- TCP and UDP -

Transport Layer Protocols

The **Transport layer (OSI Layer-4)** does *not* actually transport data, despite its name. Instead, this layer is responsible for the *reliable* transfer of data, by ensuring that data arrives at its destination error-free and in order.

The Transport layer is referred to as the **Host-to-Host layer** in the Department of Defense (DoD) reference model.

Transport layer communication falls under two categories:

- **Connection-oriented** – requires that a connection with specific agreed-upon parameters be established before data is sent.
- **Connectionless** – requires no connection before data is sent.

Connection-oriented protocols provide several important services:

- **Connection establishment –** connections are established, maintained, and ultimately terminated between devices.
- **Segmentation and sequencing –** data is *segmented* into smaller pieces for transport. Each segment is assigned a *sequence number*, so that the receiving device can reassemble the data on arrival.
- **Acknowledgments –** receipt of data is confirmed through the use of *acknowledgments*. If a segment is lost, data can be retransmitted to guarantee delivery.
- **Flow control** (or **windowing) –** data transfer rate is negotiated to prevent congestion.

The TCP/IP protocol suite incorporates two Transport layer protocols:

- **Transmission Control Protocol (TCP) –** connection-oriented
- **User Datagram Protocol (UDP) -** connectionless

Both TCP and UDP provide a mechanism to differentiate applications running on the same host, through the use of **port numbers.** When a host receives a packet, the port number tells the transport layer which higher-layer application to hand the packet off to.

Both TCP and UDP will be covered in detail in this guide. Please note that the *best* resource on the Internet for TCP/UDP information is the exemplary TCP/IP Guide, found here: http://www.tcpipguide.com/free/index.htm

Port Numbers and Sockets

Both TCP and UDP provide a mechanism to differentiate applications (or **services**) running on the same host, through the use of **port numbers.** When a host receives a segment, the port number tells the transport layer which higher-layer application to hand the packet off to. This allows multiple network services to operate simultaneously on the same logical address, such as a web *and* an email server.

The range for port numbers is **0 – 65535**, for both TCP and UDP.

The combination of the IP address and port number (identifying both the *host* and *service)* is referred to as a **socket**, and is written out as follows:

192.168.60.125:443

Note the colon separating the IP address (*192.168.60.125*) from the port number (*443*).

The first 1024 ports (**0-1023**) have been reserved for widely-used services, and are recognized as **well-known** ports. Below is a table of several common TCP/UDP ports:

Port Number	**Transport Protocol**	**Application**
20, 21	TCP	FTP
22	TCP	SSH
23	TCP	Telnet
25	TCP	SMTP
53	UDP or TCP	DNS
80	TCP	HTTP
110	TCP	POP3
443	TCP	SSL
666	TCP	Doom

Ports ranging from **1024 – 49151** are referred to as **registered ports**, and are allocated by the IANA upon request. Ports ranging from **49152 – 65535** cannot be registered, and are considered **dynamic.** A client *initiating* a connection will randomly choose a port in this range as its *source* port (for some operating systems, the dynamic range starts at *1024* and higher).

For a complete list of assigned port numbers, refer to the IANA website:

http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml

Transmission Control Protocol (TCP)

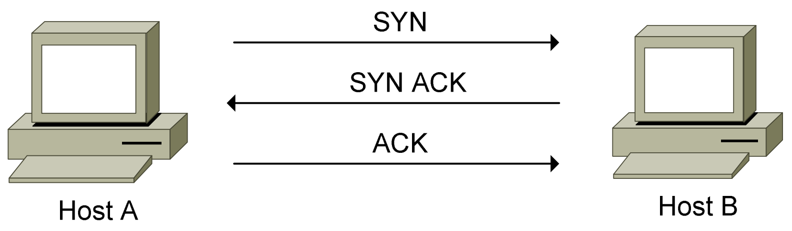
The **Transmission Control Protocol (TCP)** is a **connection-oriented** transport protocol, providing reliable delivery over an Internet Protocol (IP) network. Together, TCP and IP provide the core functionality for the **TCP/IP** or **Internet protocol suite**.

TCP was originally defined in RFC 675, and initially designed to perform *both* Network and Transport layer functions. When this proved to be an inflexible solution, those functions were separated - with IP providing Network layer services, and TCP providing Transport layer services. This separation was formalized in **version 4** of TCP, defined in RFC 793.

Because TCP is connection-oriented, parameters must be agreed upon by both the sending and receiving devices before a connection is established.

Establishing a TCP Connection

TCP employs a **three-way handshake** to form a connection. Control messages are passed between the two hosts as the connection is set up:

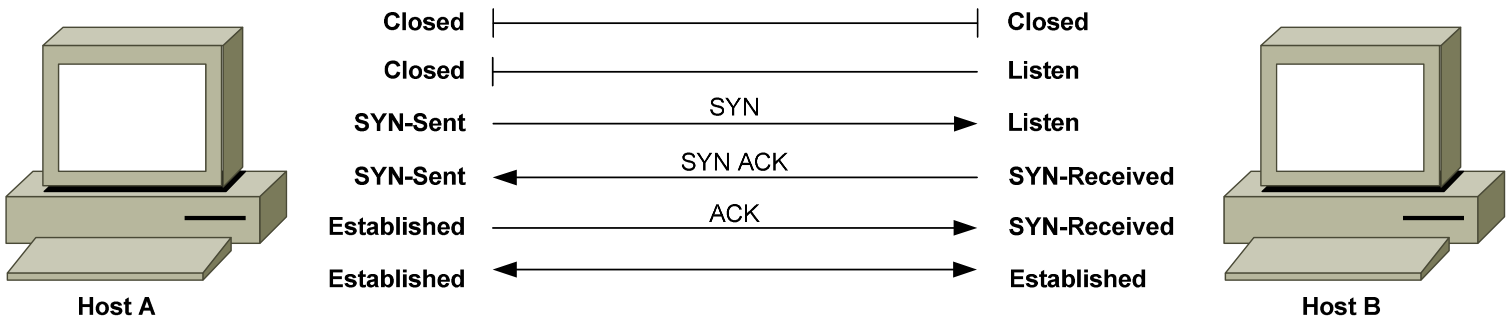


- HostA sends a **SYN** (short for **synchronize)** message to HostB to initiate a connection.
- HostB responds with an **ACK** (short for **acknowledgement)** to HostA's SYN message, and sends its own **SYN** message. The two messages are combined to form a single **SYN+ACK** message.
- HostA completes the three-way handshake by sending an **ACK** to HostB's SYN.

The TCP header contains six different **flags,** including a SYN flag and an ACK flag. Thus, when a particular *type* of message needs to be sent, the appropriate flag is marked as **on,** or changed from a *0* to a *1*. A SYN+ACK message has both flags set to on (*1*).

Establishing a TCP Connection (continued)

As the three-way handshake occurs, the sending and receiving hosts will pass through several **states:**



A **closed** state indicates a complete absence of a TCP connection.

Before a host can accept a request for a TCP connection, the host must enter a **listen** state, also known as a ***passive* open**. For example, a web server will passively *listen* on the HTTP port, waiting for incoming connection requests. A host must listen on each port it wishes to accept connections on.

A host will enter a **SYN-sent** state once it sends a **SYN message** to initiate a connection, also known as an ***active* open**. The sending host will remain in this state as it waits for the remote host's **ACK message.**

The receiving host will respond to the SYN message with a **SYN+ACK message,** and enter a **SYN-received** state**.**

The sending host will respond to the SYN+ACK message with its own **ACK message** and enter an **Established** state. The receiving host will enter an Established state once it receives this final ACK.

An Established state indicates that data transfer can occur. The communication becomes **bidirectional**, regardless of which host initiated the connection.

TCP can support many simultaneous connections, and must track and maintain each connection individually. Connections are identified by the *sockets* of both the source and destination host, and data specific to each connection is maintained in a **Transmission Control Block (TCB).**

(Reference: http://www.tcpipguide.com/free/t_TCPConnectionEstablishmentProcessTheThreeWayHandsh-3.htm)

TCP Segmentation and Sequencing

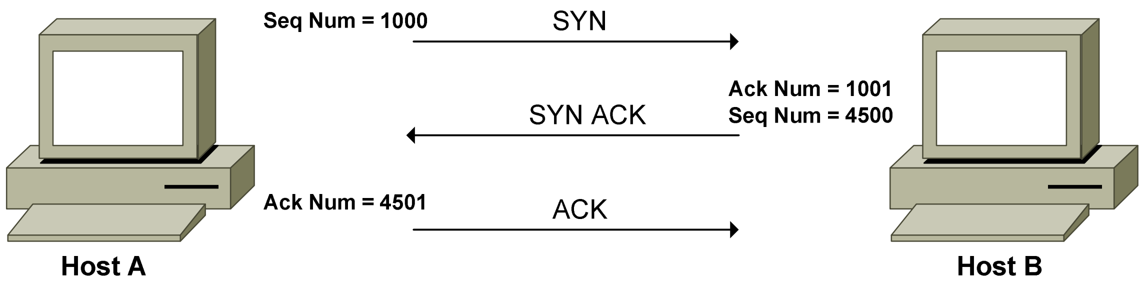
TCP is a **stream-oriented** transport protocol. This allows the application layer to send a continuous stream of unstructured data and rely on TCP to package the data as **segments**, regardless of the amount of data.

TCP will not only segment data into smaller pieces for transport, but will also assign a **sequence number** to each segment. Note though that this sequence number identifies the data (bytes) *within* the segment rather than the segment *itself.*

Sequencing serves two critical purposes:

- It allows the receiving host to reassemble the data from multiple segments in the correct order, upon arrival.
- It allows receipt of data within a segment to be *acknowledged,* thus providing a mechanism for dropped segments to be detected and resent.

When establishing a connection, a host will choose a 32-bit **initial sequence number (ISN).** The ISN is chosen from a randomizing timer, to prevent accidental overlap or predictability.

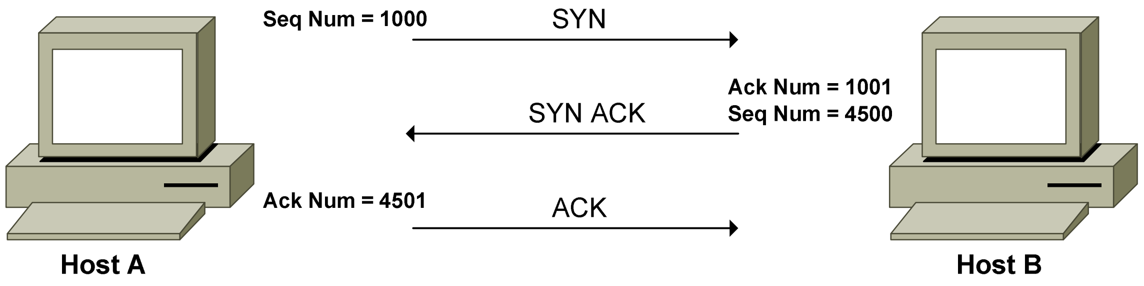


The receiving host responds to this sequence number with an **acknowledgment number**, set to the **sequence number + 1**. In the above example, HostB's acknowledgment number would thus be *1001*.

HostB includes an initial sequence number with *its* SYN message as well – *4500* in the above example. HostA would respond to this sequence number with an acknowledgement number of *4501.*

The TCP header contains both a 32-bit Sequence Number and 32-bit Acknowledgement Number field.

TCP Sliding Window



Once the TCP connection is established, the sequence numbers are used to identify the *data* within the segment. Using the above example again, HostA's first byte of data will be assigned a sequence number *1001*. Note that this is HostB's acknowledgment number, which essentially identifies which byte the receiving host is expecting *next*. HostB's first byte of data will be assigned a sequence number of *4501*.

Note that each *individual* byte of data **is not** assigned a sequence number and acknowledged independently, as this would introduce massive overhead. Instead, data is sequenced and acknowledged in *groups*, dictated by the **TCP window size.** The window size can never exceed the **maximum segment size (MSS)**, which is **536 bytes** by default.

The TCP window size is dictating by the *receiving host*, and informs the sender how many bytes it is permitted to send, before waiting for an acknowledgement. This window size can be dynamically changed to provide a measure of **flow control,** preventing buffer congestion on the receiving host.

A window size of **0** would instruct the sender to send no further data, usually indicating significant congestion on the receiving host.

TCP employs a ***sliding* window mechanism**. Bytes in a sliding window fall into one of four categories:

- Bytes that have *already* been sent *and* acknowledged.
- Bytes that have been sent, but *not acknowledged.*
- Bytes that have *not yet been sent*, but the receiving host *is ready for*.
- Bytes that have *not yet been sent*, and the receiving host *is not ready for.*

(Reference: http://www.tcpipguide.com/free/t_TCPSlidingWindowAcknowledgmentSystemForDataTranspo-5.htm; http://docwiki.cisco.com/wiki/Internet_Protocols#Transmission_Control_Protocol_.28TCP.29)

TCP Sliding Window (continued)

Consider the following conceptual example:

	Byte #	***Category***
	1-50	Bytes sent and acknowledged
TCP Window	51-75	Bytes sent and not yet acknowledged
	76-100	Bytes not sent, receiving host is ready
	101-200	Bytes not sent, receiving host is *not ready*

Several determinations can be made:

- The TCP stream is 200 bytes total.
- The TCP window size is 50 bytes total.
- The sending host can immediately send another 25 bytes of data (bytes 76-100)

Once bytes 51-75 are acknowledged, and bytes 76-100 are sent, the window will *slide* down*:*

	Byte #	***Category***
	1-75	Bytes sent and acknowledged
TCP Window	76-100	Bytes sent and not yet acknowledged
	101-125	Bytes not sent, receiving host is ready
	126-200	Bytes not sent, receiving host is *not ready*

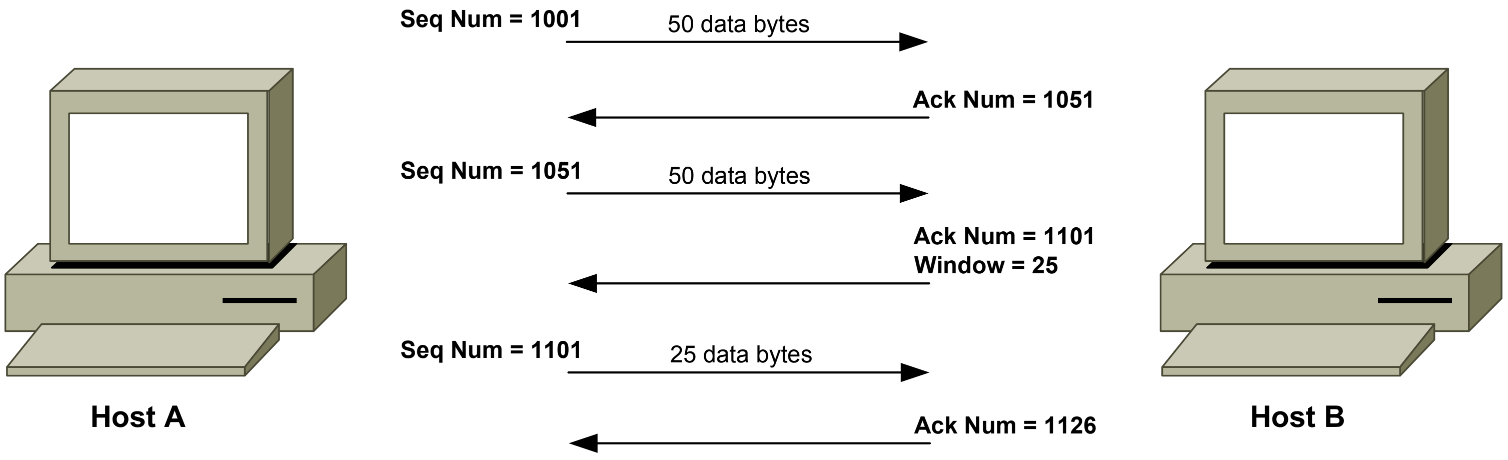
This assumes that that TCP window stays at 50 bytes. Remember that the window size is dictated by the receiving host (in a 16-bit **Window field** in the TCP header), and can be dynamically adjusted.

For efficiency, TCP will generally wait to send a segment until the agreed-upon TCP window size is full. This may not be acceptable for certain types of applications, which may not tolerate this latency.

The TCP header provides a **PSH (Push) flag** to accommodate this, allowing data to be sent *immediately*, regardless if the TCP window has been filled. The PSH flag can be used in conjunction with the **URG (Urgent) flag,** which allows specified data to be prioritized over other data. The URG flag must be used with the **Urgent Pointer field**, which identifies the *last* byte of urgent data, to identify where non-urgent data begins in a segment.

TCP Sliding Window (continued)

How do sequence and acknowledgement numbers fit within the sliding window concept? Consider the following *very* basic example, and assume the TCP connection is already established:



Recall that during the setup of a TCP connection, the acknowledgement number was set to the sequence number + 1. However, during data transfer, the acknowledgement number is used to acknowledge receipt of a *group* of data bytes.

In the above example, the initial TCP window size is set to *50* bytes, and the first byte in the stream is assigned a sequence number of *1001*. HostB acknowledges receipt of these 50 data bytes with an acknowledgement number of *1051* (for the mathematically disinclined, this is 1001 + 50). ☺

Once acknowledged, HostA then sends another 50 bytes of data, identifying the first byte with a sequence number of 1051. HostB acknowledges receipt again, with an ACK number of 1101. However, HostB also adjusts the TCP window size to 25 bytes, perhaps due to congestion.

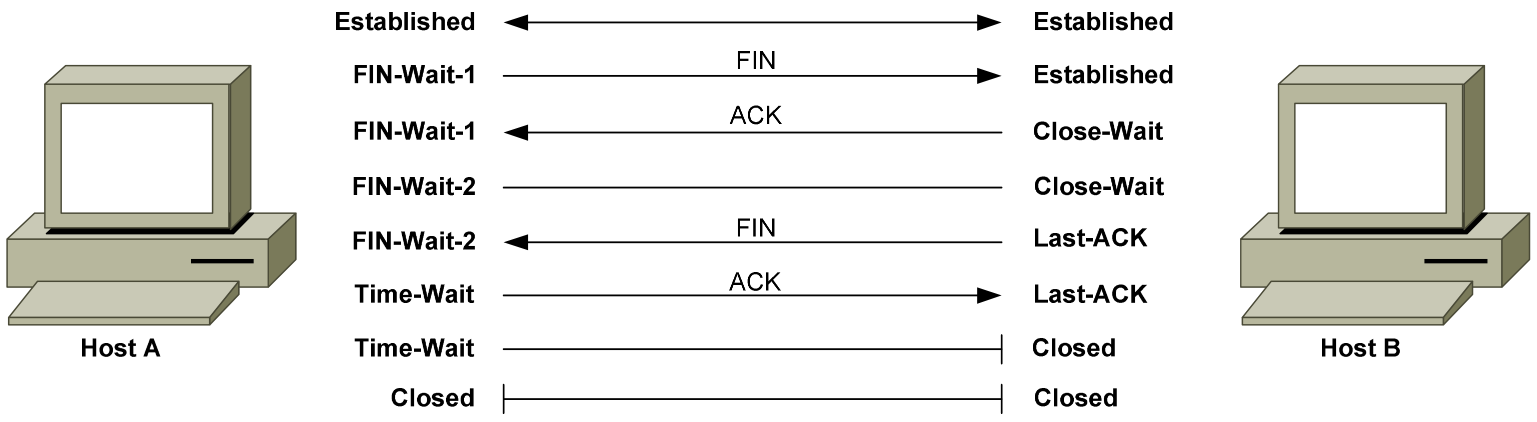
HostA's next segment will thus only contain 25 bytes of data, with a sequence number of 1101. HostB acknowledges these 25 bytes with an ACK number of 1126.

Every time a segment is sent, the sending host starts a **retransmission timer,** dynamically determined (and adjusted) based on the round-trip time between the two hosts. If an acknowledgement is not received before the retransmission timer expires, the segment is resent. This allows TCP to guarantee delivery, even when segments are lost.

Gracefully Terminating a TCP Connection

A TCP connection will remain established until it is purposely **terminated** by either host. The most common reason for connection termination is that both hosts have finished sending data. The termination process is handled separately by each host, allowing both hosts to fully complete data transfer before the connection is terminated.

Hosts can terminate an established TCP connection by sending a message with the **FIN (Finish) flag** set:



Once HostA sends the FIN message, it will enter a **FIN-Wait-1** state, waiting for the FIN to be acknowledged.

HostB responds to the FIN with an ACK message, and enters a **Close-Wait** state, allowing the local application to finish its processes. HostA receives the ACK and enters a **FIN-Wait-2** state, waiting for HostB to send a FIN message of its own, indicating it is safe to close the connection.

HostB sends a FIN message to HostA once the application process is complete, and enters a **Last-ACK** state.

HostA receives the FIN message and responds with an ACK message. HostA then enters a **Time-Wait** state, allowing time for the ACK to be received by HostB.

HostB receives the ACK message and enters a **Closed** state.

HostA's Time-Wait timer expires, and it also enters a Closed state. The connection is now gracefully terminated.

(Reference: http://www.tcpipguide.com/free/t_TCPConnectionTermination-2.htm)

Less than Graceful TCP Connection Termination

A TCP connection can become **half-open,** indicating that one host is an established state while the other is not. Half-open connections can result from **interruption** by an intermediary device (such as a firewall), or from a software or hardware issue.

TCP utilizes the **Reset message,** using the **RST flag**, to address half-open connections. Sending a RST message will force the remote host to reset the TCP connection and return to a *closed* state, or return to a passive *listen* state if the remote host is a server listening on that port.

There are a few scenarios in which a RST might be sent:

- A host receives a TCP segment from a host that it does not have a connection with.
- A host receives a segment with an incorrect sequence or acknowledgement number.
- A host receives a SYN request on a port it is not listening on.

Note on half-open connections: A **SYN flood** is a common denial-of-service attack that sends a large number of TCP SYN messages to a host, while spoofing the source address. The host will respond with an equal number of SYN+ACK messages, and will wait for the final ACK message that never comes. This spawns a large amount of half-open connections, which can prevent the host from responding to legitimate requests.

Modern firewalls can detect SYN flood attacks and minimize the number of accepted half-open connections.

(Reference: http://www.tcpipguide.com/free/t_TCPConnectionManagementandProblemHandlingtheConnec.htm)

The TCP Header

The TCP header is comprised of **12 fields,** and has a minimum size of **160 bits (20 bytes)**:

Field	***Length***	***Description***
Source Port	16 bits	*Source TCP Port*
Destination Port	16 bits	*Destination TCP Port*
Sequence Number	32 bits	*Sequence Number*
Ack Number	32 bits	*Acknowledgement Number*
Data Offset	4 bits	*Indicates where the data begins in a TCP segment*
Reserved	6 bits	*Always set to 0*
Control Bits	6 bits	*URG, ACK, PSH, RST, SYN, and FIN flags*
Window	16 bits	*Used for Flow Control*
Checksum	16 bits	*Used for Error-Checking*
Urgent Pointer	16 bits	*Identifies last byte of Urgent traffic*
Options	Variable	
Padding	Variable	*To ensure the TCP header ends at a 32-bit boundary*

The 16-bit **Source Port field** identifies the application service on the *sending* host. The 16-bit **Destination Port field** identifies the application service on the *remote* host.

The 32-bit **Sequence Number field** is used both during connection establishment, and during data transfer. During connection establishment (SYN message), an *initial sequence number* is randomly chosen. Subsequently, sequence numbers are used to identify data bytes in a stream.

The 32-bit **Acknowledgement Number field**, as its name suggests, is used to acknowledge a sequence number. During connection setup, this is set to the sending host's initial sequence number + 1. During data transfer, this value is used to acknowledge receipt of a group of data bytes.

The 4-bit **Data Offset field** indicates where data begins in a TCP segment, by identifying the number of 32-bit multiples in the TCP header. A TCP header *must* end on a 32-bit boundary.

Following the data offset field is the 6-bit **Reserved** (for future use) **field,** which is always set to zeroes.

The TCP Header (continued)

Field	***Length***	***Description***
Source Port	16 bits	*Source TCP Port*
Destination Port	16 bits	*Destination TCP Port*
Sequence Number	32 bits	*Sequence Number*
Ack Number	32 bits	*Acknowledgement Number*
Data Offset	4 bits	*Indicates where the data begins in a TCP segment*
Reserved	6 bits	*Always set to 0*
Control Bits	6 bits	*URG, ACK, PSH, RST, SYN, and FIN flags*
Window	16 bits	*Used for Flow Control*
Checksum	16 bits	*Used for Error-Checking*
Urgent Pointer	16 bits	*Identifies last byte of Urgent traffic*
Options	Variable	
Padding	Variable	*To ensure the TCP header ends at a 32-bit boundary*

The 6-bit **Control Bits** field contains six 1-bit flags, in the following order:

- **URG (Urgent) –** prioritizes specified traffic.
- **ACK (Acknowledgment) –** acknowledges a SYN or receipt of data.
- **PSH (Push) –** forces an immediate send even if window is not full.
- **RST (Reset) –** forcefully terminates an improper connection.
- **SYN (Synchronize) –** initiates a connection.
- **FIN (Finish) –** gracefully terminates a connection when there is further data to send.

The 16-bit **Window field** identifies the number of data octets that the receiver is able to accept.

The 16-bit **Checksum field** is used for error-checking, and is computed using both the TCP segment and select fields from the IP header. The receiving host will discard the segment if it fails the checksum calculation.

The 16-bit **Urgent Pointer field** is used to identify the *last* byte of prioritized traffic in a segment, when the URG flag is set.

The variable-length **Options field** provides additional optional TCP parameters, outside the scope of this guide.

The variable-length **Padding field** ensures the TCP header ends on a 32-bit boundary, and is always set to zeroes.

User Datagram Protocol (UDP)

The **User Datagram Protocol (UDP)** is a **connectionless** transport protocol, and is defined in RFC 768.

UDP, above all, is *simple.* It provides no three-way handshake, no flow-control, no sequencing, and no acknowledgment of data receipt. UDP essentially forwards the segment and takes no further interest.

Thus, UDP is **inherently unreliable,** especially compared to a connection-oriented protocol like TCP. However, UDP **experiences less latency** than TCP, due to the reduced overhead. This makes UDP ideal for applications that require speed over reliability. For example, DNS primarily uses UDP as its transport protocol, though it supports TCP as well.

Like TCP, UDP does provide basic error-checking using a checksum, and uses port numbers to differentiate applications running on the same host.

The UDP header has only 4 **fields**:

Field	***Length***	***Description***
Source Port	16 bits	*Source UDP Port*
Destination Port	16 bits	*Destination UDP Port*
Length	16 bits	*Length of the header and the data*
Checksum	16 bits	*Used for Error-Checking*

The following provides a quick comparison of TCP and UDP:

TCP	***UDP***
Connection-oriented	Connectionless
Guarantees delivery	Does *not* guarantee delivery
Sends acknowledgments	Does *not* send acknowledgments
Reliable, but slower than UDP	Unreliable, but faster than TCP
Segments and sequences data	Does *not* provide sequencing
Resends dropped segments	Does *not* resend dropped segments
Provides flow control	Does *not* provide flow control
Performs CRC on data	Also performs CRC on data
Uses port numbers	Also uses port numbers