

## FEATURE ARTICLE

Tracy Thomas

# A Practical Guide to TCP/IP Protocols

Tracy gives us a brief introduction to some of the issues involved in using TCP/IP protocols and tells us what information is most often misunderstood by programmers who are new to the networking world.



It's difficult for embedded systems programmers to sift through the wealth of reference material regarding networking and TCP/IP protocols. Yet, an increasing number of embedded applications benefit from the addition of networking capability. This article is a practical guide to the basics of TCP/IP and contains information that is most often misunderstood by programmers who are new to networking.

### THE BASICS

The first point to understand is that TCP/IP often refers to the family of protocols, which includes TCP (Transport Control Protocol), UDP (User Datagram Protocol), IP (Internet Protocol), and some underlying link layers such as Ethernet. Data sent using the TCP protocol is called a segment, and data sent using the UDP protocol is referred to as a packet. IP is the network layer that lies under TCP and UDP. IP provides connectionless packet delivery to a

specified host address, making it unreliable because IP packets (called datagrams) can be lost, duplicated, or delivered out of order. The advantage of IP is that a datagram provides a universal method for delivering data, independent of the underlying network technology.

UDP and TCP sit on top of IP at the transport layer. Both use port numbers to de-multiplex data sent to a host. A port number is specific to an application. The use of multiple port numbers allows a single host to run multiple networking applications. Each UDP packet and TCP segment has a source and destination port number.

The host that waits for incoming connections is called the server, and the host that initiates a connection is the client. Servers "listen" to well-known port numbers for common applications such as FTP (File Transfer Protocol), e-mail, and HTTP. Clients generally choose a random source port number and connect to a server at a well-known port. Custom applications should use a port number greater than 1024, because port numbers less than 1024 are reserved for well-known applications.

### DATA DELIVERY

UDP provides best-effort data delivery with an optional checksum to preserve data integrity. UDP packets have the same reliability as IP, where packets are not guaranteed to be received in order at the remote host. TCP provides a reliable stream of data by using sequence and acknowledge numbers to recover lost data, detect out-of-order segments, and resolve transmission errors.

So, the TCP protocol is more complicated than the UDP protocol. TCP is connection-oriented, but UDP lacks connections. This implies that, to send a UDP packet, a client addresses a packet to a remote host and sends it

without any preliminary contact to determine if the host is ready for data.

If a client sends a UDP packet to a host that is not listening to the destination port, the host returns an ICMP (Internet Control Message Protocol) error. The remote host can send back this error message as long as it is running a TCP/IP stack capable of processing incoming datagrams to the ICMP layer. Thus, the client has notification if the remote host is not accepting data, but this notification is limited. The client receives no confirmation if the data was received.

In addition to its lack of reliability, UDP also has no throttling mechanism. UDP packets can be sent at full speed—as fast as the underlying physical device can send them. With slow processors, UDP's lack of overhead can make a significant difference in the throughput when compared to TCP. With fast processors, the difference is not as significant. The lack of an end-to-end connection in UDP means that it can be used to send one-to-many and many-to-many type messages. This feature is used to send broadcast and multicast messages. One example is Dynamic Host Control Protocol, which uses a broadcast message when booting to find a DHCP server to supply network configuration information.

TCP provides a connection-oriented, reliable stream of data. Before sending data to a remote host, a TCP connection must be established. This means that only data between two end hosts may be exchanged over a TCP connection. Establishing a TCP connection involves a three-way handshake. First, the client sends a SYN segment to the server to request connection. Second, the server sends the client a SYN-ACK segment to request connection and acknowledge the client's connection request segment. Third, the client sends ACK to the server to acknowledge the server's connection request segment.

Every TCP segment that contains data is acknowledged to provide reliability. The acknowledge segments themselves are not acknowledged to prevent an infinite recursion. When a connection is requested, the client

randomly picks an initial sequence number. In step two of the connection, the server picks its own initial sequence number and acknowledges the client's sequence number in its acknowledge number. Thus, every TCP segment contains a sequence number that acts as a placeholder in the datastream and an acknowledge number to notify the remote host of reception of its data.

Transmission errors are resolved by keeping each data segment until it's acknowledged. If the data is not acknowledged in a set time period, the client retransmits the data to the remote host. If the data still is not acknowledged, the client continually transmits the segment with successively greater time intervals between transmissions. After a set time passes without an acknowledgment, the connection will not be usable, and an error condition will be returned to the application. A TCP connection can remain open indefinitely. If no data is sent on an open connection, the connection remains open. The only way for a host to determine if the remote side is still there is to send data. A TCP connection remains open until it is closed by both hosts.

The TCP close sequence requires four steps. First, the client initiates the close by sending a FIN segment to the server. At this point, the client may no longer send data to the remote host, but it can still receive data. Second, the server sends an ACK segment to the client to acknowledge the FIN segment. Although it's not common, the server may still send data to the client. This condition is called the half-close. Third, the server sends a FIN segment to the client to close its connection. Now, the server may no longer send any data. Finally, the client sends an ACK segment to acknowledge the server's FIN segment.

After the final step, the connection is closed. However, the client must not reuse this connection (port number) for a period called the 2MSL timeout. This delay attempts to ensure that a subsequent connection will

not be confused with the original connection. In the world of enterprise networking, the timeout is two minutes and doesn't significantly effect the client because the client system has vast resources available for creating new connections. In an embedded system, two minutes is equivalent to an eternity and resources are often limited. Therefore, in embedded TCP/IP implementations, the 2MSL timeout is often reduced.

## REGULATING DATA

TCP regulates the data rate, or throughput, with a sliding window. With TCP, a client cannot establish a connection and blast the remote host with data at full speed. The data flow is regulated by the requirement for acknowledgement of data and by the TCP window. In each TCP segment, the host advertises the amount of data it's ready to accept. This amount is the TCP window (see Figure 1). The sliding window refers to the progression of data sequence numbers as data is sent throughout a connection. The window size advertised specifies the maximum amount of data the sender can have outstanding (not acknowledged).

Data to the left of the current window is in the past and can't be resent. Data to the right of the current window can't be sent until the window moves. So, only data within the sliding window may be sent to the remote host. The sliding window moves to the right only when a data segment is acknowledged. Thus, the acknowledgement of data along with the window keeps TCP from sending data too fast.

For example, as shown in Figure 1,

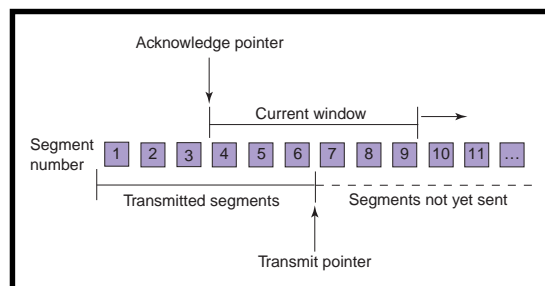


Figure 1—The TCP data window slides to the right as data is sent and acknowledged. TCP's sliding window acts as a throttle on a connection to limit the data rate to what the data receiver can actually handle.

the client can send segments 7, 8, and 9, while segments 4, 5, and 6 are waiting for acknowledgement. When you receive an acknowledgement for segment 4 and the window slides one place to the right, you can send segment 10. This concept of the sliding window is also referred to as flow control and limits the amount of traffic one host can place on the network to the amount that the remote host is actually processing. The remote host can close its window (advertise zero bytes can be sent). If the remote window is closed, the client can no longer send data to the remote host, but instead must probe the remote host to discover when the window opens.

One important feature of TCP to understand is that the data is delivered as a stream. If the client application performs ten 128-byte writes, there is no way to know if the underlying TCP will deliver ten 128-byte segments, or one 1280-byte segment. Application developers note that an additional protocol often needs to be used to mark the beginning, end, and type of data. FTP, Telnet, and e-mail (SMTP) are common application examples that use protocol headers in addition to TCP. When you design your own application with TCP, you'll need a way to break up the stream of data into recognizable parts.

## PROS AND CONS

Now you understand that TCP's reliability comes at the price of being more complicated than UDP. Embedded programmers need to understand the basic concepts of TCP and UDP to design the best application. Keep in mind that UDP is connectionless, unreliable, and has no throttle on its throughput. TCP offers a reliable datastream but more processing overhead and reduced throughput.

Embedded designers have many concerns—for some, code size is important, and for others, data rate and reliability are a higher priority. The simplicity of UDP is misleading. UDP applications typically require code at the application layer to build reliability. Data sequencing is the most common addition because it allows the application to mark missing data and

to resolve out-of-order packets.

Some example applications illustrate the merits of using UDP and TCP for different needs. A central monitor tracks the status of sensors throughout a building for temperature and humidity control, receiving data from each sensor once per minute. This application may use either UDP or TCP. If an individual report is dropped with minimal consequences, then UDP may be used. If the sensor data needs to be tracked in a database and each reading is important, then TCP should be used. In this example, reliability guides the choice. Restricting the protocol to UDP may allow this type of application to fit on a minimal microprocessor that otherwise could not be network-enabled.

Video-conferencing and voice-over IP are ideal examples for using UDP. Throughput is important in the real-time reception of audio and video. By the time a reliable protocol could retransmit a packet, the moment to play that packet has passed. However, these types of applications require a method to track data sent and received, so an application layer protocol is layered on top of UDP to provide data sequencing.

TCP is appropriate for applications that require reliability in data transmission. Data acquisition and control applications are some examples. Common applications such as e-mail and the 'Net use TCP, as well.

Embedded applications benefit from standardized network protocols. With minimal resources, an embedded device can be monitored and controlled remotely. This article is a brief introduction to some of the issues involved in using the TCP/IP protocols. Embedded programmers should consider the positive and negative aspects of the networking protocols before choosing the appropriate set for their design. ■

*Tracy Thomas received her PhD. in high-energy physics from Northwestern University in 1997. During her thesis research at Fermi National Accelerator Laboratory, she did network-*

*ing projects and programming for embedded data acquisition systems. She is currently a software engineer at US Software, working on embedded TCT/IP and related protocols.*

## REFERENCES

- J.B. Postel, ed. "Transmission Control Protocol," RFC 793, September, 1981.
- W.R. Stevens, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, Massachusetts, 1994.
- D.E. Comer, *Internetworking with TCP/IP Volume 1*, Prentice Hall, New Jersey, 1995.