_prv.h`.

```
#define TCP_INITIAL_RTO_DEFAULT 3000 /* Retransmission timeout when opening
                                     a connection, default 3 seconds
                                     as per RFC1122 */
```

However this value can be also modified in the application using function `setsockopt()` with `OPT_RETRANSMISSION_TIMEOUT` option instead of rebuilding RTCS library.

But this is not the only logic in TCP, in many cases, mainly when devices are close to each other, waiting 3 second for a packet retransmission may be too long and the application performance can be impacted. In these cases retransmission logic should be quicker; this is where RTT starts impacting RTO.

When a TCP connection is established, there is one RTT value, and the RTO will be adjusted based on the Smoothed RTT (SRTT) calculation. This calculation makes accurate estimates of Round-Trip Time, once it is calculated it's used to modify RTO value by determining how long the host should wait before retransmitting the segment. E.g. let's say your connection is fast enough to reduce the RTO to 500ms, in this case each packet loss won't need to wait 3 seconds (default value) to retransmit and the application's performance is impacted in a positive way. This strategy is known as Karn's Algorithm and is considered to be extremely effective, especially in networks with high packet loss and latency. In RTCS these calculations are performed in function `TCP_Process_RTT_measure()`.

TCP provides a lock to avoid that RTO keeps decreasing more than the OS timing specs allows. The lowest RTO will depend on the operating system or TCP implementation; in Windows it is 300ms, and in Linux it is 200ms. In RTCS the lowest value is 3 times the tick length (5ms), so `TCP_RTO_MIN` value is 15ms. This is defined in `tcp_prv.h`.

```
#define TCP_RTO_MIN (TICK_LENGTH*3) /* Wait at least two ticks before
retransmitting a packet; imposed
by granularity of timer hardware
(if we waited only one tick, the
tick might expire right away...) */
```

Once `RTCS_create()` is called in your application the global variable `_TCP_rto_min` gets `TCP_RTO_MIN` as its initial value. This assignment is done in `tcp.c`. However, this value can be modified in the user application.

```
/*
** Global used for TCP_RTO_MIN because we may want to let our
** users adjust this.
*/
_TCP_rto_min = TCP_RTO_MIN;
```

To modify `_TCP_rto_min` it is only necessary to overwrite its value **AFTER** calling `RTCS_create()`. The new value may depend on your application's requirements, in this case a value of 250ms was used.

```
/* Initialize RTCS */
error = RTCS_create();
if (error != RTCS_OK)
{
    fputs("Fatal Error: RTCS initialization failed.", stderr);
    _task_block();
}
_TCP_rto_min = 250;
```

In the same way there is a min RTO there is also a max RTO. Its default value is 4 minutes; this is 2 times the Maximum Segment Lifetime which is also defined in `tcp_prv.h`. This value can also be modified using function `setsockopt()` with option `OPT_MAXRTO`

```
#define TCP_MSL          120000L /* Maximum Segment Lifetime; the
                                longest time that a packet can
                                travel in the Internet (2 min) */

#define TCP_WAITTIMEOUT (2 * TCP_MSL) /* timeout for TIME_WAIT state, defined
                                        as 2 * MSL (4 min) */
```

Since each connection is distinct, we must maintain SRTT calculations about each connection separately, so one connection does not impact the other. TCP uses a special data structure for this purpose, called a Transmission Control Block (TCB).

#### Summary:

The Smoothed RTT retransmission logic exists to ensure that users do not experience long latency when there is congestion in a low latency connection. If it wasn't for the application of these accurate predictions then you would experience even more latency on your network.