

Implementing Modbus/TCP Avoiding Multi-Vendor Pitfalls

By: Lynn August Linse
<http://www.iatips.com>

ABSTRACT

Today, Modbus/TCP is the richest option for Ethernet users desiring the widest supply of equipment from the most vendors. Even if Modbus is not your native device protocol, you can likely add a simple Modbus/TCP server (ie: answer remote queries) with only a few hundred lines of code. This talk covers the basic implementation issues of adding Modbus/TCP to your Ethernet product, including a range of lessons learned from the field. With over 5 years of multi-vendor Modbus/TCP experience, the speaker is well qualified to point out pitfalls to avoid and questionable behavior to expect from other implementers.

BACKGROUND – UNDERSTANDING MY CONTEXT

I craft firmware for Ethernet to Serial Converters – including one of the first embedded Modbus/TCP (Ethernet) to Modbus/RTU (Serial) bridge products on the market. I initiated that product back in 1996 and wrote the firmware. Its update and support has given me considerable exposure to multi-vendor issues.

So what is a “Bridge Product”? It moves Modbus commands (or application data units) between Modbus/TCP and Modbus/RTU (or Modbus/ASCII). It allows a Modbus/RTU master to poll Modbus/TCP slaves, or Modbus/TCP masters to poll Modbus/RTU slaves. It doesn't need to understand Modbus commands to do this – only what I call the protocol framing or data link issues of each media.

Modbus/TCP requires a 6-byte header before the Modbus command, but relies upon the underlying (and standard) Ethernet with TCP/IP, while Modbus/RTU as no header and requires a 16-bit CRC be appended. Figure 1 below shows the relationship between Modbus/TCP and Modbus/RTU.

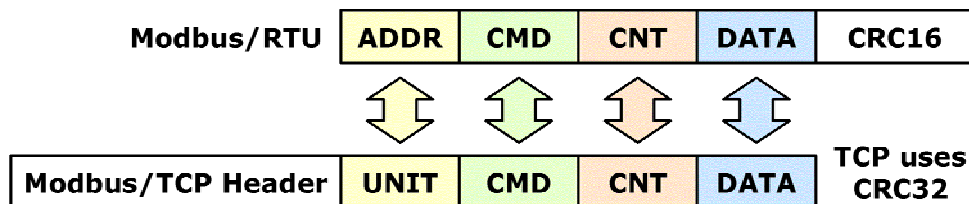


FIGURE 1 – MODBUS/TCP COMPARED TO MODBUS/RTU.

This meeting of two-worlds - one fast and one slow - exposes problems that traditional Modbus/RTU-only devices or Modbus/TCP-only devices don't see. Most importantly, it causes the unspoken assumptions programmers built into their products to clash.

This paper summarizes some of these clashes in hopes it helps others avoid them – plus will make my life easier as their products will clash less with mine! A series of check-marked items are included you should review and answer.

- o o Check off the issues! (here's a sample)

While this document is geared explicitly towards Modbus, you'll find many of the issues apply equally well to other protocols when faced with Ethernet or TCP/IP.

MODBUS IN A NUT-SHELL

Thousands of serial industrial protocols have been introduced over the years. Some are undocumented proprietary secrets. Many are passively open (ie: quietly published) but with limited interest only to users of a single-vendor's equipment. Some are even published and promoted with the high expectations of becoming "the One Open Standard".

However, decades old Modbus is probably the only protocol that a room full of industrial users could agree can rightly be labeled a "common, standard". Why?

- Modbus is simple and forces few design restrictions onto implementers.
- Modbus can be adequately described in a few pages of a product manual
- Modbus test routines can be written and working in a few hours.

Possibly the most important incentive for Modbus's growth in the 3rd party product market is that Modbus allows your salesman to go after new customers and have an easy answer to the question "Where can I find a software driver for your products?". It's likely they already have Modbus drivers and expertise.

Modbus treats your product data as just an array of 16-bit registers – with an optional 2nd array of 1-bit Boolean status or coil bits. Modbus can be satisfactorily implemented with just 3 commands – 6 if you want the Boolean bit array also. Modbus allows you to read multiple values, write a single value, or write multiple values. You don't need a dozen commands for a dozen data types. You don't need to negotiate link parameters or mimic complex device status responses. You don't need to use different read or write commands for different products from the same vendor.

Is multi-vendor Modbus trouble-free or instantly realizable? No! But one of the remarkable assets of Modbus is that nearly all "problems" one meets with multi-vendor systems can be worked-around by users. Unlike more complex protocols which either work between 2 vendors or they don't, incompatible in Modbus can usually be made compatible without vendor product changes.

The full specification is available online. There is also an Internet-based "Modbus Users Group" forming to help promote Modbus/TCP to RFC level – see details at www.modbus.org.

NEW CONCEPTS IN MODBUS/TCP

One of the main problems with Modbus/TCP is that since Modbus/RTU has been around for so long many implementers assume they already know everything about Modbus/TCP and overlook the subtle difference in behavior. This section lists some of the new things to watch out for.

EXCEPTION RESPONSES

Besides the traditional Modbus serial exception responses, the Modbus/TCP specification includes 2 new *important* exception responses that are often overlooked.

10 (Hex: 0x0A) Gateway Path Unavailable

A Bridge issues this exception when a requested Unit Id (or Slave Id) is not configured in the bridge table used to map device ids between media. The bridge was unable to forward the command to the requested destination.

If none of the traditional Modbus exception codes are appropriate, you should use exception 0x0A for any "**hard error**" that won't fix itself without user intervention. So use it when retries won't help.

- ○ A Modbus/TCP client seeing exception 0x0A should issue a dialog box or other active warning to the user.
- ○ A Modbus/TCP server should make use of exception 0x0A when appropriate.

11 (Hex: 0x0B) Target Device Failed to Respond

A Bridge issues this exception when the target device failed to respond. Keep in mind that the bridge (either Modbus/TCP to ModbusPlus or Modbus/TCP to Modbus/RTU) is acting as an intermediary. Therefore the traditional Modbus serial behavior of no response is not expected and will cause numerous problems in the Modbus/TCP world.

If none of the traditional Modbus exception codes are appropriate, you can use exception 0x0B for any “**soft error**” that may resolve itself naturally and more retries may help.

- ○ A Modbus/TCP client seeing exceptions 0x0B should **NOT** issue a dialog box or active warning, but continue to retry and log events as appropriate.
- ○ A Modbus/TCP server should make use of exception 0x0B when appropriate.

Modbus Serial Masters don't expect exception 0x0A or 0x0B

This is an issue for people supporting both Modbus/TCP and Modbus/RTU. This is especially critical when the Modbus/RTU master is a DCS or SCADA gateway!

You should by default never issue the exceptions 0x0A or 0x0B over the serial link – this includes actively filtering them out and discarding them from a Modbus/TCP channel being bridged to Modbus/RTU. The simple reason is few (no?) serial masters expect or properly handle these responses.

- ○ A device bridging Modbus/TCP messages to Modbus serial masters should by default filter out (discard) responses with exceptions 0x0A and 0x0B.

Best case, the serial master may mistake the unknown exception as “a response”, and thus fail to take the desired recovery or fail-over behavior. These masters are expecting “no response” as the signal to fail-over.

Worst case, the serial master may crash or mark the serial channel as 100% bad and basically go off-line or stop polling it for ANY slaves.

NO-RESPONSE IS BAD – RETRIES MAKE THINGS WORSE

This is the biggest legacy assumption Modbus/TCP products inappropriately inherit from Modbus serial products.

Modbus serial uses “no response” to mean try again

When a Modbus serial master receives no response, it merely tries again. Since it is the only master on the line, it will never consider that other masters are loading the serial line and slowing the slave response. Also, by design serial drivers flush out old requests at the start of each new communication cycle that reduces (but doesn't eliminate) the risk that an old response is mistakenly linked to a new request.

Modbus/TCP is different

TCP/IP is reliable transport. As long as the TCP socket is active, Modbus/TCP requests can NEVER be lost or corrupted. Every request WILL be received by the intended server and it can respond with either a good or exception response.

Although the Modbus/TCP specification requires half-duplex behavior, many OPC servers “pipeline” requests on a single socket. This means they use the TCP/IP flow-control and buffering to queue up many Modbus/TCP requests. For example, if they have 10 blocks of Modbus data to poll, many issue all 10 Modbus/TCP requests at once. Why? They do this because it both simplifies their design and offers a significant improvement in performance.

- Modbus/TCP server must tolerate a queue of requests in the TCP buffer.

However, this either complicates the Modbus/TCP server side or risks “system meltdown” (explained in the example below). This is magnified when the Modbus responses require significant time to return. A server which sees 160 bytes in its TCP buffer cannot be sure if these consist of 10 unique requests - or 5 sets of 2 requests repeated (ie: retried) - or even 1 request retried 10 times.

The simple solution is for the Modbus/TCP server to just ignore the size of the queue and pull requests off one by one. This risks an ever-increasing queue if the Modbus/TCP client doesn't realize it's over-running the Modbus/TCP server.

The complex solution is for the Modbus/TCP server to rapidly extract all data from the TCP buffer and create an internal queue of waiting requests. These must be processed for duplicates and aged. While ideal, this greatly increases the programming complexity and CPU power needed, plus adds extra response latency even when the bridge is lightly loaded.

Example of Modbus Bridge "System Meltdown"

Suppose the Modbus/RTU channel can handle about 10 poll cycles per second.

Suppose a remote Modbus/TCP client (say an OPC server) issues 8 requests per second and has no ability to back-off from this schedule. It has the inappropriate Modbus/RTU assumption "no response means retry ASAP" design. Once a second each request is repeated whether the previous request was answered or not.

The above system works fine – until a 2nd Modbus/TCP client comes in and also starts issuing 8 requests per second. Now the bridge has 2 TCP queues, each receiving 8 requests per second. However, the Modbus serial channel is still limited to 10 polls per second. Assuming a round-robin design, this means the bridge can only remove and process 5 from each TCP queue each second.

You should see the "Meltdown" potential here – after 1 minute (a mere 60 seconds) each queue will have about 180 old, stale requests queued and waiting. What worked fine a minute ago, now no longer works fine!

Worse, if the client is using the Modbus/TCP Sequence Number then likely ALL responses it is receiving are old and being discarded. So even though the bridge is returning 5 valid Modbus responses per second to each Client, the client is seeing none of them and timing out all 8 newer requests each second. The functioning serial slave is now marked as "off-line".

Much worse, since the Modbus/TCP specification incorrectly calls the sequence number optional, there are a few Modbus/TCP clients that do not use sequence numbers. These clients will be blissfully ignorant that they may be incorrectly interpreting the response data. For example, if the client requests 125 registers from slave #3 starting at 400001 and also pipelines a 2nd request for 125 registers from slave #3 starting at 400126, when a response comes back from slave #3 with 250 bytes of data, which request is it for? If the client assumed the Modbus/RTU design for timeouts, it may discard the 1st request and improperly apply the data to the 2nd request.

This system killer becomes almost certain in high-latency systems such as satellite – but this will be covered in the Infrastructure section below.

- A Modbus/TCP client MUST assign a unique sequence number to each Modbus/TCP request and verify that only a response with a matching sequence number is processed as the response. *The Modbus/TCP spec incorrectly calls this optional – IT IS NOT!*
- A Modbus/TCP server MUST echo (return) the first 2 bytes of the header unchanged and uninterpreted. The Modbus/TCP spec correctly requires this.

Good Socket verse Good Response

Many commercial products have bugs in this respect. Most products correctly assume that a good socket doesn't necessarily mean an assured response, however they do positive testing of their Modbus/TCP clients with commercial Modbus/TCP servers only. How do you force a PLC or I/O block accept a Modbus/TCP socket, but not respond? Well, you cannot.

- ○ A Modbus/TCP client must be tested to verify that even if a socket opens, the request timeout works properly.

Common Client Access Paradigms

This section covers the behavior your Modbus/TCP server may see when connected to common commercial applications.

COMMON OPC/DDE DRIVERS

Each device is an independent topic and thus sometimes 1 TCP socket.

This is the most common TCP socket behavior, as (in theory at least) it allows priority handling of individual topics. However, it overlooks the Bridge maker's fact that with RS-485 32 or more Modbus/TCP "topics" may be sharing the same IP address. Considering that each TCP socket may require up to 10K of memory, all small Modbus/TCP servers support some limited number of sockets.

Suppose a Bridge (Modbus/TCP server) supports only 8 TCP sockets at once and there is a multi-drop of RS-485 Modbus/RTU slaves attached. In this design, an OPC server (Modbus/TCP client) can only access 8 of those RS-485 slaves at once. If you want to allow a 2nd OPC server to act as warm-standby, then this further limits access to only 4 slaves per RS-485 multi-drop!

- o A Modbus/TCP client must allow – at least as user option – consolidating requests to a single IP into a single socket.
- o A Modbus/TCP server with limited socket resources should support an idle time-out and close client connections that stop polling the server.

MODICON PLC

Modicon PLC – MAST block

Many Modicon PLC support a function referred to as a Master block. This block allows the PLC to issue Modbus read or write requests over ModbusPlus™ or Modbus/TCP (Ethernet). You can have many MAST blocks in one program, and they can all target the same IP address or many IP addresses. It has been my experience that MAST requests triggered sequentially to a single IP share the same socket, while those triggered in parallel or to different IP use different concurrent sockets.

The MAST block assumes all outstanding requests will be answered either with the desired data or an exception response. Therefore if your Modbus/TCP server doesn't issue an exception response 0x0Bon timeout, users must add timer logic to their program to abort the MAST block on timeout.

- o MAST block users should expect to trigger MAST block sequentially unless they know their target Modbus/TCP server supports enough TCP sockets to do it in parallel.
- o A Modbus/TCP server must answer all client requests either with the expected answer or an exception response. No response is not acceptable.

Modicon NOE I/O Scanner

Ethernet-based Modicon PLC support a function referred to as the "I/O Scanner" or sometimes the "NOE". It runs in the background acting as a Modbus/TCP client to copy data to and from the PLC's memory and remote Modbus/TCP servers. Unlike the MAST block that is part of your ladder logic, the I/O scanner is a bit of magic and much less work to set up. However, the NOE I/O Scanner has some very specific behavior that adds limitations based on Modbus/TCP server capability.

The NOE I/O Scanner attempts to open one read - and a 2nd write socket if required - to each slave configured. So 2 sockets to each RS-485 slave attached to the bridge may be required. Review the OPC discussion above. A system with 1 NOE I/O scanner and 1 OPC server combined with a bridge supporting only 8 sockets can only have 2 active RS-485 slaves! 4 sockets are required for the NOE I/O Scanner and 2 for the OPC server.

Next, early versions of the NOE I/O scanner rapidly open each socket, use it for 1 poll cycle, and then close it again. The good effect of this is it doesn't lock that resource 100% of time. The bad effect is that a socket resource on your Modbus/TCP server takes a finite amount of time to close, clean-up, and be available to

reopen. So if the bridge has exactly 8 sockets, likely an NOE I/O scanner assuming 8 sockets existing will have intermittent errors as occasionally it's attempt to open a 7th or 8th socket hits a time when those resources are still cycling from closed to open and not available. I believe this behavior has been modified in newer versions.

Modicon PLC – Gratuitous ARP

Modicon PLC support redundancy by sharing IP Addresses. When a backup PLC takes over from the primary PLC, it confiscates the active IP, issues a Gratuitous ARP, and assumes remote partners will update their ARP cache with the new IP-to-MAC mapping within about 250msec. This is a low-level stack issue that you may not have much control over. If your stack doesn't properly remap the ARP cache, a work around may be to manually flush the ARP cache each time a socket open or request fails. Otherwise, your client may not recognize the new active PLC for many minutes.

- A Modbus/TCP server must issue a Gratuitous ARP on start up.
- A Modbus/TCP client must properly use Gratuitous ARP received to modify the ARP cache.

SURVIVING WIDE-AREA-NETWORK (WAN)

One of the blessings of TCP/IP is also its greatest curse. TCP/IP runs on literally hundreds of physical media – that's good for paper system designs, but bad for fine-tuning performance on real systems. In the past many SCADA systems relied on a menagerie of communications technologies – their 250 remote stations may have been managed as 120 dialups, 70 radio links, 40 leased lines and 20 satellite links. Now with TCP/IP, they can be viewed as 250 IP addresses.

Is this design "simpler"? Well, on paper it is. The lower level technologies are still as complex as before – perhaps more complex. But the issues have been localized and hidden within subsystems. Plus users now have an every growing range of new options such as GSM, DSL, Broadband and other emerging technologies that can be deployed with little or no impact on the SCADA.

TCP/IP on a WAN behaves very differently than the common TCP/IP on Ethernet used and test by most vendors!

Latency

The largest change in supporting WAN is the greatly increased latencies. Some Modbus/TCP clients make unrealistic expectations that all requests can be answered in from 10 to 1000 milliseconds. However many WAN media won't support this. The average satellite link has round-trip delays in the range of 1-3 seconds. If using a low-priority (ie: low cost) channel, this could easily stretch to 10 seconds or more as the low-priority traffic waits for higher cost traffic.

- A Modbus/TCP client should support a user-defined Service Timeout of at least 60 seconds.

The other issue with large latencies is that it increases the probability that old responses arrive after they are no longer expected. This is aggravated in Modbus since a response doesn't include explicit information about what the data bytes represent. A client application that accidentally matches an old response to a new, different request risks serious system disruption. So all client applications must assign new requests a unique sequence number and validate that only a response with the matching sequence number is processed.

- A Modbus/TCP client must use and validate the Modbus/TCP Sequence Number
- A Modbus/TCP server must properly echo the Sequence Number to the requesting client.

Fragmentation

The other big change in supporting WAN is the impact of message fragmentation. Ethernet's inherent data limit of about 1500 bytes is a real luxury in the world of wide-area packet networks. Many WAN links tend to break TCP packets up into smaller fragments – for example 128 byte or even 32 byte fragments. You'll find messages behave very differently based on data size. For example, a satellite customer may find a 43 register Modbus read averages 1.5 seconds, while a 44 register read averages 2.8 seconds. The difference is caused by the 44 register read being broken into 2 fragments and incurring 2 queuing delays, while the 43 register read is sent as 1 fragment with a single delay. (I just picked 43 and 44 to illustrate the point – don't build those into your application!)

You should allow the user to force a reduced register or coil count in all reads or writes. Most OPC servers already support this for the Modbus write command since the formal specification limits all writes to 100 registers but many

devices permit writes of 122 registers. The upper limit would be the Modbus specified limit, and the lowest limit could in theory be 1.

- o A Modbus/TCP client should support a user-defined max size limit for both read and writes.

The other issue of fragmentation is how long it takes a Modbus/TCP server to receive a full message. A product which has a fixed limit of say 2 seconds for fragmentation reassembly will likely not be useable with satellite systems.

- o A Modbus/TCP server should support a user-defined Reassembly Timeout of at least 30 seconds (1/2 the 60 second client Service Timeout?)

SECURITY

Modbus grew up in an age of very-local area networking – meaning a single master talking a few hundred feet to a few dedicated slaves. The idea that a master hard wired to slaves is unauthorized to communicate with them was not an issue. Therefore Modbus has no concept of data communications security. There is no password just to ask for data (programs can be password protected, but that's a different issue). There is no encryption.

Firewalls, Closed Systems, and Wireless

Most people either use no security, or rely on the notion that their network is a private, closed system and therefore any device physically able to connect is inherently authorized to communicate.

However, in this day of Internet connectivity and wireless media, that is a pretty fragile notion of “security”. A firewall has been called the “Chocolate-Covered Cherry” approach to security – hard on the outside and soft on the inside. Once a Modbus/TCP client is past the firewall, they have complete access to all Modbus/TCP servers. Plus history is ripe with stores of the unintentional hole in this hard shell – the executive who puts a modem into his PC or the engineer who uses a standard dial-up link for system support.

And anyone who worries about “soft on the inside” security should see stars when “wireless” is used. The magic of wireless networking is ... well, you don't need wires. Any Modbus/TCP client within the appropriate physical space can access data with little chance of detection. There have been several cases of disgruntled employees or contractors using wireless access parameters (passwords etc) to access SCADA device from their cars to disrupt operations.

Router Filtering – “IT” managed

The most traditional approach to intra-system security is to use routers or VLAN that can be configured to permit or deny IP traffic based on IP. I won't go into detail on these, as there are many vendors willing to sell you such things. Just be aware that they speak “IT-ish”, not “Automation-ish” so you're best to coordinate with your IT department to act as translator and cultural advisor.

IP Filtering – end-user managed

The easiest form of “user applied” security is for Modbus/TCP servers to add “IP Filtering”. This basically means the user can predefine a list of specific IP addresses or subnets that are permitted or forbidden to connect to this server.

Here is a simple example.

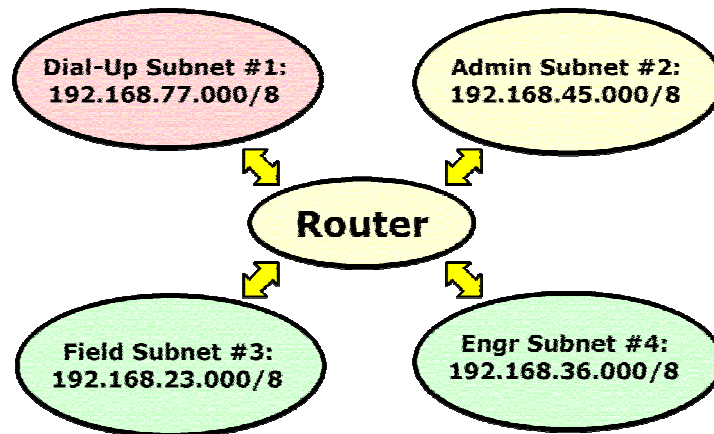


FIGURE 2 – EXAMPLE SUBNETWORK DESIGN

A reasonable security definition based on IP would be that any field device can access any other field device or it can be accessed by a list of 4 PC on the engineering network – say 2 engineering workstations and 2 OPC/Data servers. Any other access is not desired – especially if by the dialup modem bank.

So a Modbus/TCP server IP filtering mechanism could be defines as follows:

Rule	Mask	IPv4	Port	Description
Permit	255.255.255.000	192.168.23.000	TCP: 502	Other Field Devices
Permit	255.255.255.255	192.168.36.126	TCP: 502	Jim’s Engr PC
Permit	255.255.255.255	192.168.36.8	TCP: 502	Tech Asst’s Engr PC
Permit	255.255.255.255	192.168.36.200	TCP: 502	OPC Server A
Permit	255.255.255.255	192.168.36.201	TCP: 502	OPC Server B
Deny	255.255.255.000	192.168.72.000	TCP: 502	Any Dialup/PPP Device

These rules would be applied on opening sockets only and thus have minimal impact on Modbus/TCP clients who use a sustained socket. Actually, the final “deny” rule wouldn’t be required since by definition any IP not permitted is denied – however I added it to illustrate the point.

High-Security and public WAN support

If you are launching your Modbus/TCP traffic onto a public network or very concerned about security, none of the above approaches may be suitable. Instead, you’ll likely need to fall-back on the well-established and existing mechanisms for encryption and 3rd-party authentication.

Encryption – such as SSL (Secure-Sockets-Layer) – is common these days, plus there are many newer, wizzier standards that can also be used. Unfortunately SSL and most other encryption systems are workstation and virtual-memory centric. They assume large arrays of precalculated data exists to speed up the process – data likely in the 256K byte to many MB range.

So far I’ve never seen an off-the-shelf Modbus/TCP server device with SSL build in – and this is an all or nothing solution. Both ends must support the same encryption standard or it cannot be used.

CONCLUSION

Modbus/TCP is a very simple Ethernet protocol that allows rapid creation of multi-vendor systems. Just be aware that Modbus/TCP requires some different behaviors than you may be used to with Modbus serial. Plus be aware that the wider opportunity TCP/IP offers for wide-area-network use adds yet some more new behaviors that neither Ethernet nor Serial savvy Modbus developers may be used to.