

## **Transport Layer**

### **TCP/IP Ports: Transport Layer (TCP/UDP) Addressing**

A typical host on a TCP/IP internetwork has many different software application processes running concurrently. Each generates data that it sends to either TCP or UDP, which in turn passes it to IP for transmission. This [multiplexed stream of datagrams](#) is sent out by the IP layer to various destinations. Simultaneously, each device's IP layer is receiving datagrams that originated in numerous application processes on other hosts. These need to be demultiplexed, so they end up at the correct process on the device that receives them.

#### ***Multiplexing and Demultiplexing Using Ports***

The question is: how do we demultiplex a sequence of IP datagrams that need to go to many different application processes? Let's consider a particular host with a single network interface bearing the IP address 24.156.79.20. Normally, every datagram received by the IP layer will have this value in the IP *Destination Address* field. Consecutive datagrams received by IP may contain a piece of a file you are downloading with your Web browser, an e-mail sent to you by your brother, and a line of text a buddy wrote in an IRC chat channel. How does the IP layer know which datagrams go where, if they all have the same IP address?

The first part of the answer lies in the *Protocol* field included in the header of each IP datagram. This field carries a code that identifies the protocol that sent the data in the datagram to IP. Since most end-user applications use TCP or UDP at the transport layer, the *Protocol* field in a received datagram tells IP to pass data to either TCP or UDP as appropriate.

Of course, this just defers the problem to the transport layer: both TCP and UDP are used by many applications at once. This means TCP or UDP must figure out which process to send the data to. To make this possible, an additional addressing element is necessary. This address allows a more specific location—a software process—to be identified within a particular IP address. In TCP/IP, this transport layer address is called a *port*.



**Key Concept:** TCP/IP transport layer addressing is accomplished using TCP and UDP *ports*. Each port number within a particular IP device identifies a particular software process.

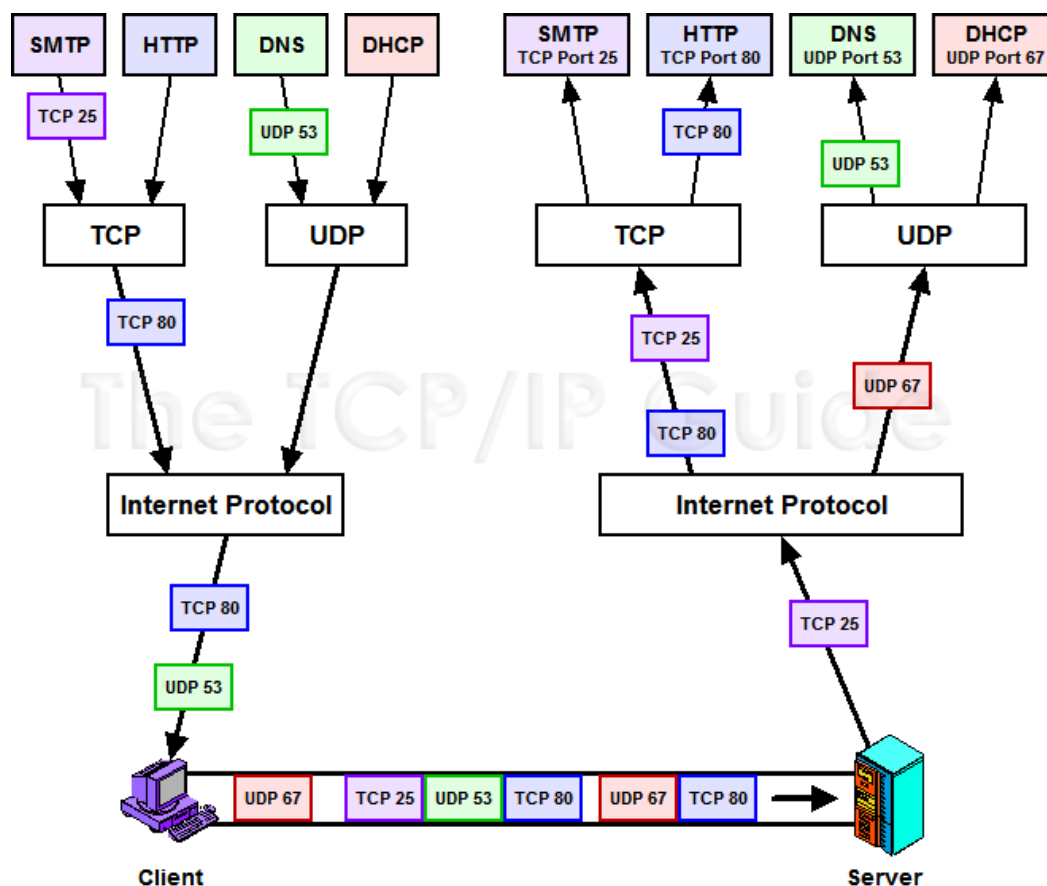
#### ***Source Port and Destination Port Numbers***

In both UDP and TCP messages two addressing fields appear, for a *Source Port* and a *Destination Port*. These are analogous to [the fields for source address and](#)

[destination address at the IP level](#), but at a higher level of detail. They identify the originating process on the source machine, and the destination process on the destination machine. They are filled in by the TCP or UDP software before transmission, and used to direct the data to the correct process on the destination device.

TCP and UDP port numbers are 16 bits in length, so valid port numbers can theoretically take on values from 0 to 65,535. [As we will see in the next topic](#), these values are divided into ranges for different purposes, with certain ports reserved for particular uses.

One fact that is sometimes a bit confusing is that both UDP and TCP use the same range of port numbers, and they are independent. So, in theory, it is possible for UDP port number 77 to refer to one application process and TCP port number 77 to refer to an entirely different one. There is no ambiguity, at least to the computers, because as mentioned above, each IP datagram contains a *Protocol* field that specifies whether it is carrying a TCP message or a UDP message. IP passes the datagram to either TCP or UDP, which then sends the message on to the right process using the port number in the TCP or UDP header. This mechanism is illustrated in Figure 198.



**Figure 198: TCP/IP Process Multiplexing/Demultiplexing Using TCP/UDP Ports**


This is a more “concrete” version of Figure 197, showing how TCP and UDP ports are used to accomplish software multiplexing and demultiplexing. Again here there are four different TCP/IP applications communicating, but this time I am showing only the traffic going from the client to the server. Two of the applications are using TCP and two UDP. Each application on the client sends messages using a specific TCP or UDP port number. These port numbers are used by the server’s UDP and TCP software to pass the datagrams to the appropriate application process.


In practice, having TCP and UDP use different port numbers is confusing, especially for the reserved port numbers used by common applications. For this reason, by convention, most reserved port numbers are reserved for both TCP and UDP. For example, port #80 is reserved for the Hypertext Transfer Protocol (HTTP) for both TCP and UDP, even though HTTP only uses TCP. [We'll examine this in greater detail in the next topic.](#)

## ***Summary of Port Use for Datagram Transmission and Reception***

So, to summarize, here's the basics of how transport-layer addressing (port addressing) works in TCP and UDP:

- **Sending Datagrams:** An application specifies the source and destination port it wishes to use for the communication. These are encoded into the TCP or UDP header, depending on which transport layer protocol the application is using. When TCP or UDP passes data to IP, IP indicates the protocol type appropriate for TCP or UDP in the *Protocol* field of the IP datagram. The source and destination port numbers are encapsulated as part of the TCP or UDP message, within the IP datagram's data area.
- **Receiving Datagrams:** The IP software receives the datagram, inspects the *Protocol* field and decides to which protocol the datagram belongs (in this case, TCP or UDP, but of course there are other protocols that use IP directly, such as ICMP). TCP or UDP receives the datagram and passes its contents to the appropriate process based on the destination port number.

 **Key Concept:** Application process multiplexing and demultiplexing in TCP/IP is implemented using the IP *Protocol* field and the UDP/TCP *Source Port* and *Destination Port* fields. Upon transmission, the *Protocol* field is given a number to indicate whether TCP or UDP was used, and the port numbers are filled in to indicate the sending and receiving software process. The device receiving the datagram uses the *Protocol* field to determine whether TCP or UDP was used, and then passes the data to the software process indicated by the *Destination Port* number.

 **Note:** As an aside, I should point out that the term *port* has many meanings aside from this one in TCP/IP. For example, a physical outlet in a network device is often called a *port*. Usually one can discern whether the “port” in question refers to a hardware port or a software port from context, but you may wish to watch out for this.

## **TCP/IP Sockets and Socket Pairs: Process and Connection Identification**

The preceding topics have illustrated the key difference between addressing at the level of the Internet Protocol, and addressing as it is seen by application processes. To summarize, at layer three, an IP address is all that is really important for properly transmitting data between IP devices. In contrast,

application protocols must be concerned with [the port assigned to each instance of the application](#), so they can properly use TCP or UDP.

### **Sockets: Process Identification**

What this all means is that the overall identification of an application process actually uses the **combination** of the IP address of the host it runs on—or the network interface over which it is talking, to be more precise—and the port number which has been assigned to it. This combined address is called a *socket*. Sockets are specified using the following notation:

<IP Address>:<Port Number>

So, for example, if we have a Web site running on IP address 41.199.222.3, the socket corresponding to the HTTP server for that site would be *41.199.222.3:80*.



**Key Concept:** The overall identifier of a TCP/IP application process on a device is the combination of its IP address and port number, which is called a *socket*.

You will also sometimes see a socket specified using a host name instead of an IP address, like this:

<Host Name>:<Port Number>

To use this descriptor, the name must first be resolved to an IP address using DNS. For example, you might find a Web site URL like this: “<http://www.thisisagreatsite.com:8080>”. This tells the Web browser to first *resolve* the name “[www.thisisagreatsite.com](http://www.thisisagreatsite.com)” to an IP address using [DNS](#), and then send a request to that address using the non-standard server port 8080, which is occasionally used instead of port 80 since it resembles it. ([See the discussion of application layer addressing using URLs for much more.](#))

The *socket* is a very fundamental concept to the operation of TCP/IP application software. In fact, it is the basis for an important TCP/IP application program interface (API) with the same name: *sockets*. A version of this API for Windows is called *Windows Sockets* or *WinSock*, which you may have heard of before. These APIs allow application programs to easily use TCP/IP to communicate.

### **Socket Pairs: Connection Identification**

So, the exchange of data between a pair of devices consists of a series of messages sent from a socket on one device to a socket on the other. Each device will normally have multiple such simultaneous conversations going on. In the case of [TCP](#), a connection is established for each pair of devices for the

duration of the communication session. These connections must be managed, and this requires that they be uniquely identified. This is done using the pair of socket identifiers for each of the two devices that are connected.



**Key Concept:** Each device may have multiple TCP connections active at any given time. Each connection is uniquely identified using the combination of the client socket and server socket, which in turn contains four elements: the client IP address and port, and the server IP address and port.

Let's return to the example we used in the previous topic (Figure 199). We are sending an HTTP request from our client at 177.41.72.6 to the Web site at 41.199.222.3. The server for that Web site will use well-known port number 80, so its socket is 41.199.222.3:80, as we saw before. We have been ephemeral port number 3,022 for our Web browser, so the client socket is 177.41.72.6:3022. The overall connection between these devices can be described using this socket pair:

(41.199.222.3:80, 177.41.72.6:3022)

For much more on how TCP identifies connections, see [the topic on TCP ports and connection identification](#) in [the section on TCP fundamentals](#).

Unlike TCP, [UDP](#) is a connectionless protocol, so it obviously doesn't use connections. The pair of sockets on the sending and receiving devices can still be used to identify the two processes exchanging data, but since there are no connections the socket pair doesn't have the significance that it does in TCP.

## TCP/IP Client (Ephemeral) Ports and Client/Server Application Port Use

The significance of the asymmetry between clients and servers in TCP/IP becomes evident when we examine in detail how port numbers are used. Since clients initiate application data transfers using TCP and UDP, it is they that need to know the port number of the server process. Consequently, it is servers that are required to use universally-known port numbers. Thus, [well-known and registered port numbers](#) identify server processes. They are used as the destination port number in requests sent by clients.

In contrast, servers respond to clients; they do not initiate contact with them. Thus, the client doesn't need to use a reserved port number. In fact, this is really an understatement: a server **shouldn't** use a well-known or registered port number to send responses back to clients. The reason is that it is possible for a particular device to have both client and server software of the same protocol

running on the same machine. If a server received an HTTP request on port 80 of its machine and sent the reply back to port 80 on the client machine, it would be sending the reply to the client machine's HTTP **server** process (if present) and not the client process that sent the initial request.

To know where to send the reply, the server must know the port number the client is using. This is supplied by the client as the *Source Port* in the request, and then used by the server as the destination port to send the reply. Client processes don't use well-known or registered ports. Instead, each client process is assigned a temporary port number for its use. This is commonly called an *ephemeral port number*.



**Note:** Your \$10 word for the day: *ephemeral*: “short-lived; existing or continuing for a short time only.” — Webster's Revised Unabridged Dictionary.

### ***Ephemeral Port Number Assignment***

Ephemeral port numbers are assigned as needed to processes by the TCP/IP software. Obviously, each client process running concurrently needs to use a unique ephemeral port number, so the TCP and UDP layers must keep track of which are in use. These port numbers are generally assigned in a *pseudo-random* manner from a reserved pool of numbers. I say “pseudo-random” because there is no specific meaning to an ephemeral port number assigned to a process, so a random one could be selected for each client process. However, since it is necessary to reuse the port numbers in this pool over time, many implementations use a set of rules to minimize the chance of confusion due to reuse.

Consider a client process that just used ephemeral port number 4,121 to send a request, received a reply, and then terminated. Suppose we immediately reallocate 4,121 to some other process. However, the server accessed by the prior user of port 4,121 for some reason sent an extra reply. It would go to the new process, creating confusion. To avoid this, it is wise to wait as long as possible before reusing port number 4,121 for another client process. Some implementations will therefore cycle through the port numbers in to ensure the maximum amount of time elapses between consecutive uses of the same ephemeral port number.



**Key Concept:** Well-known and registered port numbers are needed for server processes since a client must know the server's port number to initiate contact. In contrast, client processes can use any port number. Each time a client process initiates a UDP or TCP communication it is assigned a temporary,



or *ephemeral*, port number to use for that conversation. These port numbers are assigned in a pseudo-random way, since the exact number used is not important, as long as each process has a different number.

### ***Ephemeral Port Number Ranges***

The range of port numbers that is used for ephemeral ports on a device also depends on the implementation. The “classic” ephemeral port range was established by the TCP/IP implementation in BSD (Berkeley Standard Distribution) UNIX, where it was defined as 1,024 to 4,999, providing 3,976 ephemeral ports. This seems like a very large number, and it is indeed usually more than enough for a typical client. However, the size of this number can be deceiving. Many applications use more than one process, and it is theoretically possible to run out of ephemeral port numbers on a very busy IP device. For this reason, most of the time the ephemeral port number range can be changed. The default range may be different for other operating systems.

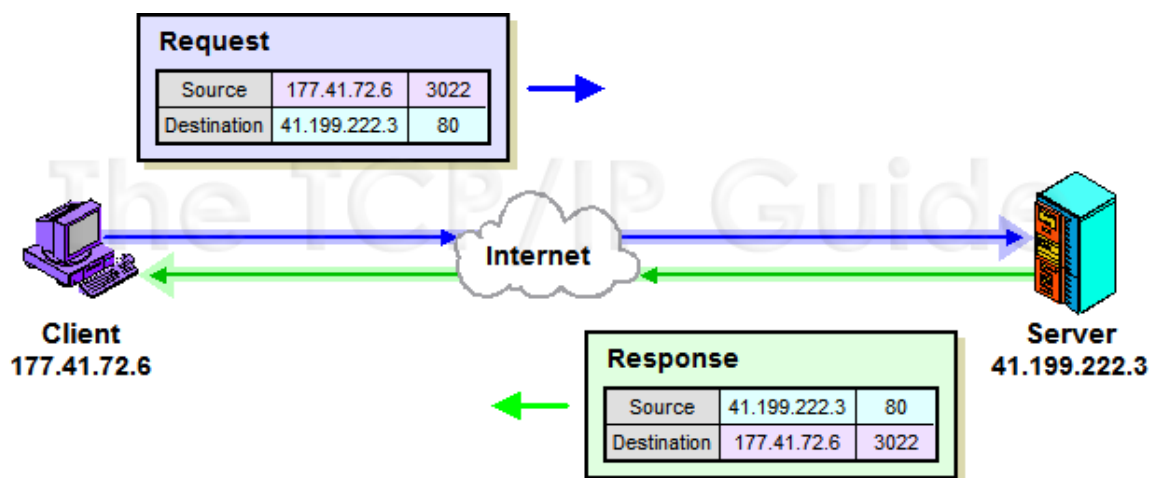
Just as well-known and registered port numbers are used for server processes, ephemeral port numbers are for client processes only. This means that the use of a range of addresses from 1,024 to 4,999 does not conflict with the use of that same range for registered port numbers as seen in the previous topic.

### ***Port Number Use During a Client/Server Exchange***

So, let's return to the matter of client/server application message exchange. Once assigned an ephemeral port number, it is used as the source port in the client's request TCP/UDP message. The server receives the request, and then generates a reply. In forming this response message, it *swaps* the source and destination port numbers, just as it does the source and destination IP addresses. So, the server's reply is sent **from** the well-known or registered port number on the server process, back **to** the ephemeral port number on the client machine.

Phew, confusing... quick, back to our example! ☺ Our Web browser, with IP address 177.41.72.6 wants to send an HTTP request to a particular Web site at IP address 41.199.222.3. The HTTP request is sent using TCP, with a *Destination Port* number of 80 (the one reserved for HTTP servers). The *Source Port* number is allocated from a pool of ephemeral ports; let's say it's port 3,022. When the HTTP request arrives at the Web server it is conveyed to port 80 where the HTTP server receives it. That process generates a reply, and sends it back to 177.41.72.6, using *Destination Port* 3,022 and *Source Port* 80. The two processes can exchange information back and forth; each time the source port number and destination port number are swapped along with the source and destination IP addresses. This example is illustrated in Figure 199.





**Figure 199: TCP/IP Client/Server Application Port Mechanics**

This highly simplified example shows how clients and servers use port numbers for a request-reply exchange. The client is making an HTTP request and sends it to the server at HTTP's well-known port number, 80. Its port number for this exchange is the pseudo-randomly-selected 3,022. The server sends its reply back to that port number, which it reads from the request.

**Key Concept:** In most TCP/IP client/server communications, the client uses a random ephemeral port number and sends a request to the appropriate reserved port number at the server's IP address. The server sends its reply back to whatever port number it finds in the *Source Port* field of the request.

## TCP and UDP Overview and Role In TCP/IP

The transport layer in a protocol suite is responsible for a specific set of functions. For this reason, one might expect that the TCP/IP suite would have a single main transport protocol to perform those functions, just as it has [IP](#) as its core protocol at the network layer. It is a curiosity, then, that there are **two** different widely-used TCP/IP transport layer protocols. This arrangement is probably one of the best examples of [the power of protocol layering](#)—and hence, an illustration that it was worth all the time you spent learning to understand that pesky [OSI Reference Model](#). ☺

### ***Differing Transport Layer Requirements in TCP/IP***

Let's start with a look back at layer three. In my [overview of the key operating characteristics of the Internet Protocol](#), I described several important limitations of how IP works. The most important of these are that IP is *connectionless*, *unreliable* and *unacknowledged*. Data is sent over an IP internetwork without first establishing a connection, using a “best effort” paradigm. Messages **usually** get

where they need to go, but there are no guarantees, and the sender usually doesn't even know if the data got to its destination.

These characteristics present serious problems to software. Many, if not most, applications need to be able to count on the fact that the data they send will get to its destination without loss or error. They also want the connection between two devices to be automatically managed, with problems such as congestion and flow control taken care of as needed. Unless some mechanism is provided for this at lower layers, every application would need to perform these jobs, which would be a massive duplication of effort.

In fact, one might argue that establishing connections, providing reliability, and handling retransmissions, buffering and data flow is sufficiently important that it would have been best to simply build these abilities directly into the Internet Protocol. Interestingly, that was exactly the case in the early days of TCP/IP. “In the beginning” [there was just a single protocol](#) called “TCP” that combined the tasks of the Internet Protocol with the reliability and session management features just mentioned.

There's a big problem with this, however. Establishing connections, providing a mechanism for reliability, managing flow control and acknowledgments and retransmissions: these all come at a cost: time and bandwidth. Building all of these capabilities into a single protocol that spans layers three and four would mean all applications got the benefits of reliability, but also the costs. While this would be fine for many applications, there are others that both don't need the reliability, and “can't afford” the overhead required to provide it.

### ***The Solution: Two Very Different Transport Protocols***


Fixing this problem was simple: let the network layer (IP) take care of basic data movement on the internetwork, and define two protocols at the transport layer. One would provide a rich set of services for those applications that need that functionality, with the understanding that some overhead was required to accomplish it. The other would be simple, providing little in the way of classic layer-four functions, but it would be fast and easy to use. Thus, the result of two TCP/IP transport-layer protocols:

- **Transmission Control Protocol (TCP):** [A full-featured, connection-oriented, reliable transport protocol for TCP/IP applications.](#) TCP provides transport-layer addressing to allow multiple software applications to simultaneously use a single IP address. It allows a pair of devices to establish a virtual connection and then pass data bidirectionally. Transmissions are managed using a special *sliding window* system, with unacknowledged transmissions detected and automatically retransmitted.

Additional functionality allows the flow of data between devices to be managed, and special circumstances to be addressed.

- **User Datagram Protocol (UDP):** [A very simple transport protocol that provides transport-layer addressing like TCP, but little else.](#) UDP is barely more than a “wrapper” protocol that provides a way for applications to access the Internet Protocol. No connection is established, transmissions are unreliable, and data can be lost.

By means of analogy, TCP is a fully-loaded luxury performance sedan with a chauffeur and a satellite tracking/navigation system. It provides lots of frills and comfort, and good performance. It virtually guarantees you will get where you need to go without any problems, and any concerns that do arise can be corrected. In contrast, UDP is a stripped-down race car. Its goal is simplicity and speed, speed, speed; everything else is secondary. You will probably get where you need to go, but hey, race cars can be finicky to keep operating.

 **Key Concept:** To suit the differing transport requirements of the many TCP/IP applications, two TCP/IP transport layer protocols exist. The *Transmission Control Protocol (TCP)* is a full-featured, connection-oriented protocol that provides acknowledged delivery of data while managing traffic flow and handling issues such as congestion and transmission loss. The *User Datagram Protocol (UDP)*, in contrast, is a much simpler protocol that concentrates only on delivering data, to maximize the speed of communication when the features of TCP are not required.

### ***Applications of TCP and UDP***

Having two transport layer protocols with such complementary strengths and weaknesses provides considerable flexibility to the creators of networking software:

- **TCP Applications:** Most “typical” applications need the reliability and other services provided by TCP, and don't care about loss of a small amount of performance to overhead. For example, most applications that transfer files or important data between machines use TCP, because loss of any portion of the file renders the entire thing useless. Examples include such well-known applications as the [Hypertext Transfer Protocol \(HTTP\) used by the World Wide Web \(WWW\)](#), the [File Transfer Protocol \(FTP\)](#) and the [Simple Mail Transfer Protocol \(SMTP\)](#). [I describe TCP applications in more detail in the TCP section.](#)
- **UDP Applications:** I'm sure you're thinking: “what sort of application doesn't care if its data gets there, and why would I want to use it?” You

might be surprised: UDP is used by lots of TCP/IP protocols. UDP is a good match for applications in two circumstances. The first is when the application doesn't really care if some of the data gets lost; streaming video or multimedia is a good example, since one lost byte of data won't even be noticed. The other is when the application itself chooses to provide some other mechanism to make up for the lack of functionality in UDP. Applications that send very small amounts of data, for example, often use UDP under the assumption that if a request is sent and a reply is not received, the client will just send a new request later on. This provides enough reliability without the overhead of a TCP connection. [I discuss some common UDP applications in the UDP section.](#)



**Key Concept:** Most classical applications, especially ones that send files or messages, require that data be delivered reliably, and therefore use TCP for transport. Applications using UDP are usually those where loss of a small amount of data is not a concern, or that use their own application-specific procedures for dealing with potential delivery problems that TCP handles more generally.

In the next few sections we'll first examine the common transport layer addressing scheme used by TCP and UDP, and then look at each of the two protocols in detail. Following these sections is a [summary comparison](#) to help you see at a glance where the differences lie between TCP and UDP. Incidentally, if you want a good “real-world” illustration of why having both UDP and TCP is valuable, consider [message transport under the Domain Name System \(DNS\)](#), which actually uses UDP for certain types of communication and TCP for others.

Before leaving the subject of comparing UDP and TCP, I want to explicitly point out that even though TCP is often described as being *slower* than UDP, this is a **relative** measurement. TCP is a very well-written protocol that is capable of highly efficient data transfers. It is only slow compared to UDP because of the overhead of establishing and managing connections. The difference can be significant, but is not enormous, so keep that in mind.

**Read TechNote: [The Transmission Control Protocol](#)**