Chapter 3. Transport layer protocols: UDP and TCP

3.1 Ports and sockets

The most important and commonly used protocols of the transport layer include:

- User Datagram Protocol (UDP)
- Transmission Control Protocol (TCP)

By building on the functionality provided by the Internet Protocol (IP), the transport protocols deliver data to applications executing in the internet. This is done by making use of ports. The transport protocols can provide additional functionality such as congestion control, reliable data delivery, duplicate data suppression, and flow control as is done by TCP.

This section introduces the concepts of the port and socket, which are needed to determine which local process at a given host actually communicates with which process, at which remote host, using which protocol. If this sounds confusing, consider the following points:

- An application process is assigned a process identifier number (process ID), which is likely to be different each time that process is started.
- Process IDs differ between operating system platforms, thus they are not uniform.
- A server process can have multiple connections to multiple clients at a time, thus simple connection identifiers are not unique. The concept of ports and sockets provides a way to uniformly and uniquely identify connections and the programs and hosts that are engaged in them, irrespective of specific process IDs.

The concept of ports and sockets provides a way to uniformly and uniquely identify connections and the programs and hosts that are engaged in them, irrespective of specific process IDs.

3.1.1 Ports

Each process that wants to communicate with another process identifies itself to the TCP/IP protocol suite by one or more ports. A port is a 16-bit number used by the host-to-host protocol to identify to which higher-level protocol or application program (process) it must deliver incoming messages. There are two types of ports:

- **Well-known**: Well-known ports belong to standard servers, for example, Telnet uses port 23, http (www) uses port 80, SMTP (e-mail) uses port 25 etc. Well-known port numbers range between 1 and 1023 (prior to 1992, the range between 256 and 1023 was used for UNIX-specific servers). Well-known port numbers are typically odd, because early systems using the port concept required an odd/even pair of ports for duplex operations. Most servers require only a single port. Exceptions are the BOOTP server, which uses two: 67 and 68 and the FTP server, which uses two: 20 and 21. The well-known ports are controlled and assigned by the Internet Assigned Number Authority (IANA) and on most systems can only be used by system processes or by programs executed by privileged users. Well-known ports allow clients to find servers without configuration information.

- **Ephemeral**: Some clients do not need well-known port numbers because they initiate communication with servers, and the port number they are using is contained in the UDP/TCP datagrams sent to the server. Each client process is allocated a port number, for as long as it needs, by the host on which it is running. Ephemeral port numbers have values greater than 1023, normally in the range of 1024 to 65535. Ephemeral ports are not controlled by IANA and can be used by ordinary user-developed programs on most systems.

Confusion, due to two different applications trying to use the same port numbers on one host, is avoided by writing those applications to request an available port from TCP/IP. Because this port number is dynamically assigned, it can differ from one invocation of an application to the next. UDP and TCP use the same port principle. To the best possible extent, the same port numbers are used for the same services on top of UDP and TCP.

Note: Normally, a server will use either TCP or UDP, but there are exceptions. For example, domain name servers use both UDP port 53 and TCP port 53.

3.1.2 Sockets

The socket interface is one of several application programming interfaces to the communication protocols. Designed to be a generic communication programming interface, socket APIs were first introduced by 4.2 Berkeley Software Distribution (BSD). Although it has not been standardized, Berkeley socket API has become a de facto industry standard abstraction for network TCP/IP socket implementation.
Consider the following terminologies:

- A socket is a special type of file handle, which is used by a process to request network services from the operating system.
  A socket address is the triple:
  <protocol, local-address, local port>
  For example, in the TCP/IP (version 4) suite:
  <tcp, 192.168.14.234, 8080>
  A conversation is the communication link between two processes.
- An association is the 5-tuple that completely specifies the two processes that comprise a connection:
  <protocol, local-address, local-port, foreign-address, foreign-port>
  In the TCP/IP (version 4) suite, the following could be a valid association:
  <tcp, 192.168.14.234, 1500, 192.168.44, 22>
- A half-association is either one of the following, which each specify half of a connection:
  <protocol, local-address, local-process>

Or:
<protocol, foreign-address, foreign-process>
The half-association is also called a socket or a transport address. That is, a socket is an endpoint for communication that can be named and addressed in a network.

Two processes communicate through TCP sockets. The socket model provides a process with a full-duplex byte stream connection to another process. The application need not concern itself with the management of this stream; these facilities are provided by TCP.

TCP uses the same port principle as UDP to provide multiplexing. Like UDP, TCP uses well-known and ephemeral ports. Each side of a TCP connection has a socket that can be identified by the triple <TCP, IP address, port number>. If two processes are communicating over TCP, they have a logical connection that is uniquely identifiable by the two sockets involved, that is, by the combination <TCP, local IP address, local port, remote IP address, remote port>. Server processes are able to manage multiple conversations through a single port.

3.2 User Datagram Protocol (UDP)

UDP is a standard protocol and almost every implementation intended for small data units transfer or those which can afford to lose a little amount of data (such as multimedia streaming) will include UDP.

UDP is basically an application interface to IP. It adds no reliability, flow-control, or error recovery to IP. It simply serves as a multiplexer/demultiplexer for sending and receiving datagrams, using ports to direct the datagrams, as shown in Fig. 3.1.

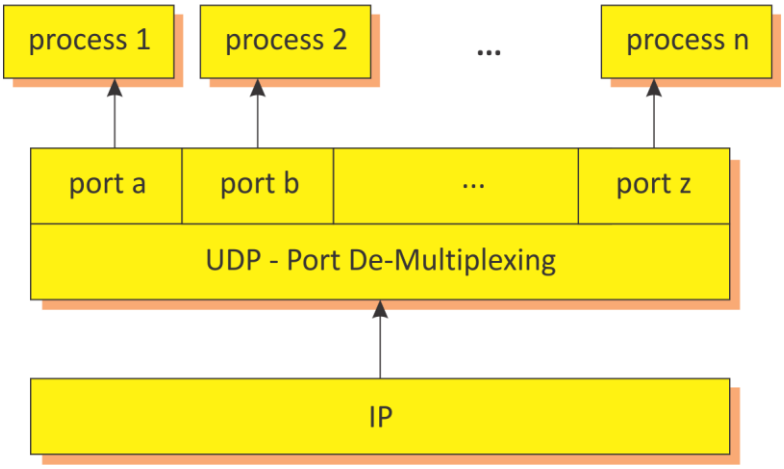


Fig. 3.1. UDP: Demultiplexing based on ports

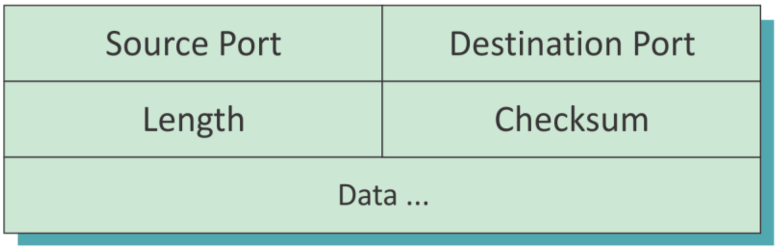
UDP provides a mechanism for one application to send a datagram to another. The UDP layer can be regarded as being extremely thin and is, consequently, very efficient, but it requires the application to take responsibility for error recovery and so on.

Applications sending datagrams to a host need to identify a target that is more specific than the IP address, because datagrams are normally directed to certain processes/application and not to the system as a whole. UDP provides this by using ports.

3.2.1 UDP datagram format

Each UDP datagram is sent within a single IP datagram. Although, the IP datagram might be fragmented during transmission, the receiving IP implementation will reassemble it before presenting it to the UDP layer. All IP implementations are required to accept datagrams of 576 bytes, which means that, allowing for maximum-size IP header of 60 bytes, a UDP datagram of 516 bytes is acceptable to all implementations. Many implementations will accept larger datagrams, but this is not guaranteed.

The UDP datagram has an 8-byte header, as described in Fig. 3.2.



Source Port	Destination Port
Length	Checksum
Data ...	

Fig. 3.2. UDP Datagram format

Where:

Source Port Indicates the port of the sending process. It is the port to which replies are addressed.

Destination Port Specifies the port of the destination process on the destination host.

Length The length (in bytes) of this user datagram, including the header.

Checksum An optional 16-bit one's complement of the one's complement sum of a pseudo-IP header, the UDP header, and the UDP data. In Figure 4-3, we see a pseudo-IP header. It contains the source and destination IP addresses, the protocol, and the UDP length.

3.2.2 UDP application programming interface

The way this interface is implemented is left to the discretion of each vendor. Be aware that UDP and IP do not provide guaranteed delivery, flow-control, or error recovery, so these must be provided by the application.

Standard applications using UDP include:

- Trivial File Transfer Protocol (TFTP)
- Domain Name System (DNS) name server
- Remote Procedure Call (RPC), used by the Network File System (NFS)
- Simple Network Management Protocol (SNMP)
- Lightweight Directory Access Protocol (LDAP)

3.3 Transmission Control Protocol (TCP)

TCP is a standard protocol and in practice, every TCP/IP implementation that is not used exclusively for routing will include TCP.

TCP provides considerably more facilities for applications than UDP. Specifically, this includes error recovery, flow control, and reliability. TCP is a connection-oriented protocol, unlike UDP, which is connectionless. Most of the user application protocols, such as Telnet and FTP, use TCP. The two processes communicate with each other over a TCP connection (InterProcess Communication, or IPC), as shown in Figure 3.3. In the figure, processes 1 and 2 communicate over a TCP connection carried by IP datagrams.

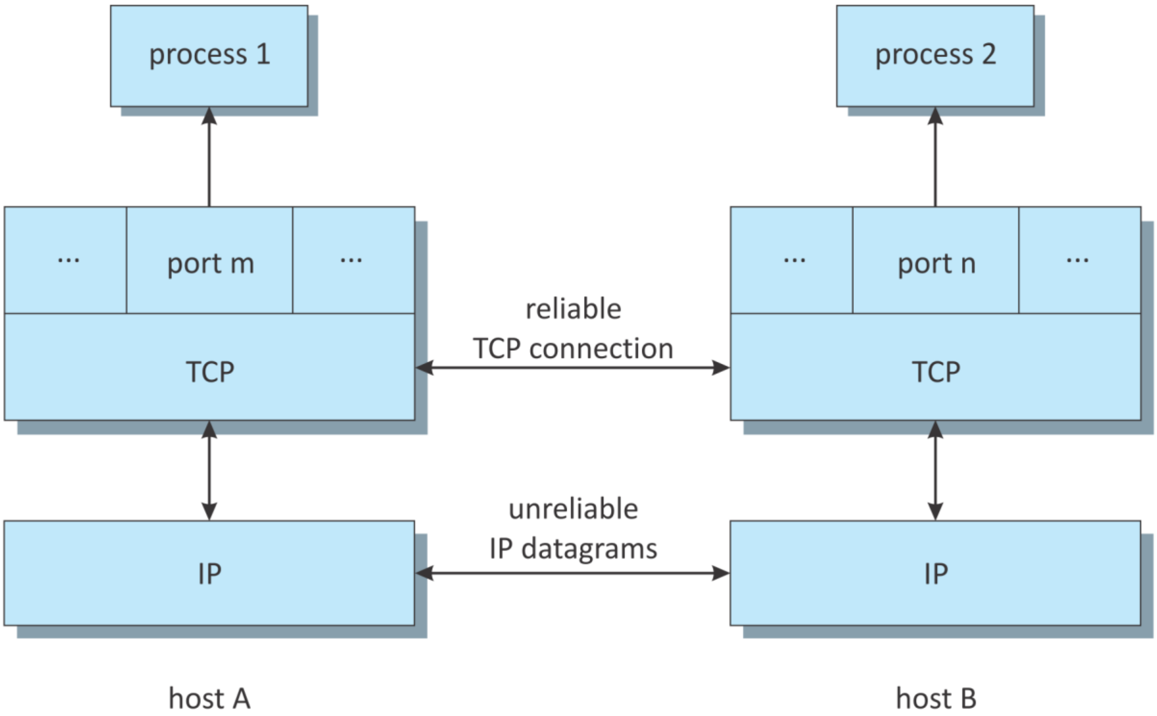


Fig. 3.3. TCP: Connection between processes

3.3.1 TCP concept

As noted earlier, the primary purpose of TCP is to provide a reliable logical circuit or connection service between pairs of processes. It does not assume reliability from the lower-level protocols (such as IP), so TCP must guarantee this itself.

TCP can be characterized by the following facilities it provides for the applications using it:

- **Stream data transfer**: From the application's viewpoint, TCP transfers a continuous stream of bytes through the network. The application does not have to bother with chopping the data into basic blocks or datagrams. TCP does this by grouping the bytes into TCP segments, which are passed to the IP layer for transmission to the destination. Also, TCP itself decides how to segment the data, and it can forward the data at its own convenience. Sometimes, an application needs to be sure that all the data passed to TCP has actually been transmitted to the destination. For that reason, a push function is defined. It will push all remaining TCP segments still in storage to the destination host. The normal close connection function also pushes the data to the destination.

- **Reliability**: TCP assigns a sequence number to each byte transmitted, and expects a positive acknowledgment (ACK) from the receiving TCP layer. If the ACK is not received within a timeout interval, the data is retransmitted. Because the data is transmitted in blocks (TCP segments), only the sequence number of the first data byte in the segment is sent to the destination host. The receiving TCP uses the sequence numbers to rearrange the segments when they arrive out of order, and to eliminate duplicate segments.
- **Flow control**: The receiving TCP, when sending an ACK back to the sender, also indicates to the sender the number of bytes it can receive (beyond the last received TCP segment) without causing overrun and overflow in its internal buffers. This is sent in the ACK in the form of the highest sequence number it can receive without problems. This mechanism is also referred to as a window-mechanism.
- **Multiplexing**: Achieved through the use of ports, just as with UDP.
- **Logical connections**: The reliability and flow control mechanisms described here require that TCP initializes and maintains certain status information for each data stream. The combination of this status, including sockets, sequence numbers, and window sizes, is called a logical connection. Each connection is uniquely identified by the pair of sockets used by the sending and receiving processes.
- **Full duplex**: TCP provides for concurrent data streams in both directions.

3.3.2 The window principle

A simple transport protocol might use the following principle: send a packet and then wait for an acknowledgment from the receiver before sending the next packet. If the ACK is not received within a certain amount of time, retransmit the packet. See Figure 3.4. for more details.

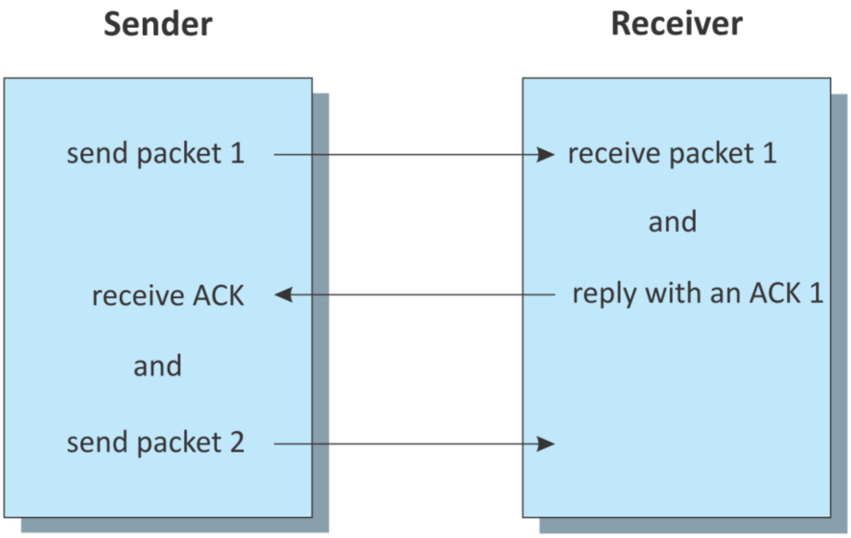


Fig. 3.4. TCP: The window principle

Now, consider a protocol where the sender groups its packets to be transmitted, as in Fig. 3.5, and uses the following rules:

- The sender can send all packets within the window without receiving an ACK, but must start a timeout timer for each of them.
- The receiver must acknowledge each packet received, indicating the sequence number of the last well-received packet.
- The sender slides the window on each ACK received.

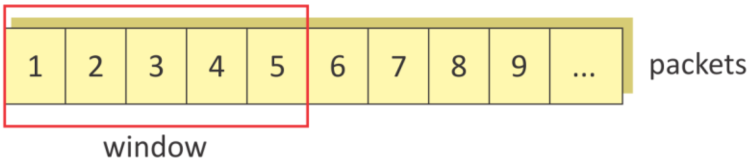


Fig. 3.5. Window principle (1)

As shown in Fig.3.6, the sender can transmit packets 1 to 5 without waiting for any acknowledgment.

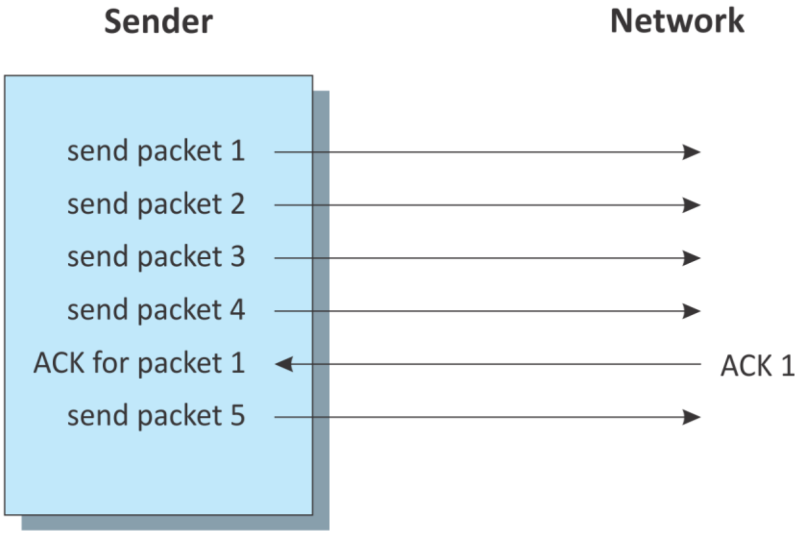


Fig. 3.6 Window principle (2)

As shown in Fig. 3.7, at the moment the sender receives ACK 1 acknowledgment for packet 1), it can slide its window one packet to the right.

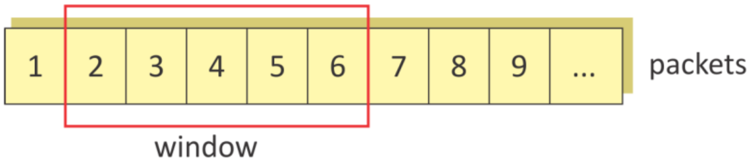


Fig. 3.7 Window principle (3)

At this point, the sender can also transmit packet 6.

Imagine some special cases:

- Packet 2 gets lost: The sender will not receive ACK 2, so its window will remain in position 1. In fact, because the receiver did not receive packet 2, it will acknowledge packets 3, 4, and 5 with an ACK 1, because packet 1 was the last one received in sequence. At the sender's side, eventually a timeout will occur for packet 2 and it will be retransmitted. Note that reception of this packet by the receiver will generate ACK 5, because it has now successfully received all packets 1 to 5, and the sender's window will slide four positions upon receiving this ACK 5.
- Packet 2 did arrive, but the acknowledgment gets lost: The sender does not receive ACK 2, but will receive ACK 3. ACK 3 is an acknowledgment for all packets up to 3 (including packet 2) and the sender can now slide its window to packet 4.

This window mechanism ensures:

- Reliable transmission.
- Better use of the network bandwidth (better throughput).
- Flow-control, because the receiver can delay replying to a packet with an acknowledgment, knowing its free buffers are available and the window size of the communication.

3.3.3 The window principle applied to TCP

The previously discussed window principle is used in TCP, but with a few differences:

- Because TCP provides a byte-stream connection, sequence numbers are assigned to each byte in the stream. TCP divides this contiguous byte stream into TCP segments to transmit them. The window principle is used at the byte level, that is, the segments sent and ACKs received will carry byte-sequence numbers and the window size is expressed as a number of bytes, rather than a number of packets.
- The window size is determined by the receiver when the connection is established and is variable during the data transfer. Each ACK message will include the window size that the receiver is ready to deal with at that particular time.

The sender's data stream can now be seen as follows in Fig. 3.8.

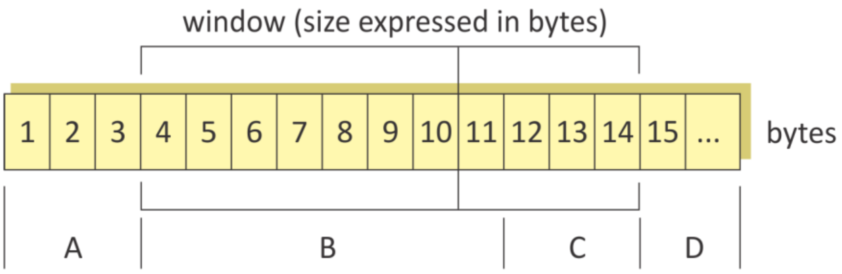


Fig. 3.8. Window principle applied to TCP

Where:

A Bytes that are transmitted and have been acknowledged

B Bytes that are sent but not yet acknowledged

C Bytes that can be sent without waiting for any acknowledgment
D Bytes that cannot be sent yet

Remember that TCP will block bytes into segments, and a TCP segment only carries the sequence number of the first byte in the segment.

3.3.4 TCP segment format

Fig. 3.9 shows the TCP segment format.

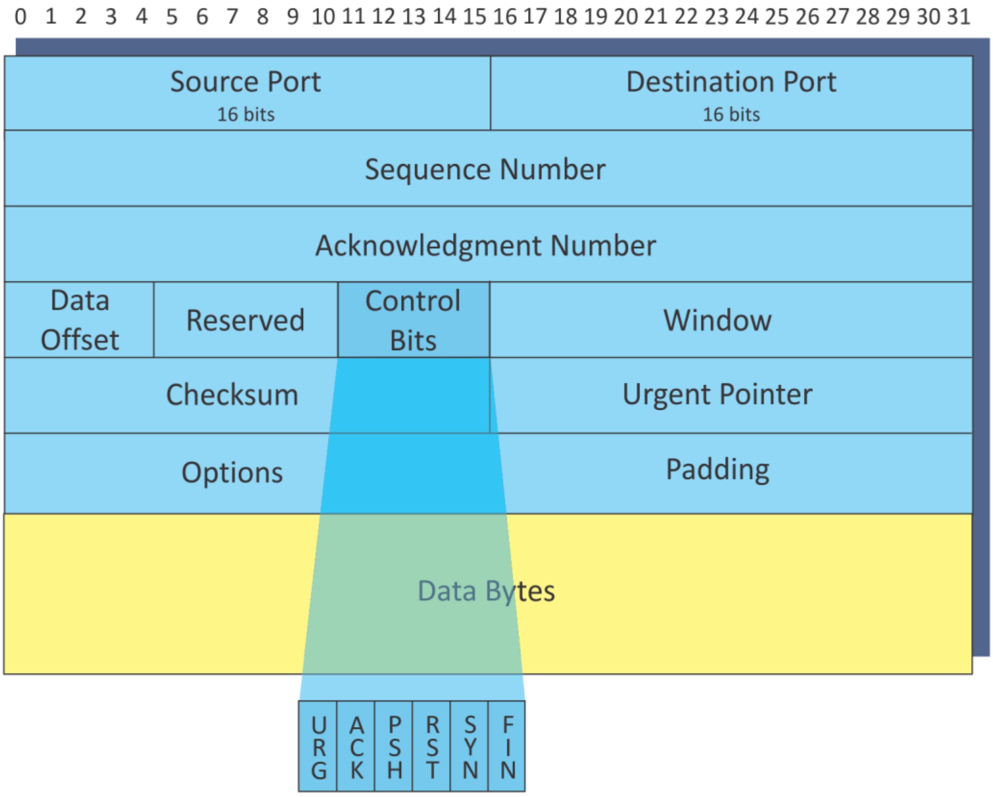


Fig. 3.9. TCP Segment format

Where:
Source Port The 16-bit source port number, used by the receiver to reply.
Destination Port The 16-bit destination port number.
Sequence Number The sequence number of the first data byte in thissegment. If the SYN control bit is set, the sequence number is the initial sequence number (n) and the first data byte is n+1.
Acknowledgment Number
If the ACK control bit is set, this field contains the value of the next sequence number that the receiver is expecting to receive.
Data Offset The number of 32-bit words in the TCP header. It indicates where the data begins.
Reserved Six bits reserved for future use; must be zero.
URG Indicates that the urgent pointer field is significant in this segment.
ACK Indicates that the acknowledgment field is significant in this segment.

PSH Push function.

RST Resets the connection.

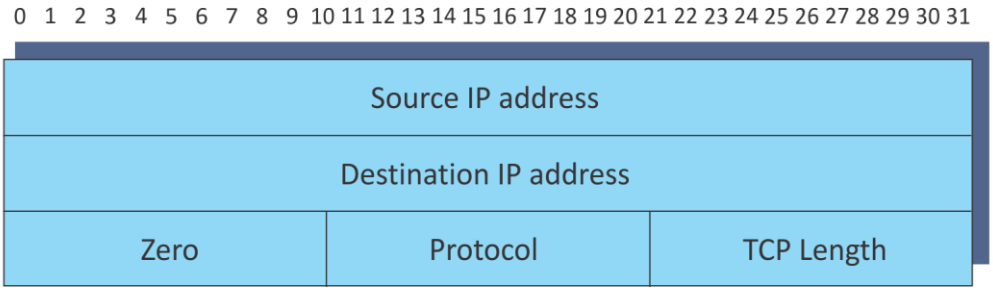
SYN Synchronizes the sequence numbers.

FIN No more data from sender.

Window Used in ACK segments. It specifies the number of data bytes, beginning with the one indicated in the acknowledgment number field that the receiver (the sender of this segment) is willing to accept.

Checksum The 16-bit one's complement of the one's complement sum of all 16-bit words in a pseudo-header, the TCP header, and the TCP data. While computing the checksum, the checksum field itself is considered zero.

The pseudo-header is the same as that used by UDP for calculating the checksum. It is a pseudo-IP-header, only used for the checksum calculation, with the format shown in Fig. 3.10.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Source IP address

Destination IP address

Zero | Protocol | TCP Length

Fig. 3.10. Pseudo-IP header

Urgent Pointer Points to the first data octet following the urgent data. Only significant when the URG control bit is set.

Options Just as in the case of IP datagram options, options can be either:

– A single byte containing the option number

– A variable length option

Maximum segment size option

This option is only used during the establishment of the connection (SYN control bit set) and is sent from the side that is to receive data to indicate the maximum segment length it can handle. If this option is not used, any segment size is allowed.

Window scale option

This option is not mandatory. Both sides must send the Window scale option in their SYN segments to enable windows scaling in their direction.

SACK-permitted option

This option is set when selective acknowledgment is used in that TCP connection.

Timestamps option

The timestamps option sends a time stamp value that indicates the current value of the time stamp clock of the TCP sending the option.

Padding All zero bytes are used to fill up the TCP header to a total length that is a multiple of 32 bits.

3.3.5 Acknowledgments and retransmissions

TCP sends data in variable length segments. Sequence numbers are based on a byte count. Acknowledgments specify the sequence number of the next byte that the receiver expects to receive.

Consider that a segment gets lost or corrupted. In this case, the receiver will acknowledge all further well-received segments with an acknowledgment referring to the first byte of the missing packet. The sender will stop transmitting when it has sent all the bytes in the window. Eventually, a timeout will occur and the missing segment will be retransmitted.

Fig. 3.11. illustrates and example where a window size of 1500 bytes and segments of 500 bytes are used.

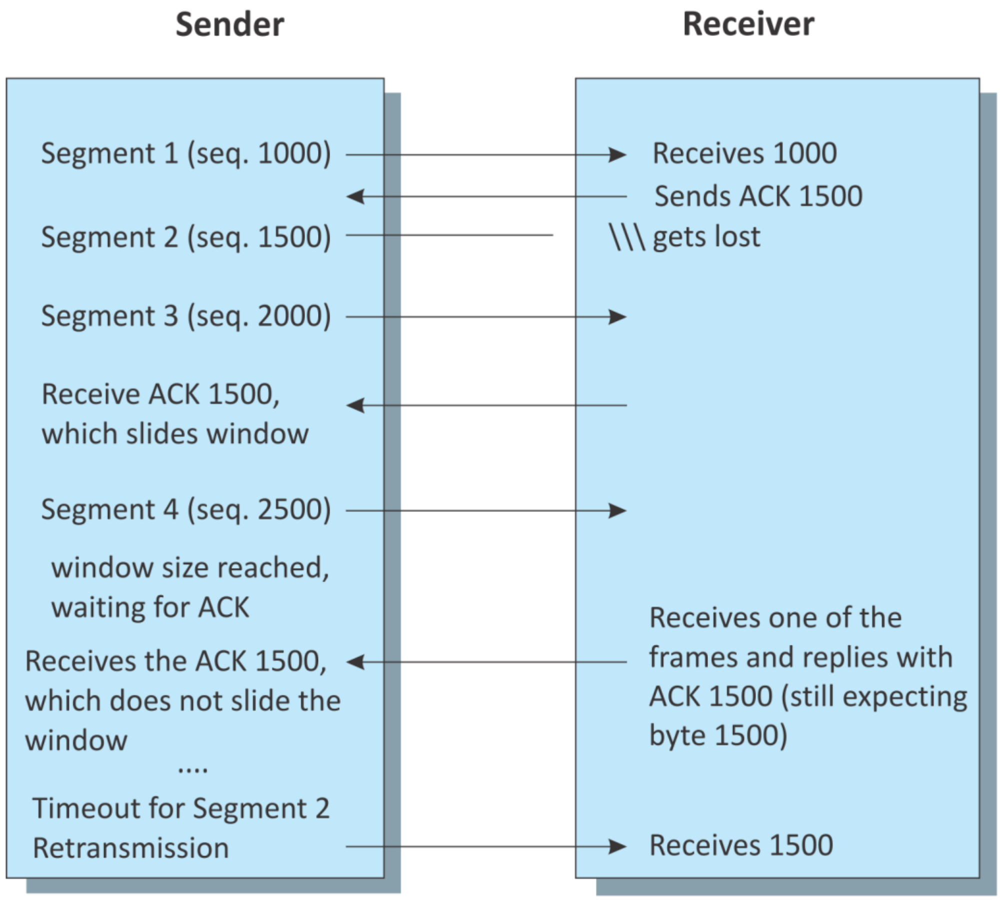


Fig. 3.11. TCP Acknowledgment and retransmission process

A problem now arises, because the sender does know that segment 2 is lost or corrupted, but does not know anything about segments 3 and 4. The sender should at least retransmit segment 2, but it could also retransmit segments 3 and 4 (because they are within the current window). It is possible that:

- Segment 3 has been received, and we do not know about segment 4. It might be received, but ACK did not reach us yet, or it might be lost.
- Segment 3 was lost, and we received the ACK 1500 on the reception of segment 4.

Each TCP implementation is free to react to a timeout as those implementing it want. It can retransmit only segment 2, but in the second case, we will be waiting again until segment 3 times out. In this case, we lose all of the throughput advantages of the window mechanism. Or TCP might immediately resend all of the segments in the current window.

Whatever the choice, maximal throughput is lost. This is because the ACK does not contain a second acknowledgment sequence number indicating the actual frame received.

3.3.6 Variable timeout intervals

Each TCP should implement an algorithm to adapt the timeout values to be used for the round trip time of the segments. To do this, TCP records the time at which a segment was sent, and the time at which the ACK is received. A weighted average is calculated over several of these round trip times, to be used as a timeout value for the next segment or segments to be sent.

This is an important feature, because delays can vary in IP network, depending on multiple factors, such as the load of an intermediate low-speed network or the saturation of an intermediate IP gateway.

3.3.7 Establishing a TCP connection

Before any data can be transferred, a connection has to be established between the two processes. One of the processes (usually the server) issues a passive OPEN call, the other an active OPEN call. The passive OPEN call remains dormant until another process tries to connect to it by an active OPEN.

As shown in Fig. 3.12., in the network, three TCP segments are exchanged.

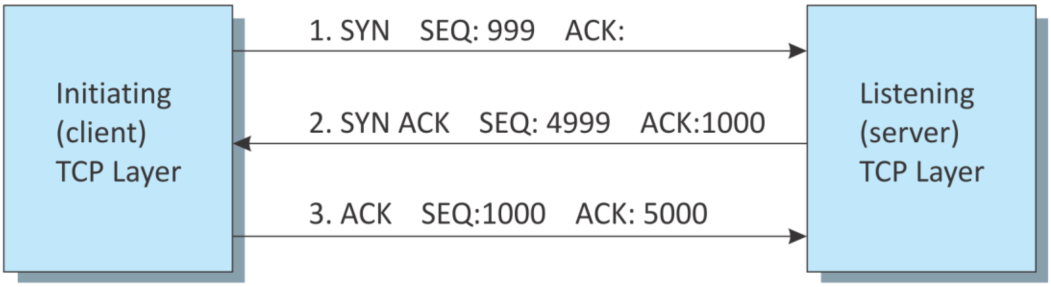


Fig. 3.12. TCP Connection establishment

This whole process is known as a three-way handshake. Note that the exchanged TCP segments include the initial sequence numbers from both sides, to be used on subsequent data transfers.

Closing the connection is done implicitly by sending a TCP segment with the FIN bit (no more data) set. Because the connection is full-duplex (that is, there are two independent data streams, one in each direction), the FIN segment only closes the data transfer in one direction. The other process will now send the remaining data it still has to transmit and also ends with a TCP segment where the FIN bit is set. The connection is deleted (status information on both sides) after the data stream is closed in both directions.

The following is a list of the different states of a TCP connection:

- LISTEN: Awaiting a connection request from another TCP layer.
- SYN-SENT: A SYN has been sent, and TCP is awaiting the response SYN.
- SYN-RECEIVED: A SYN has been received, a SYN has been sent, and TCP is awaiting an ACK.
- ESTABLISHED: The three-way handshake has been completed.
- FIN-WAIT-1: The local application has issued a CLOSE. TCP has sent a FIN, and is awaiting an ACK or a FIN.
- FIN-WAIT-2: A FIN has been sent, and an ACK received. TCP is awaiting a FIN from the remote TCP layer.
- CLOSE-WAIT: TCP has received a FIN, and has sent an ACK. It is awaiting a close request from the local application before sending a FIN.
- CLOSING: A FIN has been sent, a FIN has been received, and an ACK has been sent. TCP is awaiting an ACK for the FIN that was sent.
- LAST-ACK: A FIN has been received, and an ACK and a FIN have been sent. TCP is awaiting an ACK.

3.3.8 Congestion avoidance

The assumption of the algorithm is that packet loss caused by damage is very small (much less than 1%). Therefore, the loss of a packet signals congestion somewhere in the network between the source and destination. There are two indications of packet loss:

- A timeout occurs.
- Duplicate ACKs are received.

Congestion avoidance and slow start are independent algorithms with different objectives. But when congestion occurs, TCP must slow down its transmission rate of packets into the network and invoke slow start to get things going again. In practice, they are implemented together.

Congestion avoidance and slow start require that two variables be maintained for each connection:

- A congestion window, cwnd
- A slow start threshold size, ssthresh

The combined algorithm operates as follows:

1. Initialization for a given connection sets cwnd to one segment and ssthresh to 65535 bytes.

2. The TCP output routine never sends more than the lower value of cwnd or the receiver's advertised window.
3. When congestion occurs (timeout or duplicate ACK), one-half of the current window size is saved in ssthresh. Additionally, if the congestion is indicated by a timeout, cwnd is set to one segment.
4. When new data is acknowledged by the other end, increase cwnd, but the way it increases depends on whether TCP is performing slow start or congestion avoidance. If cwnd is less than or equal to ssthresh, TCP is in slow start; otherwise, TCP is performing congestion avoidance.

Slow start continues until TCP is halfway to where it was when congestion occurred (since it recorded half of the window size that caused the problem in step 2), and then congestion avoidance takes over. Slow start has cwnd begin at one segment, and incremented by one segment every time an ACK is received. As mentioned earlier, this opens the window exponentially: send one segment, then two, then four, and so on.

Congestion avoidance dictates that cwnd be incremented by segsize*segsize/cwnd each time an ACK is received, where segsize is the segment size and cwnd is maintained in bytes. This is a linear growth of cwnd, compared to slow start's exponential growth. The increase in cwnd should be at most one segment each round-trip time (regardless of how many ACKs are received in that round-trip time), while slow start increments cwnd by the number of ACKs received in a round-trip time. Many implementations incorrectly add a small fraction of the segment size (typically the segment size divided by 8) during congestion avoidance. This is wrong and should not be emulated in future releases.