INDUSTRIAL CONTROL COMMUNICATIONS, INC.

1600 Aspen Commons, Suite 210
Middleton, WI, USA 53562
Tel: (608) 831-1255
Fax: (608) 831-2045
Email: sales@iccdesigns.com

# A Discussion of Modbus/TCP Server-Side Timeout Processing

While the concept of timeout processing in Modbus/TCP seems obvious at first glance, there are quite a few nuances that can affect timeout behavior in unexpected manners. First, we need to examine how timeout timers are typically used in industrial communications. In general, a timeout timer monitors the health of a connection between two endpoints (such as a master device and a slave device). From a human logic perspective, we could describe in plain English that if a connection to a master "goes away", then the timeout should fire. This is all good and fine, but note that this timeout behavioral explanation introduces the new term of "connection". Therefore, we now have to define "what is a connection?" in order to be able to accurately describe the behavior of a timeout timer.

Some industrial communication protocols (such as EtherNet/IP) are referred to as "connection-oriented", which is to say that communication can only take place within the auspices of a "connection". Regardless of the connection-oriented protocol being discussed, these specifications all define a connection as having three distinct phases in its typical lifecycle:

1. A connection with a target (server) is established by an originator (client)
2. The connection is used to transfer data between the originator and the target
3. The connection is torn down by the originator

In connection-oriented protocols, all timeout processing is performed only during the "data transfer" stage in the lifecycle mentioned above. This makes sense, as a timeout timer must know when to start timing (when an originator has created a new connection), and when to stop timing (when the originator has torn the connection down, which is, in essence, its indication that it no longer wishes to exchange data with the target, and is therefore gracefully relinquishing its claim to any of the target's resources). As an example, in the EtherNet/IP protocol, a class 1 (I/O) connection is established by the client with a "forward open" command, which in addition to creating a connection, also includes timeout time values that the originator and target are required to use to detect timeout events during the data transfer stage. “Connected data” then flows between the originator and the target over a UDP/IP transport layer. When the originator no longer wishes to interact with the target, it then issues a "forward close" command, which tears down the connection and causes both the originator and the target to stop/delete their timeout timers that were monitoring the frequency of data transfer.

While connection-oriented protocols typically incur more overhead and are much more difficult to understand and implement due to the strict rules defining connection management (and indirectly, timeouts), they do provide the added benefit of clearly defining the lifespan of connections, allow for the recycling of connection resources in both targets and originators, and clearly define the behavior and implementation of timeout processing.

Now that we have a basic understanding of the concept of a "connection", and the fact that timeout timers monitor the frequency of data transfer solely within the lifespan of a connection, we can consider how this is relevant to the Modbus/TCP protocol. First, we must understand that Modbus/TCP principally operates at two distinct layers of the ISO/OSI network model: the application layer (which implements the concept of "holding registers" and “function codes”, etc.) and the transport layer (which implements the TCP socket portion of Modbus/TCP). Similar to traditional serial-line Modbus RTU, the application layer of the protocol is connectionless: there is no concept in Modbus/TCP from an application perspective of a connection being established or torn down. All available resources of a Modbus/TCP server are available at-will to any client (such as a PLC) at any time, without any formal "Modbus connection" first being established. On the other hand, the transport layer of Modbus/TCP uses TCP (transmission control protocol), which is by definition connection-oriented from an Ethernet perspective (versus UDP, which is not). Therefore, Modbus/TCP can be referred to either as either a connectionless or connection-oriented protocol, depending on the perspective of the observer and the portion of the protocol stack that is currently being referenced.

1600 Aspen Commons, Suite 210
Middleton, WI, USA 53562
Tel: (608) 831-1255
Fax: (608) 831-2045
Email: sales@iccdesigns.com

Next, we need to examine what the Modbus/TCP specification decrees in regard to timeout processing. In this specification, the concept of timeout processing is not just avoided, but it is actually intentionally not defined. The specification states the following:

*…There is deliberately NO specification of required response time for a transaction over MODBUS/TCP. This is because MODBUS/TCP is expected to be used in the widest possible variety of communication situations, from I/O scanners expecting sub-millisecond timing to long distance radio links with delays of several seconds…*

Therefore, all Modbus/TCP timeout-related discussions from either a client or server perspective are by definition vendor-specific implementations/extensions of the specification. That being said, however, the vast majority of vendors that implement server-side timeout processing do so in a very consistent manner (which will be detailed further below) for obvious reasons, based on our previous discussion pertaining to connection-oriented timeout timer lifespans.

Because our previous definition of a timeout timer is a mechanism that monitors the frequency of data transfer within the framework of a connection, and that the only concept of a "connection" in Modbus/TCP is that which is provided by the TCP layer itself, it should now be clear that a typical Modbus/TCP server-side timeout timer will only operate during the lifespan of a client's TCP connection to it. The obvious implication of this statement is that if a client does not have a current TCP connection (socket) established with a server, then timeout processing is not being performed on the server. If a client wishes to perpetually communicate with a server (and therefore allow the server to provide the benefit of any vendor-implemented server-side timeout processing), then it should not close its socket connection to the server. In fact, the *Modbus Messaging Implementation Guide* states the following "implementation rule" for clients in section 4.2 (TCP connection management):

*…It is recommended to keep the TCP connection opened with a remote device and not to open and close it for each MODBUS/TCP transaction…*

The manner in which such a preferred technique can be accomplished will vary depending on the client in use, but at a minimum such a recommendation carries along with it the necessity that a client desiring to perform such persistent-socket behavior cannot be asked to establish more connections than are available in its firmware implementation. For example, if a certain client allows for a maximum of 6 simultaneous connections, then it cannot be expected to communicate with more than 6 combined endpoints on any combination of remote devices without having to "recycle" its connections. Such recycling is usually manifested by the client opening a connection, performing its prescribed reads and/or writes, and then immediately closing the connection again. By closing the connection, however, the client has in essence made an explicit declaration to the server that "I no longer wish to exchange data with you, and am therefore gracefully severing my connection with you so that you may reuse your internal resources to accept connections originated by other clients." Such an explicit termination of the TCP connection therefore also terminates the timeout timer that was in use to monitor the frequency of data communications over that connection. At this point, the TCP server (not being stateful from a connection-to-connection perspective) has no preconceived notions that this particular client will necessarily ever wish to communicate with it again: it only knows that the client has informed it that it now wishes to have nothing to do with this server, and that it should therefore go about its independent business of listening for new connections. If any such new connections are established at a future time, they are once again treated by the server as entirely independently-performing data connections, regardless of whether or not they are established by a client which had previous communications with this server.

Behaviorally speaking, if a client that establishes only momentary connections with a server is used, the manner in which timeout triggering occurs can externally appear "unpredictable". If the Ethernet link is compromised (for example, if an Ethernet cable is severed or the client abruptly loses power), then the principle question is whether or not the fault occurred while a connection (socket) was active between the client and the server. If a socket was open (data transfer phase), then the server's timeout timer will be running, and it will eventually timeout and trigger the user-prescribed "timeout action" (stop the drive, etc.) If, however, the fault occurs in between the connections that the client creates with the server, then no timers are running on the server and therefore the server will not trigger any timeout behavior. The likelihood of a timeout being detected by the server therefore

1600 Aspen Commons, Suite 210
Middleton, WI, USA 53562
Tel: (608) 831-1255
Fax: (608) 831-2045
Email: sales@iccdesigns.com

becomes entirely dictated by the duty cycle (% connected time vs. % non-connected time) established by the client.

If the client being used in a certain application does not allow for the persistent socket connection behavior recommended by the Modbus specification, and if it is desired to initiate server-side timeout processing from a connection-to-connection standpoint (not just within a connection), then this is certainly a possibility. An optional "supervisory timer" thread can be instantiated in the Modbus/TCP driver which monitors the frequency of Modbus communications with the server device overall (as opposed to each socket thread monitoring the frequency of communications within the framework of a single socket connection). Such a supervisory timer would behave according to the following rules:

1. The supervisory timer timeout time will be the same value used within a socket connection, and defined by the user on the Config tab's "timeout" field.
2. Initially after boot-up (before initial contact by any client), the supervisory timer will be disabled, and no timeouts will be triggered.
3. Once a client establishes a connection (socket) with the server and begins to exchange Modbus packets, the supervisory timer will begin timing the frequency of all Modbus traffic (regardless of the originator).
4. If an interval of time greater than the "timeout" value expires (regardless of whether the timeout occurred within an open socket connection or between socket connections), a timeout event will be triggered.
5. Once a timeout event is triggered, the supervisory timer will once again be disabled until a client once again connects to the server.
6. Note that because the supervisory timer does not operate within the framework of any connection (which, by definition, always has a "connection termination" phase), there is no way to disable the timer once it is started. This means that once communication is initiated with the server, it is inevitable that a timeout event must occur at some point, even if the communication stoppage from the client is a planned event.

Note that item #6 above requires that the application be tolerant of a timeout being triggered whenever a controlled stoppage of the client is initiated (during routine plant maintenance, for example). This is due to the fact that the supervisory timer is expected to continue timing even after the client has gracefully closed its connection (TCP socket) to the server. Because there is no mechanism in TCP which can signal different "socket close events", there is therefore no way for the client to signal to the server the different interpretations of "I am closing my connection with you, but expect me to reopen another connection soon" versus "I am closing my connection with you, but no longer wish to communicate with you at all in the future, so please stop timing the frequency of my connections".