
Chapter 5. Transport layer protocols

This chapter provides an overview of the most important and common protocols of the TCP/IP transport layer. These 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 IP host. This is done by making use of ports as described in 5.1, “Ports and sockets” on page 201. 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.

5.1 Ports and sockets

This section introduces the concepts of *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:

- 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, hence they are not uniform.
- A server process can have multiple connections to multiple clients at a time, hence simple connection identifiers would not be 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.

5.1.1 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 port:

- Well-known: Well-known ports belong to standard servers, for example, Telnet uses port 23. 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 (see 3.6, “Bootstrap protocol (BOOTP)” on page 121) and the FTP server, which uses two: 20 and 21 (see 10.1, “File Transfer Protocol (FTP)” on page 365).

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. The reason for well-known ports is to allow clients to be able to find servers without configuration information. The well-known port numbers are defined in STD 2 – Assigned Internet Numbers.

- Ephemeral: 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 datagrams sent to the server. Each client process is allocated a port number for as long as it needs it by the host it is running on. Ephemeral port numbers have values greater than 1023, normally in the range 1024 to 65535. A client can use any number allocated to it, as long as the combination of <transport protocol, IP address, port number> is unique.

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 may differ from one invocation of an application to the next.

UDP, TCP and ISO TP-4 all use the same port principle. To the best possible extent, the same port numbers are used for the same services on top of UDP, TCP, and ISO TP-4.

Note:

Normally, a server will use either TCP or UDP, but there are exceptions. For example, domain name servers (see 8.1, “Domain Name System (DNS)” on page 279) use both UDP port 53 and TCP port 53.

5.1.2 Sockets

The socket interface is one of several *application programming interfaces* (APIs) to the communication protocols. Designed to be a generic

communication programming interface, APIs were first introduced by 4.2 BSD. Although it has not been standardized, it has become a *de facto* industry standard.

4.2 BSD allowed two different communication domains: Internet and UNIX.

4.3 BSD has added the Xerox Network System (XNS) protocols, and 4.4 BSD will add an extended interface to support the ISO OSI protocols.

Let us 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-process>
For example, in the TCP/IP suite:
<tcp, 193.44.234.3, 12345>
- 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-process, foreign-address, foreign-process>
In the TCP/IP suite, the following could be a valid association:
<tcp, 193.44.234.3, 1500, 193.44.234.5, 21>
- A *half-association* is either:
<protocol, local-address, local-process>
or
<protocol, foreign-address, foreign-process>
which specify each half of a connection.
- 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 via *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. Please refer to 7.2.1, “The socket API” on page 262 for more information about socket APIs.

5.2 User Datagram Protocol (UDP)

UDP is a standard protocol with STD number 6. UDP is described by RFC 768 – User Datagram Protocol. Its status is recommended, but in practice every TCP/IP implementation that is not used exclusively for routing 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 Figure 90. For a more detailed discussion of ports, refer to 5.1, “Ports and sockets” on page 201.

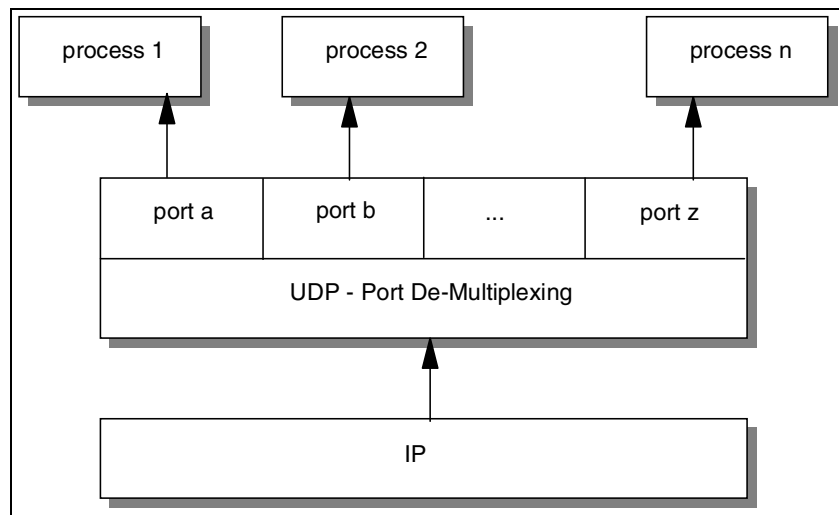


Figure 90. UDP - Demultiplexing based on ports

UDP provides a mechanism for one application to send a datagram to another. The UDP layer can be regarded as being extremely thin and consequently has low overheads, 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, since datagrams are normally directed to certain processes and not to the system as a whole. UDP provides this by using ports. The port concept is discussed in 5.1, “Ports and sockets” on page 201.

5.2.1 UDP datagram format

Each UDP datagram is sent within a single IP datagram. Although, the IP datagram may 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 a 16-byte header that is described in Figure 91.

Source Port	Destination Port
Length	Checksum
Data...	

Figure 91. UDP - Datagram format

Where:

- Source Port: Indicates the port of the sending process. It is the port to which replies should be 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. The pseudo-IP header contains the source and destination IP addresses, the protocol, and the UDP length:

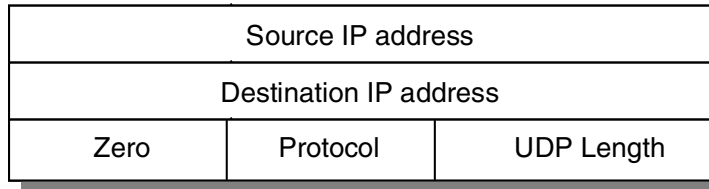


Figure 92. UDP - Pseudo-IP Header

The pseudo-IP header effectively extends the checksum to include the original (unfragmented) IP datagram.

5.2.2 UDP application programming interface

The application interface offered by UDP is described in RFC 768. It provides for:

- The creation of new receive ports.
- The receive operation that returns the data bytes and an indication of source port and source IP address.
- The send operation that has, as parameters, the data, source, and destination ports and addresses.

The way this interface should be 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)

5.3 Transmission Control Protocol (TCP)

TCP is a standard protocol with STD number 7. TCP is described by RFC 793 – Transmission Control Protocol. Its status is recommended, but 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, notably 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 - IPC), as shown in Figure 93. Please see 5.1, "Ports and sockets" on page 201 for more details about ports and sockets.

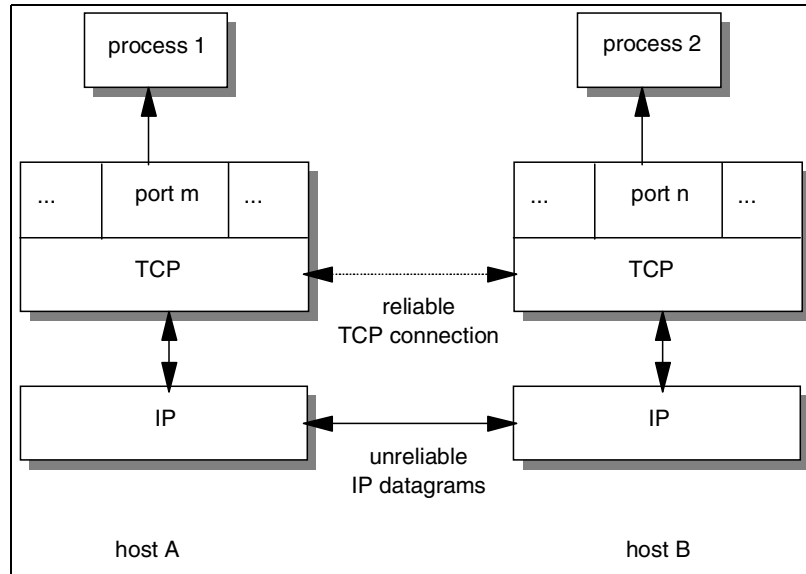


Figure 93. TCP - Connection between processes - (Processes 1 and 2 communicate over a TCP connection carried by IP datagrams.)

5.3.1 TCP concept

As noted above, the primary purpose of TCP is to provide 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 contiguous 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 in TCP segments, which are passed to IP

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:** CP assigns a sequence number to each byte transmitted and expects a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. Since 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, and we discuss it in more detail later in this chapter.
- **Multiplexing:** Achieved through the use of ports, just as with UDP.
- **Logical Connections:** The reliability and flow control mechanisms described above 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.

5.3.1.1 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 94 for more details.

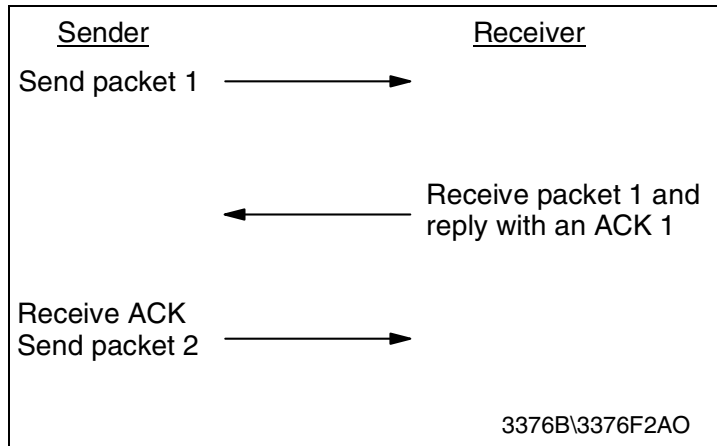


Figure 94. TCP - The window principle

While this mechanism ensures reliability, it only uses a part of the available network bandwidth.

Now, consider a protocol where the sender groups its packets to be transmitted, as in Figure 95, 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.

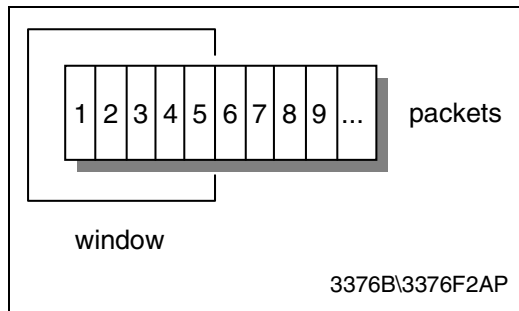


Figure 95. TCP - Message packets

In our example, the sender can transmit packets 1 to 5 without waiting for any acknowledgment:

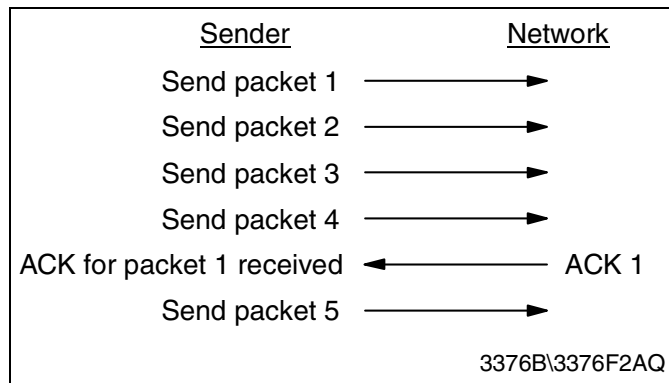


Figure 96. TCP - Window principle

At the moment the sender receives ACK 1 (acknowledgment for packet 1), it can slide its window one packet to the right:

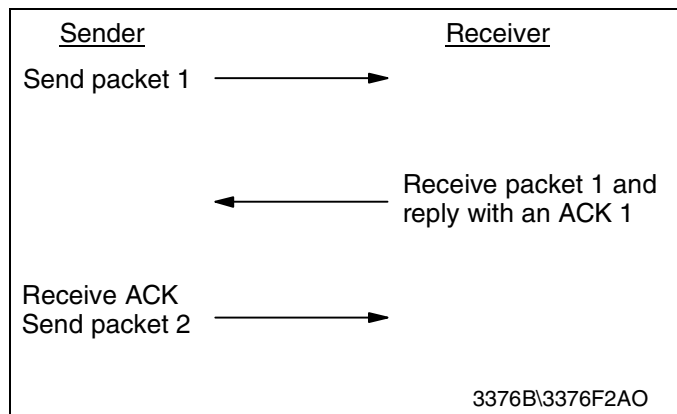


Figure 97. TCP - Message packets

At this point, the sender may 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 (as in Figure 97). In fact, since the receiver did not receive packet 2, it will acknowledge packets 3, 4, and 5 with an ACK 1, since 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, since 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, since the receiver may delay replying to a packet with an acknowledgment, knowing its free buffers are available and the window-size of the communication.

5.3.1.2 The window principle applied to TCP

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

- Since 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:

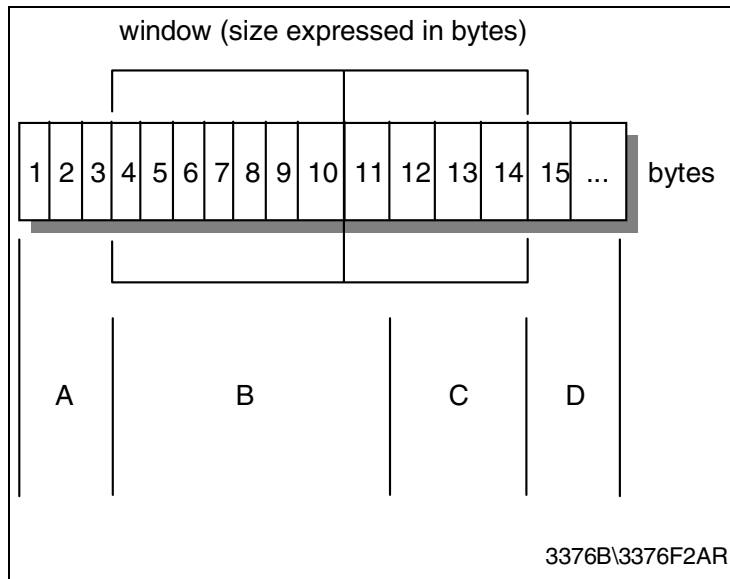


Figure 98. TCP - Window principle applied to TCP

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.

5.3.1.3 TCP segment format

The TCP segment format is shown in Figure 99.

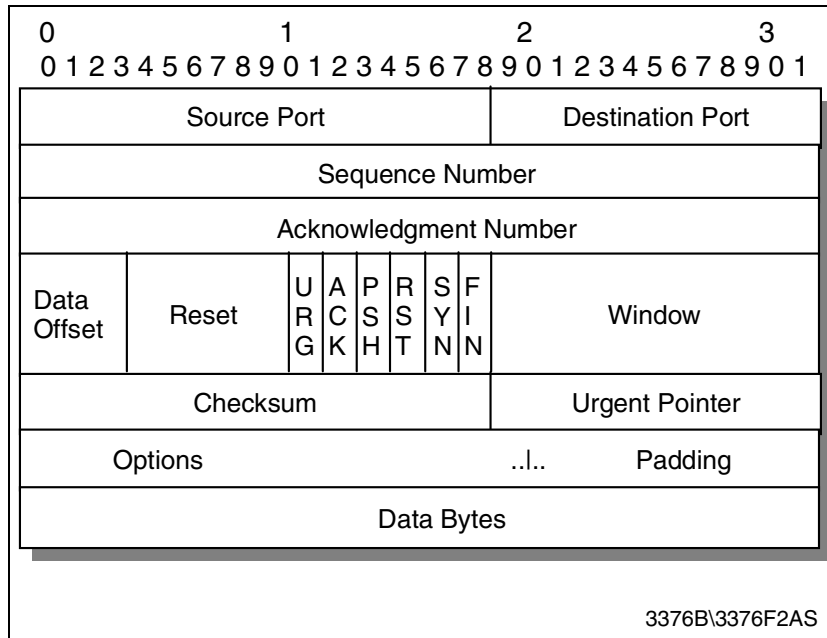


Figure 99. TCP - Segment format

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 this segment. 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 Figure 100.

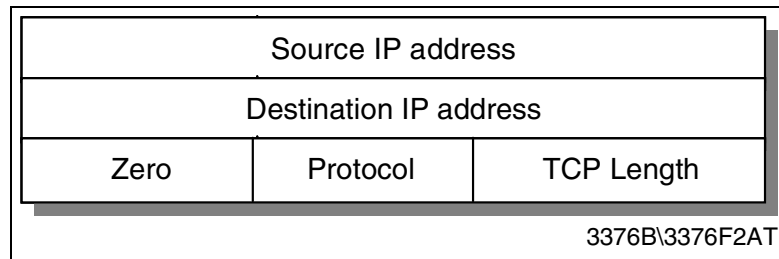


Figure 100. TCP - Pseudo-IP header

- 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 in the following format:

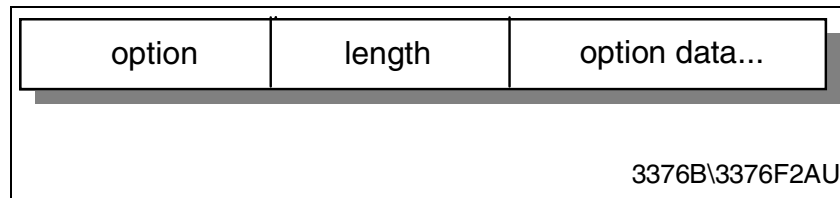


Figure 101. TCP - IP datagram option - variable length option.

There are currently seven options defined:

Table 2. TCP - IP datagram options

Kind	Length	Meaning
0	-	End of option list
1	-	No-Operation
2	4	Maximum segment size
3	3	Window scale
4	2	Sack-Permitted
5	X	Sack
8	10	Timestamps

- 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. See Figure 102 for more details.

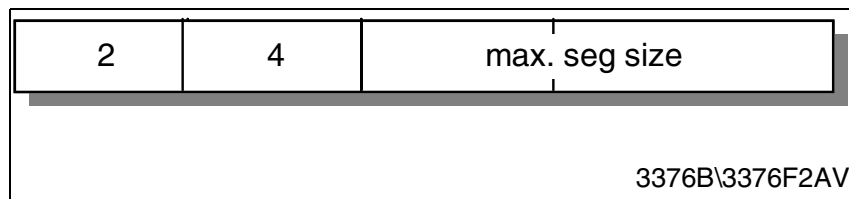


Figure 102. TCP - Maximum segment size option

- Window Scale option: This option is not mandatory. Both sides must send the Windows Scale Option in their SYN segments to enable windows scaling in their direction. The Window Scale expands the definition of the TCP window to 32 bits. It defines the 32-bit window size by using scale factor in the SYN segment over standard 16-bit window size. The receiver rebuild the 32-bit window size by using the 16-bit window size and scale factor. This option is determined while handshaking. There is no way to change it after the connection has been established. See Figure 103 for more details.

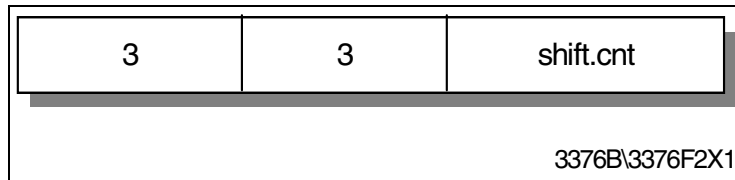


Figure 103. TCP - Window scale option

- SACK-Permitted option: This option is set when selective acknowledgment is used in that TCP connection. See Figure 104 for details.

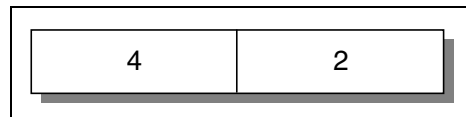


Figure 104. TCP - SACK-permitted option

- SACK option: Selective Acknowledgment (SACK) allows the receiver to inform the sender about all the segments that are received successfully. Thus, the sender will only send the segments that actually got lost. If the number of the segments that have been lost since the last SACK is too large, the SACK option will be too large. As a result, the number of blocks that can be reported by the SACK option is limited to four. To reduce this, the SACK option should be used for the most recent received data. See Figure 105 for more details.

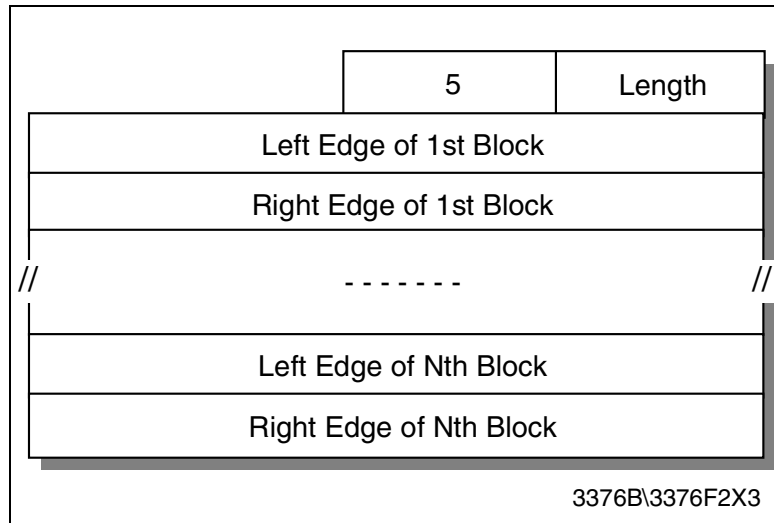


Figure 105. TCP - SACK option

- Timestamps option: The timestamps option sends a timestamp value that indicates the current value of the timestamp clock of the TCP sending the option. Timestamp Echo Value can only be used if the ACK bit is set in the TCP header. See Figure 106 for more details.

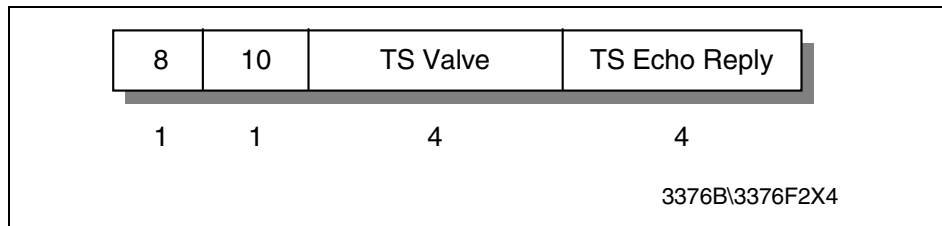


Figure 106. TCP - Timestamps option

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

5.3.1.4 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.

Figure 107 illustrates an example where a window size of 1500 bytes and segments of 500 bytes are used.

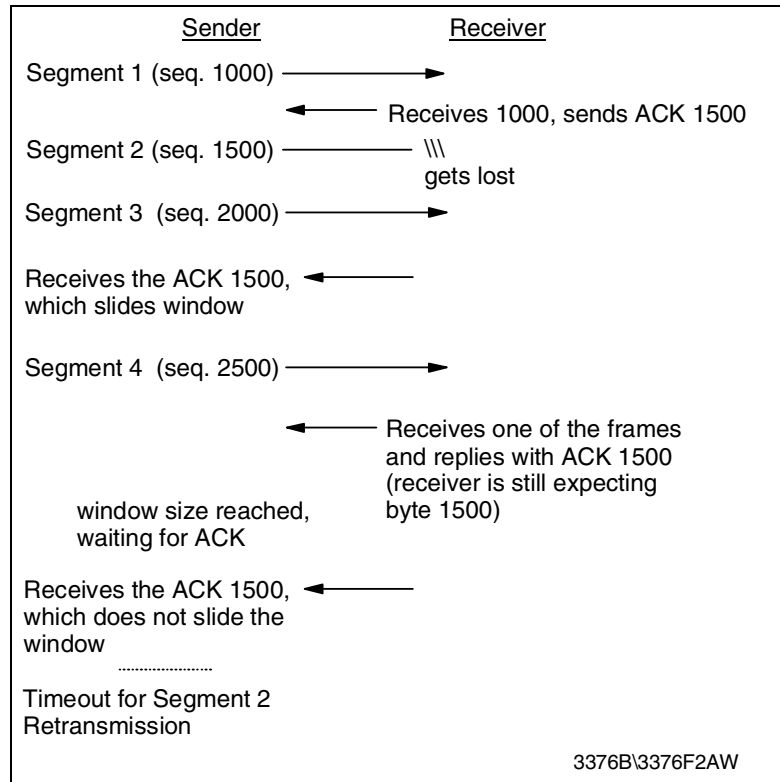


Figure 107. TCP - Acknowledgment and retransmission process

A problem now arises, since 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 (since they are within the current window). It is possible that:

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

Each TCP implementation is free to react to a timeout as the implementing wish. It could retransmit only segment 2, but in case 2, 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.

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(s) 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.

5.3.1.5 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.

On the network, three TCP segments are exchanged:

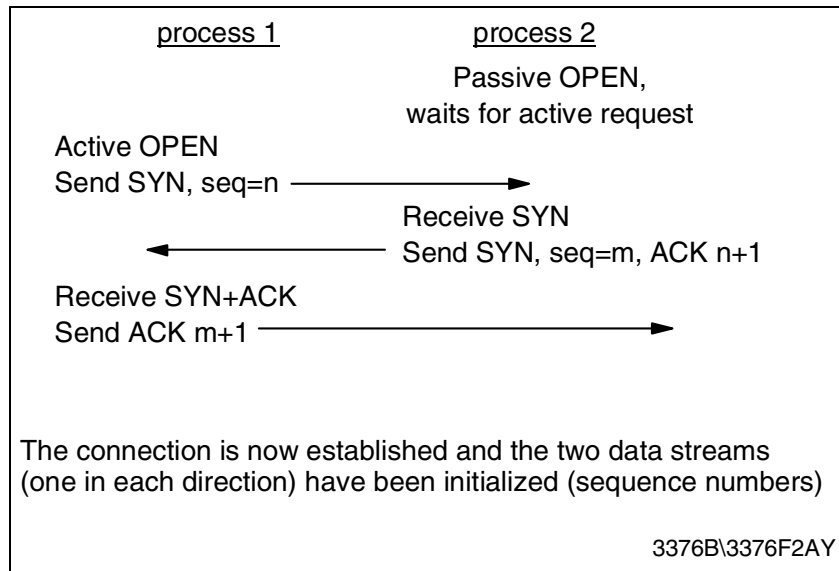


Figure 108. TCP - Connection establishment

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. Since 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) once the data stream is closed in both directions.

5.3.2 TCP application programming interface

The TCP application programming interface is not fully defined. Only some base functions it should provide are described in RFC 793 – Transmission Control Protocol. As is the case with most RFCs in the TCP/IP protocol suite, a great degree of freedom is left to the implementers, thereby allowing for optimal (operating system-dependent) implementations, resulting in better efficiency (greater throughput).

The following function calls are described in the RFC:

- Open: To establish a connection takes several parameters, such as:

- Active/passive
- Foreign socket
- Local port number
- Timeout value (optional)

This returns a local connection name, which is used to reference this particular connection in all other functions.

- Send: Causes data in a referenced user buffer to be sent over the connection. Can optionally set the URGENT flag or the PUSH flag.
- Receive: Copies incoming TCP data to a user buffer.
- Close: Closes the connection; causes a push of all remaining data and a TCP segment with FIN flag set.
- Status: An implementation-dependent call that could return information, such as:
 - Local and foreign socket
 - Send and receive window sizes
 - Connection state
 - Local connection name
- Abort: Causes all pending Send and Receive operations to be aborted, and a RESET to be sent to the foreign TCP.

Full details can be found in RFC 793 – Transmission Control Protocol.

5.3.3 TCP congestion control algorithms

One big difference between TCP and UDP is the congestion control algorithm. The TCP congestion algorithm prevents a sender from overrunning the capacity of the network (for example, slower WAN links). TCP can adapt the sender's rate to network capacity and attempt to avoid potential congestion situations. In order to understand the difference between TCP and UDP, understanding basic TCP congestion control algorithms is very helpful.

Several congestion control enhancements have been added and suggested to TCP over the years. This is still an active and ongoing research area, but modern implementations of TCP contain four intertwined algorithms as basic Internet standards:

- Slow start
- Congestion avoidance
- Fast retransmit
- Fast recovery

5.3.3.1 Slow start

Old implementations of TCP would start a connection with the sender injecting multiple segments into the network, up to the window size advertised by the receiver. While this is OK when the two hosts are on the same LAN, if there are routers and slower links between the sender and the receiver, problems can arise. Some intermediate routers cannot handle it, packets get dropped, retransmission results and performance is degraded.

The algorithm to avoid this is called slow start. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end. Slow start adds another window to the sender's TCP: the congestion window, called *cwnd*. When a new connection is established with a host on another network, the congestion window is initialized to one segment (for example, the segment size announced by the other end, or the default, typically 536 or 512). Each time an ACK is received, the congestion window is increased by one segment. The sender can transmit the lower value of the congestion window or the advertised window. The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver. The former is based on the sender's assessment of perceived network congestion; the latter is related to the amount of available buffer space at the receiver for this connection.

The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential growth, although it is not exactly exponential, because the receiver may delay its ACKs, typically sending one ACK for every two segments that it receives.

At some point, the capacity of the IP network (for example, slower WAN links) can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has gotten too large. Please see Figure 109 for an overview of slow start in action.

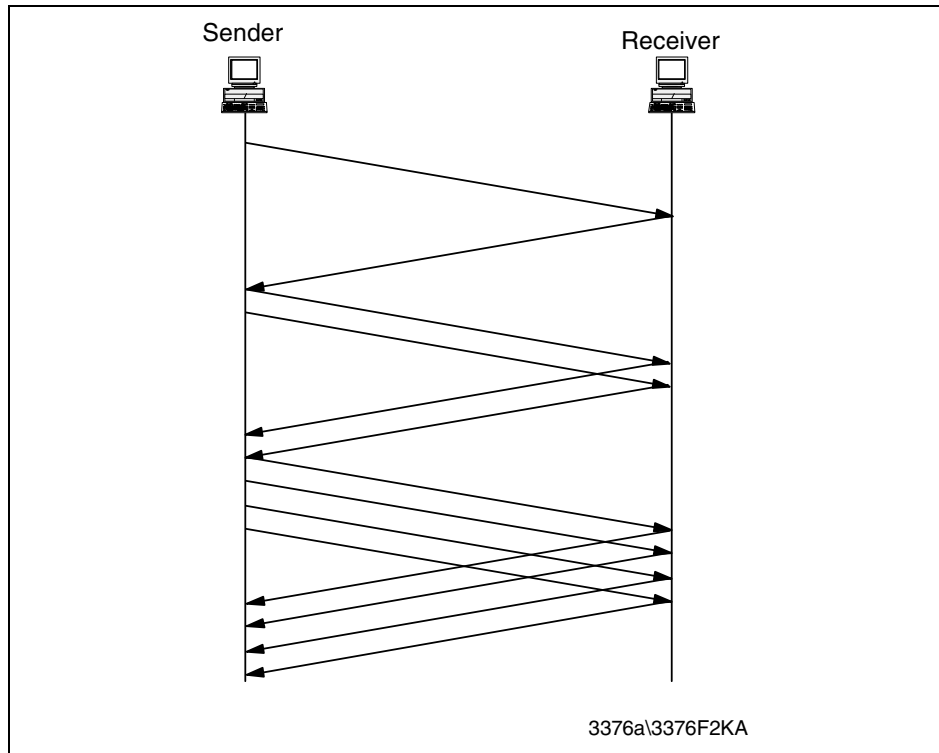


Figure 109. TCP slow start in action

5.3.3.2 Congestion avoidance

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

1. A timeout occurs.
2. 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 $\text{segsize} * \text{segsize} / \text{cwnd}$ 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), whereas 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. Please see Figure 110 for an example of TCP slow start and congestion avoidance in action.

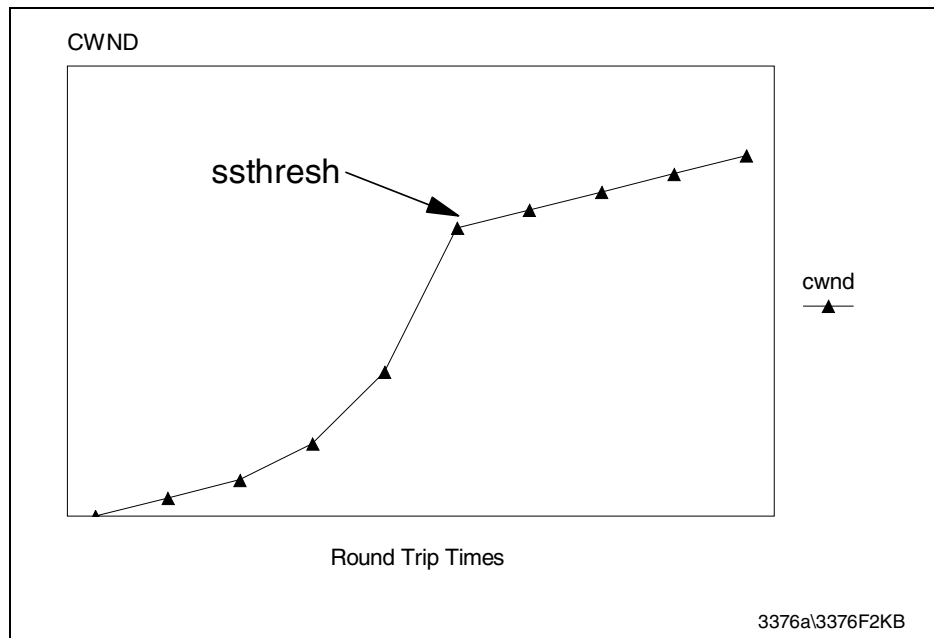


Figure 110. TCP slow start and congestion avoidance behavior in action

5.3.3.3 Fast retransmit

Fast retransmit avoids having TCP wait for a timeout to resend lost segments.

Modifications to the congestion avoidance algorithm were proposed in 1990. Before describing the change, realize that TCP may generate an immediate acknowledgment (a duplicate ACK) when an out-of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected.

Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire. Please see Figure 111 for an overview of TCP fast retransmit in action.

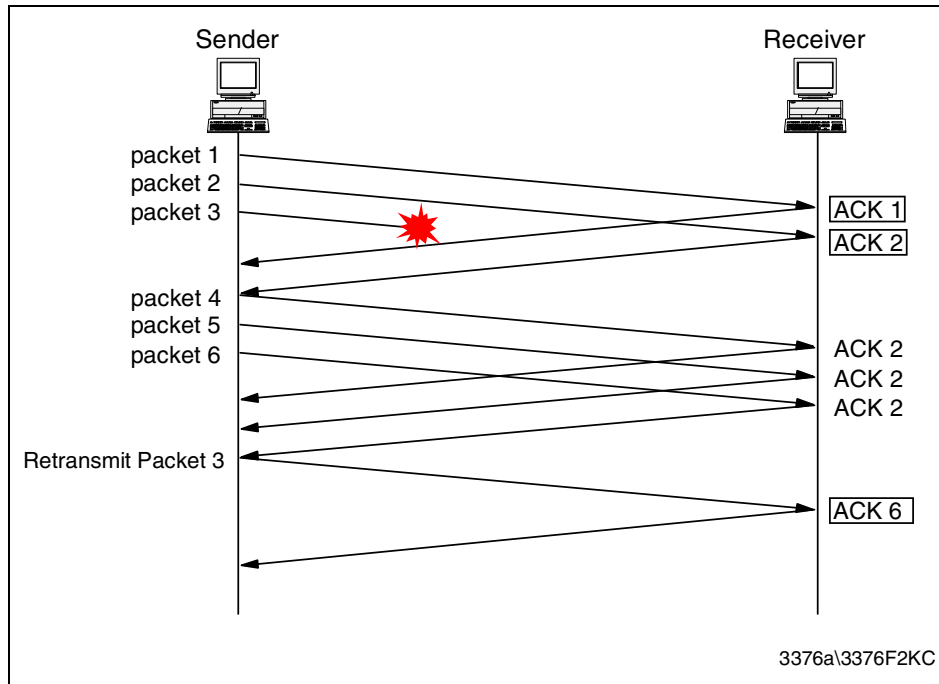


Figure 111. TCP fast retransmit in action

5.3.3.4 Fast recovery

After fast retransmit sends what appears to be the missing segment, congestion avoidance, but not slow start, is performed. This is the fast recovery algorithm. It is an improvement that allows high throughput under moderate congestion, especially for large windows.

The reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells TCP more than just a packet has been lost. Since the receiver can only generate the duplicate ACK when another segment is received, that segment has left the network and is in the receiver's buffer. That is, there is still data flowing between the two ends, and TCP does not want to reduce the flow abruptly by going into slow start. The fast retransmit and fast recovery algorithms are usually implemented together as follows:

1. When the third duplicate ACK in a row is received, set *ssthresh* to one-half the current congestion window, *cwnd*, but no less than two segments. Retransmit the missing segment. Set *cwnd* to *ssthresh* plus three times the segment size. This inflates the congestion window by the number of segments that have left the network and the other end has cached (3).

2. Each time another duplicate ACK arrives, increment cwnd by the segment size. This inflates the congestion window for the additional segment that has left the network. Transmit a packet, if allowed by the new value of cwnd.
3. When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1). This ACK should be the acknowledgment of the retransmission from step 1, one round-trip time after the retransmission. Additionally, this ACK should acknowledge all the intermediate segments sent between the lost packet and the receipt of the first duplicate ACK. This step is congestion avoidance, since TCP is down to one-half the rate it was at when the packet was lost.

