UNIT IV

THE TRANSPORT LAYER

Transport Layer :

- Together with the network layer, the transport layer is the **heart of the protocol** hierarchy.
- The network layer provides **end-to-end packet delivery using datagrams or virtual circuits.**
- The **transport layer builds on the network layer to provide data transport from a process on a source machine to a process on a destination machine with a desired level of reliability** that is **independent of the physical networks currently in use.**
- Transport layer in detail, including its services and **choice of API design to tackle issues of reliability,** connections and congestion control, protocols such as TCP and UDP, and performance.
- Transport layer offers peer-to-peer and end-to-end connection between two processes on remote hosts.
- Transport layer takes data from upper layer (i.e. Application layer) and then breaks it into smaller size segments, numbers each byte, and hands over to lower layer (Network Layer) for delivery.
  - **Transport layer Data Represented in Segments.**
  - **Transport layer provides logical communication b/w Applications on different hosts. (i.e sender side and Receiver side).**
  - **It is Responsible for delivering data End to End message delivery.**
  - **It is responsible for Providing Acknowledgement for successful transmission of data.**
  - **If any error occur during transmission of data this layer is responsible for Re-Transmission of data.**
  - **Transport layer protocols are implemented at End systems but not at routers.**
  - **Two main protocols used in Transport protocols i.e TCP and UDP.**
  - **These two protocols provide services to Transport layer.**

Different services of Transport Layer :

- Transport layer provide Different services of Transport layer.
  - End to End Data Delivery.
  - Reliable delivery of Data.(i.e with out loss of data / with out any errors).
  - Error control mechanisms.
  - Sequence control of data delivery.
  - To avoid Loss Control of data.
  - To avoid duplication control of data.
  - Flow control .(if receiver is overloaded of receiving information then data may loss.)
  - Addressing .(Each segment having header , i.e header + segment . Each header consist of port address with the help of port address directly information sends to correct destination).
  - Connection Oriented Services and Connection less services.

THE TRANSPORT SERVICE:

1. Services Provided to the Upper Layers:

- The ultimate goal of the transport layer is to provide **efficient, reliable, and cost-effective data transmission service** to its users, normally processes **in the application layer**.
- To achieve this, the **transport layer makes use of the services provided by the network layer.**
- The software (kernel)and/or hardware (network interface card)within the transport layer that does the work is called the **transport entity**.
- There are 2 types of services TCP And UDP.
- In both cases connection has 3 phase
- Connection Establish.
- Data Transfer.
- Connection Release.

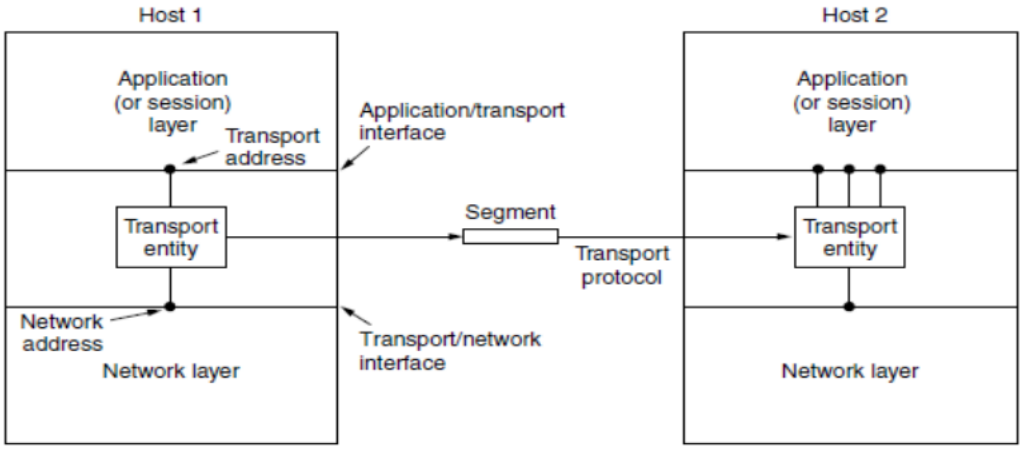


Figure 6-1. The network, transport, and application layers.

2. Transport Service Primitives:

To allow users to access the transport service, the transport layer must provide some operations to application programs, that is, a transport service interface. Each transport service has its own interface.

The transport service is similar to the network service, but there are also some important differences. The main difference is that the network service is intended to model the service offered by real networks, warts and all. Real networks can lose packets, so the network service is generally unreliable. The connection-oriented transport service, in contrast, is reliable. Of course, real networks are not error-free, but that is precisely the purpose of the transport layer—to provide a reliable service on top of an unreliable network.

To get an idea of what a transport service might be like, consider the five primitives listed in Fig. 6-2.

This transport interface is truly bare bones, but it gives the essential flavour of what a connection-oriented transport interface has to do. It allows application programs to establish, use, and then release connections, which is sufficient for many applications.

Listen	When server is ready to accept request of incoming connection, it simply put this primitive into action. Listen primitive simply waiting for incoming connection request.
Connect	This primitive is used to connect the server simply by creating or establishing connection with waiting peer.
Receive	These primitive afterwards block the server. Receive primitive simply waits for incoming message.
Send	This primitive is put into action by the client to transmit its request that is followed by putting receive primitive into action to get the reply. Send primitive simply sends or transfer the message to the peer.
Disconnect	This primitive is simply used to terminate or end the connection after which no one will be able to send any of the message.

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	Request a release of the connection

Figure 6-2. The primitives for a simple transport service.

Segments (exchanged by the transport layer) are contained in packets (exchanged by the network layer). In turn, these packets are contained in frames (exchanged by the data link layer). When a frame arrives, the data link layer processes the frame header and, if the destination address matches for local delivery, passes the contents of the frame payload field up to the network entity. The network entity similarly processes the packet header and then passes the contents of the packet payload up to the transport entity. This nesting is illustrated in Fig. 6-3.

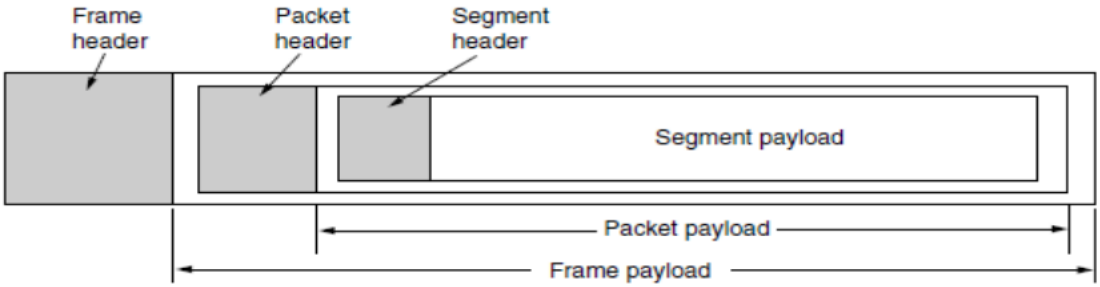


Figure 6-3. Nesting of segments, packets, and frames.

3. Berkeley Sockets:

Let us now briefly inspect another set of transport primitives, the socket primitives as they are used for TCP. Sockets were first released as part of the Berkeley UNIX 4.2BSD software distribution in 1983. They quickly became popular.

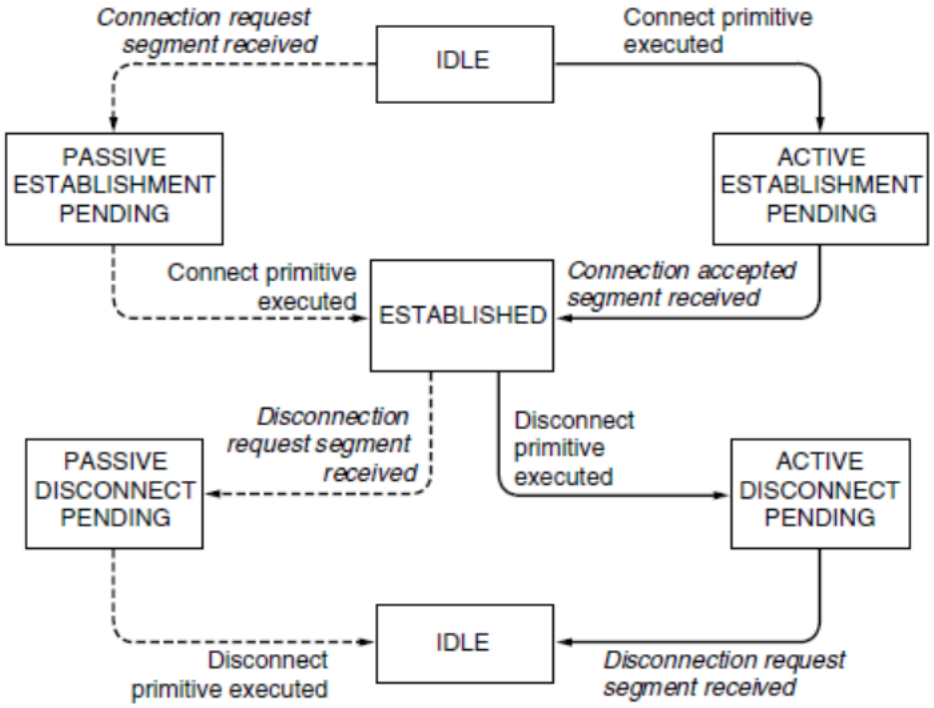


Figure 6-4. A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.

Primitive	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Figure 6-5. The socket primitives for TCP.

ELEMENTS OF TRANSPORT PROTOCOLS:

The transport service is implemented by a **transport protocol** used between the two transport entities. At the data link layer, two routers communicate directly via a physical channel, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network. This difference has many important implications for the protocols.

For one thing, over point-to-point links such as wires or optical fiber, it is usually not necessary for a router to specify which router it wants to talk to—each outgoing line leads directly to a particular router. In the transport layer, explicit addressing of destinations is required.

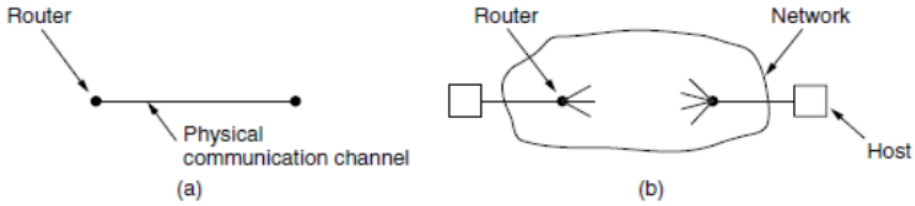


Figure 6-7. (a) Environment of the data link layer. (b) Environment of the transport layer.

Elements of Transport protocols are:

1.Addressing.
2. Connection Establishment.
3. Connection Release.
4. Error control and Flow control.
5. Multiplexing.

1. Addressing:

When an application (e.g., a user) process wishes to set up a connection to a remote application process, it must specify which one to connect to. (Connectionless transport has the same problem: to whom should each message be sent?) The method normally used is to define transport addresses to which processes can listen for connection requests. In the Internet, these endpoints are called **ports**. We will use the generic term **TSAP** (**Transport Service Access Point**) to mean a specific endpoint in the transport layer. The analogous endpoints in the network layer (i.e., network layer addresses) are not-surprisingly called **NSAPs** (**Network Service Access Points**). IP addresses are examples of NSAPs.

Figure 6-8 illustrates the relationship between the NSAPs, the TSAPs, and a transport

connection. Application processes, both clients and servers, can attach themselves to a local TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown. The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport endpoints that share that NSAP.

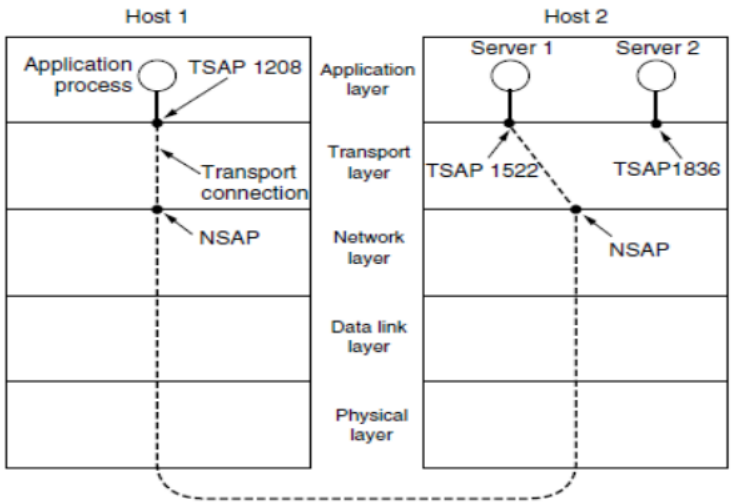


Figure 6-8. TSAPs, NSAPs, and transport connections.

2. Connection Establishment:

Establishing a connection sounds easy, but it is actually surprisingly tricky. At first glance, it would seem sufficient for one transport entity to just send a CONNECTION REQUEST segment to the destination and wait for a CONNECTION ACCEPTED reply. The problem occurs when the network can lose, delay, corrupt, and duplicate packets. This behaviour causes serious complications.

To solve connection establishment problem, Tomlinson (1975) introduced the **three-way handshake**. This establishment protocol involves one peer checking with the other that the connection request is indeed current. The normal setup procedure when host 1 initiates is shown in Fig. 6-11(a). Host 1 chooses a sequence number, x, and sends a CONNECTION REQUEST segment containing it to host 2. Host 2 replies with an ACK segment acknowledging x and announcing its own initial sequence number, y. Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data segment that it sends.

Now let us see how the three-way handshake works in the presence of delayed duplicate control segments. In Fig. 6-11(b), the first segment is a delayed duplicate CONNECTION REQUEST from an old connection. This segment arrives at host 2 without host 1's knowledge. Host 2 reacts to this segment by sending host 1 an ACK segment, in effect asking for verification that host 1 was indeed trying to set up a new connection. When host 1 rejects host 2's attempt to establish a connection, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage.

The worst case is when both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet. This case is shown in Fig. 6-11(c). As in the previous example, host 2 gets a delayed CONNECTION REQUEST and replies to it. At this point, it is crucial to realize that host 2 has proposed using y as the initial sequence number for host 2 to host 1 traffic, knowing full well that no segments containing sequence number y or acknowledgements to y are still in existence. When the second delayed segment arrives at host 2, the fact that z has been acknowledged rather than y tells host 2 that this, too, is an old duplicate. The important thing to realize here is that there is no combination of old segments that can cause the protocol to fail and have a connection set up by accident when no one wants it.

TCP uses this three-way handshake to establish connections.

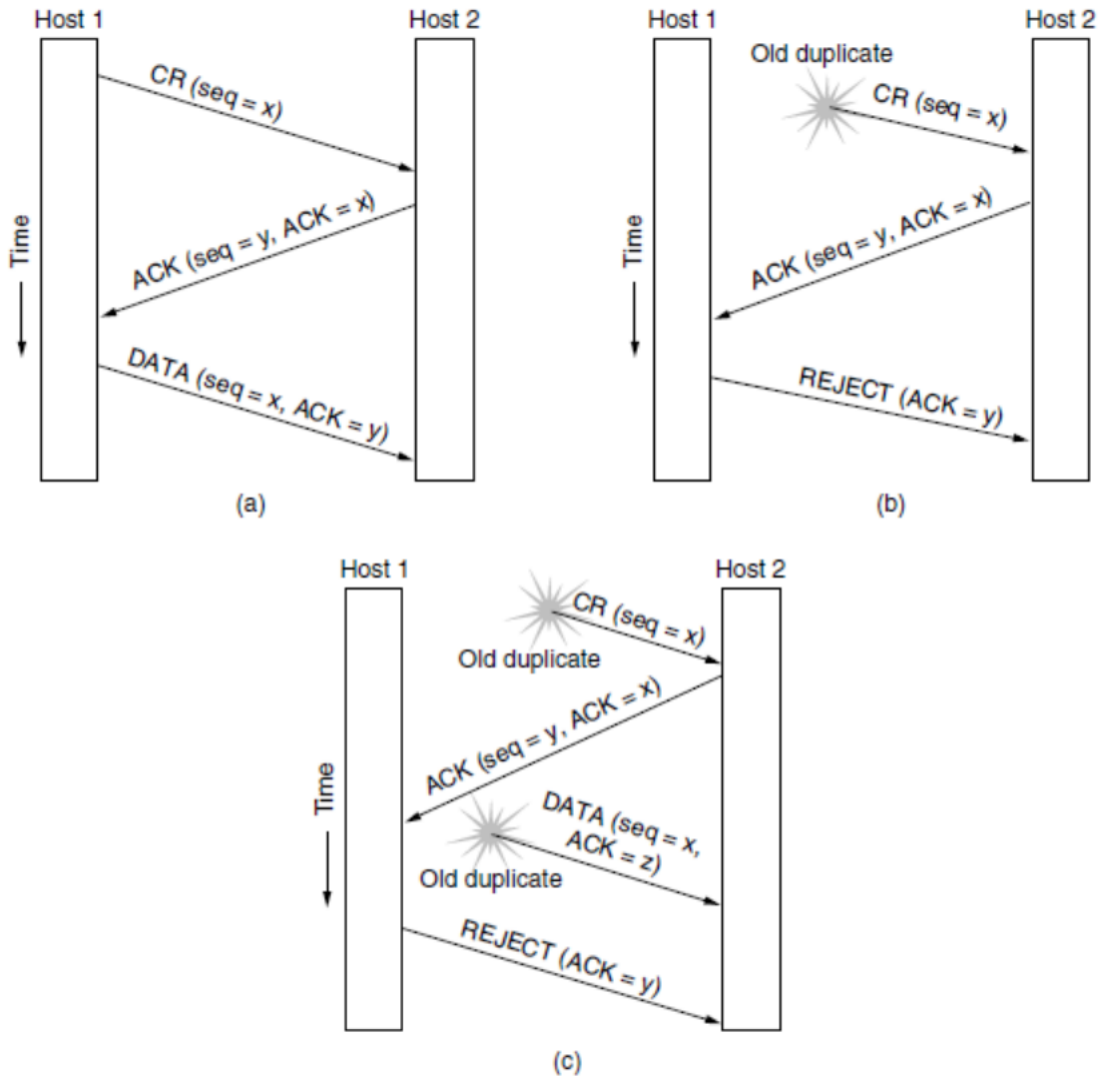


Figure 6-11. Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST. (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK.

3. 3.

3.Connection Release:

Releasing a connection is easier than establishing one. Nevertheless, there are more pitfalls than one

might expect here. As we mentioned earlier, there are two styles of terminating a connection: asymmetric release and symmetric release.

Asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken. Symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately. Asymmetric release is abrupt and may result in data loss. Consider the scenario of Fig. 6-12. After the connection is established, host 1 sends a segment that arrives properly at host 2. Then host 1 sends another segment. Unfortunately, host 2 issues a DISCONNECT before the second segment arrives. The result is that the connection is released and data are lost.

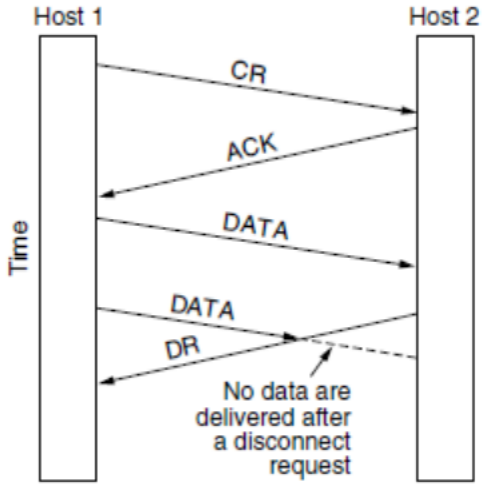


Figure 6-12. Abrupt disconnection with loss of data.

Clearly, a more sophisticated release protocol is needed to avoid data loss. One way is to use symmetric release, in which each direction is released independently of the other one. Here, a host can continue to receive data even after it has sent a DISCONNECT segment.

Symmetric release does the job when each process has a fixed amount of data to send and clearly knows when it has sent it. In other situations, determining that all the work has been done and the connection should be terminated is not so obvious. One can envision a protocol in which host 1 says "I am done. Are you done too?" If host 2 responds: "I am done too. Goodbye, the connection can be safely released."

Unfortunately, this protocol does not always work. There is a famous problem that illustrates this issue. It is called the **two-army problem**. Imagine that a white army is encamped in a valley, as shown in Fig. 6-13. On both of the surrounding hillsides are blue armies. The white army is larger than either of the blue armies alone, but together the blue armies are larger than the white army. If either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.

The blue armies want to synchronize their attacks. However, their only communication medium is to send messengers on foot down into the valley, where they might be captured and the message lost (i.e., they have to use an unreliable communication channel). The question is: does a protocol exist that allows the blue armies to win?

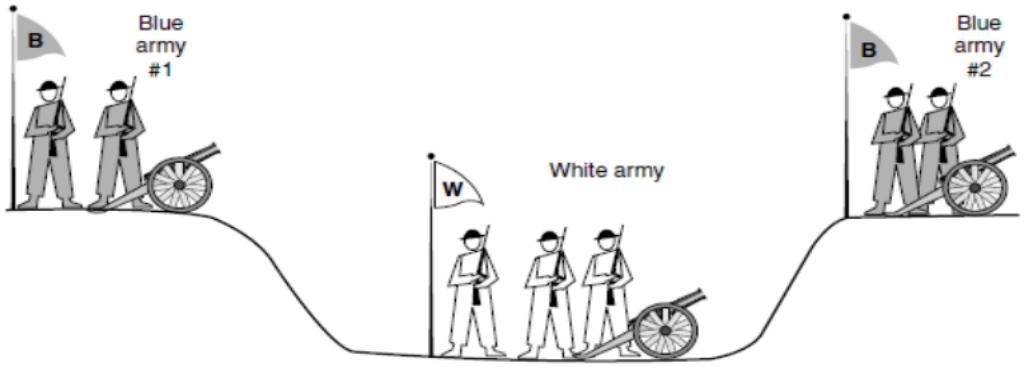


Figure 6-13. The two-army problem.

Suppose that the commander of blue army #1 sends a message reading: "I propose we attack at dawn on March 29. How about it?" Now suppose that the message arrives, the commander of blue army #2 agrees, and his reply gets safely back to blue army #1. Will the attack happen? Probably not, because commander #2 does not know if his reply got through. If it did not, blue army #1 will not attack, so it would be foolish for him to charge into battle.

Now let us improve the protocol by making it a three-way handshake. The initiator of the original proposal must acknowledge the response. Assuming no messages are lost, blue army #2 will get the acknowledgement, but the commander of blue army #1 will now hesitate. After all, he does not know if his acknowledgement got through, and if it did not, he knows that blue army #2 will not attack. We could now make a four-way handshake protocol, but that does not help either.

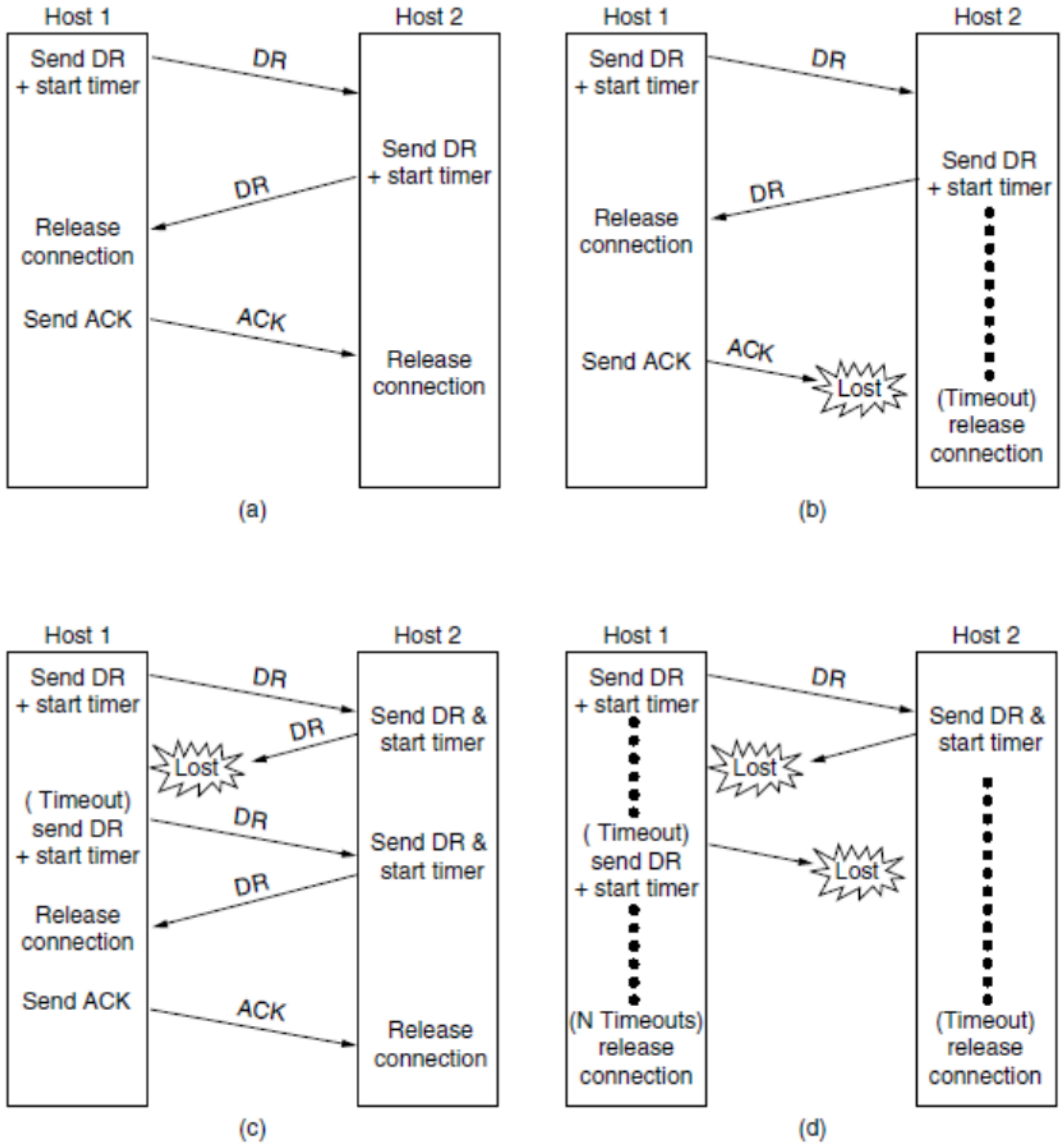


Figure 6-14. Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

In Fig. 6-14(a), we see the normal case in which one of the users sends a DR (DISCONNECTION REQUEST) segment to initiate the connection release. When it arrives, the recipient sends back a DR segment and starts a timer, just in case its DR is lost. When this DR arrives, the original sender sends back an ACK segment and releases the connection. Finally, when the ACK segment arrives, the receiver also releases the connection. Releasing a connection means that the transport entity removes the information about the connection from its table of currently open connections and signals the connection's owner (the

transport user) somehow. This action is different from a transport user issuing a DISCONNECT primitive.

If the final ACK segment is lost, as shown in Fig. 6-14(b), the situation is saved by the timer. When the timer expires, the connection is released anyway.

Now consider the case of the second DR being lost. The user initiating the disconnection will not receive the expected response, will time out, and will start all over again. In Fig. 6-14(c), we see how this works, assuming that the second time no segments are lost and all segments are delivered correctly and on time.

Our last scenario, Fig. 6-14(d), is the same as Fig. 6-14(c) except that now we assume all the repeated attempts to retransmit the DR also fail due to lost segments. After N retries, the sender just gives up and releases the connection. Meanwhile, the receiver times out and also exits.

While this protocol usually suffices, in theory it can fail if the initial DR and N retransmissions are all lost. The sender will give up and release the connection, while the other side knows nothing at all about the attempts to disconnect and is still fully active. This situation results in a half-open connection.

4. Error Control and Flow Control:

Having examined connection establishment and release in some detail, let us now look at how connections are managed while they are in use. The key issues are error control and flow control. Error control is ensuring that the data is delivered with the desired level of reliability, usually that all of the data is delivered without any errors. Flow control is keeping a fast transmitter from overrunning a slow receiver.

There still remains the question of how to organize the buffer pool. If most segments are nearly the same size, it is natural to organize the buffers as a pool of identically sized buffers, with one segment per buffer, as in Fig. 6-15(a). However, if there is wide variation in segment size, from short requests for Web pages to large packets in peer-to-peer file transfers, a pool of fixed-sized buffers presents problems. If the buffer size is chosen to be equal to the largest possible segment, space will be wasted whenever a short segment arrives. If the buffer size is chosen to be less than the maximum segment size, multiple buffers will be needed for long segments, with the attendant complexity.

Another approach to the buffer size problem is to use variable-sized buffers, as in Fig. 6-15(b). The advantage here is better memory utilization, at the price of more complicated buffer management. A third possibility is to dedicate a single large circular buffer per connection, as in Fig. 6-15(c). This system is simple and elegant and does not depend on segment sizes, but makes good use of memory only when the connections are heavily loaded.

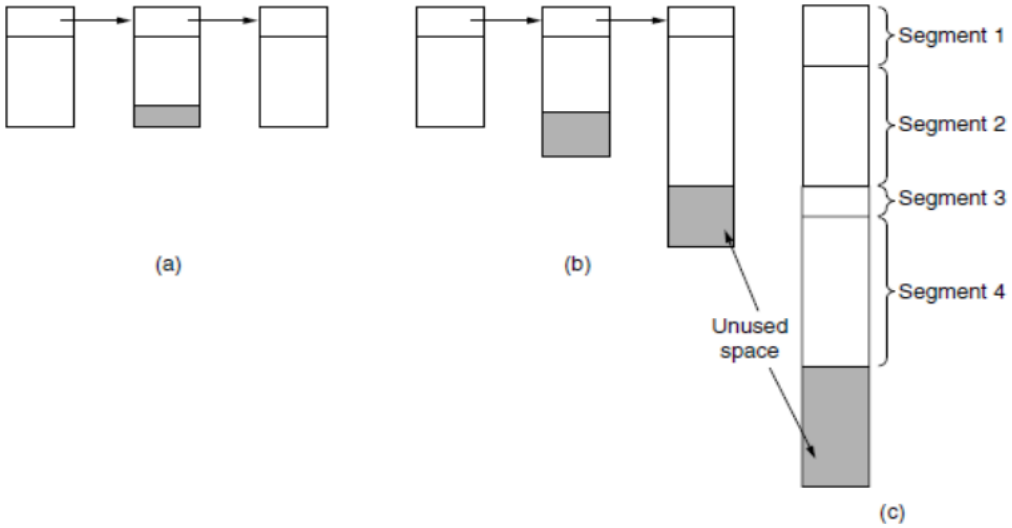


Figure 6-15. (a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connection.

5. Multiplexing:

Multiplexing, or sharing several conversations over connections, virtual circuits, and physical links plays a role in several layers of the network architecture. In the transport layer, the need for multiplexing can arise in a number of ways. For example, if only one network address is available on a host, all transport connections on that machine have to use it. When a segment comes in, some way is needed to tell which process to give it to. This situation, called ***multiplexing****, is shown in Fig. 6-17(a). In this figure, four distinct transport connections all use the same network connection (e.g., IP address) to the remote host.*

Multiplexing can also be useful in the transport layer for another reason. Suppose, for example, that a host has multiple network paths that it can use. If a user needs more bandwidth or more reliability than one of the network paths can provide, a way out is to have a connection that distributes the traffic among multiple network paths on a round-robin basis, as indicated in Fig. 6-17(b). This modus operandi is called ***inverse multiplexing****.*

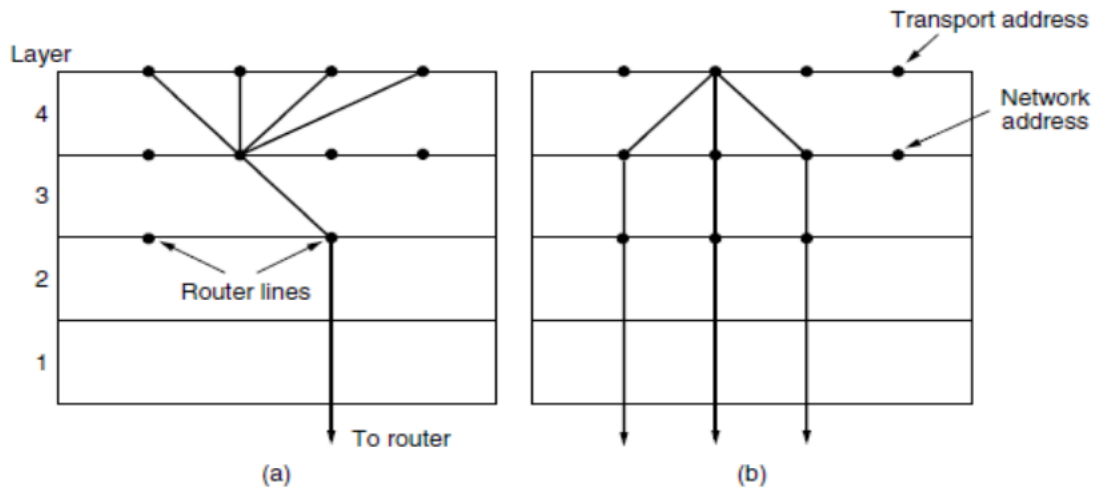


Figure 6-17. (a) Multiplexing. (b) Inverse multiplexing.

6. Crash Recovery:

If hosts and routers are subject to crashes or connections are long-lived (e.g., large software or media downloads), recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from networkand router crashes is straightforward. The transport entities expect lost segments all the time and know how to cope with them by using retransmissions.

A more troublesome problem is how to recover from host crashes. In particular, it may be desirable for clients to be able to continue working when servers crash and quickly reboot. To illustrate the difficulty, let us assume that one host, the client, is sending a long file to another host, the file server, using a simple stop-and-wait protocol. The transport layer on the server just passes the incoming segments to the transport user, one by one. Partway through the transmission, the server crashes. When it comes back up, its tables are reinitialized, so it no longer knows precisely where it was.

In an attempt to recover its previous status, the server might send a broadcast segment to all other hosts, announcing that it has just crashed and requesting that its clients inform it of the status of all open connections. Each client can be in one of two states: one segment outstanding, *S1*, or no segments outstanding, *S0*. Based on only this state information, the client must decide whether to retransmit the most recent segment.

THE INTERNET TRANSPORT PROTOCOLS: UDP

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection-oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed. The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it.

Introduction to UDP:

The Internet protocol suite supports a connectionless transport protocol called **UDP (User Datagram Protocol)**. UDP provides a way for applications to send encapsulated IP datagrams without having to establish a connection.

UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 6-27. The two **ports** serve to identify the endpoints within the source and destination machines. When a UDP packet arrives, its payload is handed to the process attached to the destination port.

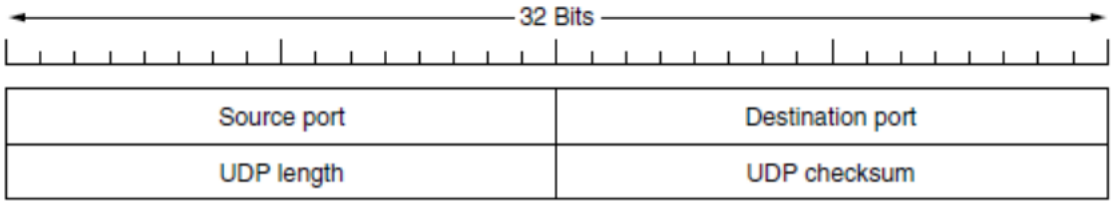


Figure 6-27. The UDP header.

The source port is primarily needed when a reply must be sent back to the source. By copying the ***Source port*** field from the incoming segment into the ***Destination port*** field of the outgoing segment, the process sending the reply can specify which process on the sending machine is to get it. The ***UDP length*** field includes the 8-byte header and the data. An optional *Checksum* is also provided for extra reliability. It checksums the header ,the data, and a conceptual IP pseudoheader. When performing this computation, the *Checksum* field is set to zero and the data field is padded out with an additional zero byte if its length is an odd number.

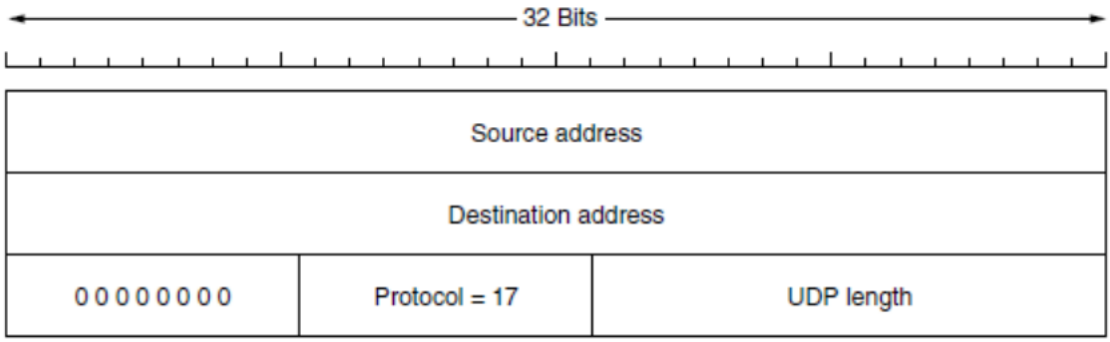


Figure 6-28. The IPv4 pseudoheader included in the UDP checksum.

• Remote Procedure Call: (RPC)

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on remote hosts. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing

is visible to the application programmer. This technique is known as **RPC (Remote Procedure Call)** and has become the basis for many networking applications. Traditionally, the calling procedure is known as the client and the called procedure is known as the server, and we will use those names here too.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure, called the **client stub**, that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local.

Step 1 is the **client calling the client stub**. This call is a local procedure call, with the **parameters pushed onto the stack in the normal way.**

Step 2 is the **client stub packing the parameters into a message and making a system call to send the message.** Packing the parameters is called **marshalling**.

Step 3 is the OS **operating system sending the message from the client machine to the server machine.**

Step 4 is the **operating system passing the incoming packet to the server stub**. Finally,

step 5 is the **server stub calling the server procedure** with the un marshalled parameters.

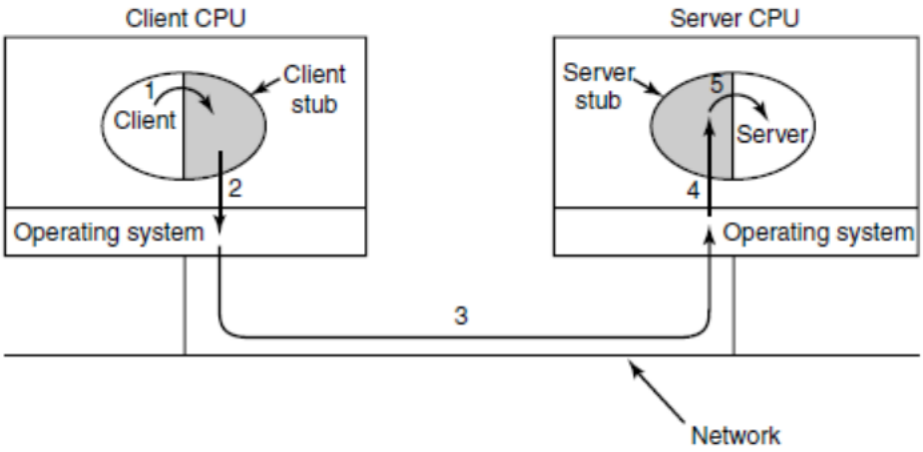


Figure 6-29. Steps in making a remote procedure call. The stubs are shaded.

• Real-Time Transport Protocols: (RTP)

Client-server RPC is one area in which UDP is widely used. Another one is for real-time multimedia applications. In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on-demand, and other multimedia applications became more commonplace, people have discovered that each application was reinventing more or less the same real-time transport protocol. It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea.

Thus was **RTP (Real-time Transport Protocol)** born. It is now in widespread use for multimedia applications. We will describe two aspects of real-time transport. The first is the RTP protocol for transporting audio and video data in packets. The second is the processing that takes place, mostly at the receiver, to play out the audio and video at the right time. These functions fit into the protocol stack as shown in Fig. 6-30.

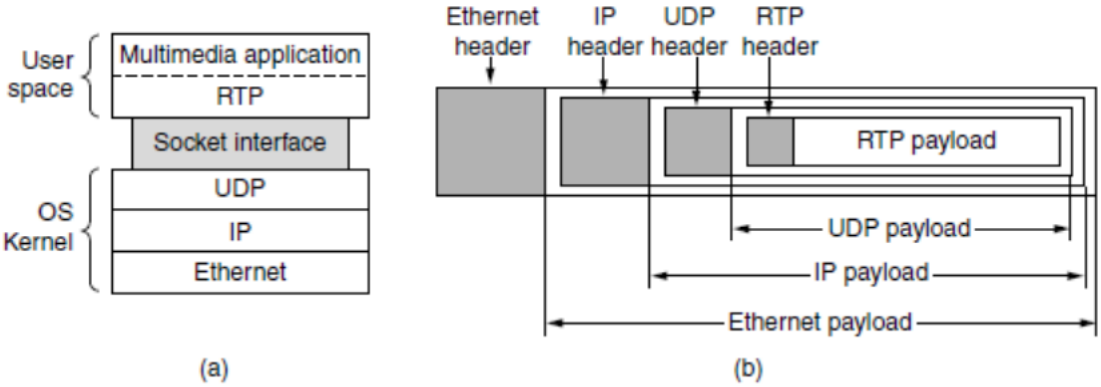


Figure 6-30. (a) The position of RTP in the protocol stack. (b) Packet nesting.

RTP—The Real-time Transport Protocol

- A protocol is designed to handle real-time traffic (like audio and video) of the Internet, is known as Real Time Transport Protocol (RTP). RTP must be used with UDP. It does not have any delivery mechanism like multicasting or port numbers. RTP supports different formats of files like MPEG and MJPEG. It is very sensitive to packet delays and less sensitive to packet loss.
- RTP time transport protocols run over UDP user datagram protocol.
- Client-server RPC is one area in which UDP is widely used.
- Another one is for real-time multimedia applications.
- In particular, as Internet radio, Internet telephony, music-on-demand, videoconferencing, video-on -demand, and other multimedia applications became more commonplace, people have discovered that each application was reinventing more or less the same real-time transport protocol.
- It gradually became clear that having a generic real-time transport protocol for multiple applications would be a good idea.

- The **basic function of RTP is to multiplex several real-time data streams onto a single stream of UDP packets.**
- The **UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting).**
  The RTP format contains several features to help receivers work with multimedia information.
- Each packet sent in an RTP stream is **given a number** one **higher than its predecessor. This numbering allows the destination to determine if any packets are missing.**
- **If a packet is missing, the best action for the destination to take is up to the application.**
- Another facility many real-time applications need is **timestamping.** The idea here is to allow the source to associate **a timestamp with the first sample in each packet.** The timestamps are relative to the start of the stream, so only the differences between timestamps are significant. The absolute values have no meaning. As we will describe shortly, this mechanism allows the destination to do a small amount of buffering and play each sample the right number of milliseconds after the start of the stream, independently of when the packet containing the sample arrived.

- **RTP–HEADER FORMAT:**

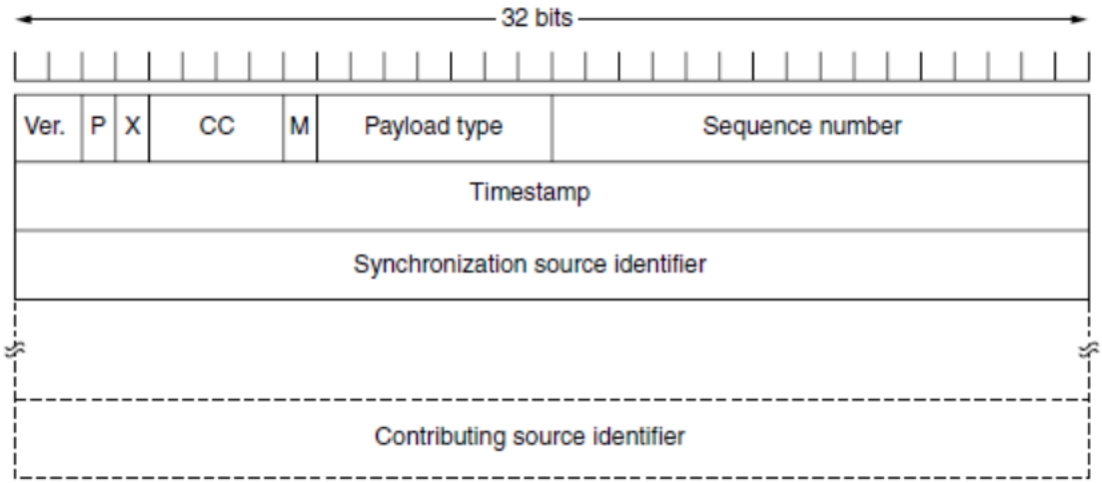


Figure 6-31. The RTP header.

Version:

This **2-bit** field defines version number. The current version is 2.

1. **P–**

   The length of this field is **1-bit**. If value is 1, then it denotes presence of padding at end of packet and if **value is 0, then there is no padding.**

1. **X–**

   The length of this field is also **1-bit**. If value of this field is **set to 1, then its indicates an extra extension** header between data and basic header and if **value is 0 then, there is no extra extension.**

2. **Contributor_count–**

   This **4-bit** field indicates number of contributors. Here maximum possible number of contributor is 15 as a 4-bit field can allows number form 0 to 15.

3. **M–**

   The length of this field is **1-bit** and it is used as **end marker by application to indicate end of its data.**

4. **Payload_types –**

   This field is of length **7-bit** to indicate type of payload. We list applications of some common types of payload.

- **Sequence Number –**

The length of this field is **16 bits**. It is **used to give serial numbers to RTP packets**. It helps in sequencing.

- The **sequence number for first packet is given a random number and then every next packet's sequence number is incremented by 1.**
- **This field mainly helps in checking lost packets and order mismatch.**

- **Time Stamp –**

The length of this field is **32-bit**. *It is used to* **find relationship between times of different RTP packets**.
The **timestamp for first packet is given randomly** *and then time stamp for* **next packets given by sum of previous timestamp and time taken to produce first byte of current packet**. *The value of 1 clock tick is varying from application to application.*

- **Synchronization Source Identifier –**

This is a **32-bit** *field used* **to identify and define the source**. *The value for this source identifier is a random number that is chosen by source itself. This mainly helps in solving conflict arises when two sources started with the same sequencing number.*

- **Contributor Identifier –**

This is also a **32-bit** *field used for source identification where there* **is more than one source present in session**. *The mixer source use Synchronization source identifier and other remaining sources (maximum 15) use Contributor identifier.*

RTCP–The Real-time Transport Control Protocol:

RTP has a little sister protocol (little sibling protocol?) called **RTCP (Realtime Transport Control Protocol)**. *It handles feedback, synchronization, and the user interface. It does not transport any media samples.*

RTCP–The Real-time Transport Control Protocol:

- *RTCP is the protocol of RTP.*
- *RTCP provides feedback on the quality of the data distribution.*
- *It does not support transport of data.*
- *RTCP provides the feedback on delay , jitter (variation in the delay of received packets in the network), congestion, Bandwidth, to the sources.*
- *This feedback information is sent in the form of RTCP sender and receiver reports.*
- *This information can be used by the encoding process to increase data rate.*
- *RTCP defines several types of packets to carry different types of control information.*
- **Types of Packets are : Sender Report (SR) , Receiver report (RR), Source description (SDES) and BYE,APP.**
- **SR sender report gives transmission and reception statistics from active sender.**
- **Receiver Report RR gives reception statistics from participants that are not active senders.**
- **SDES provides souse description items such as Cname , email, name, phone number ,location ,etc.**
- **BYE indicates the end of the participation by the sender.**
- **APP provides application specific functions that are defined in profile specifications.**

THE INTERNET TRANSPORT PROTOCOLS: TCP

- TCP is an connection oriented protocol where as UDP is connection less protocol.
- TCP Provides connection oriented and Reliable , byte stream services.
- The term Connection oriented means the two applications using TCP must establish a TCP connection with each other before they can exchange data.

TCP Services:

- TCP and UDP use the same N/W layer (IP),TCP provides totally different services.
- TCP provides a connection oriented , Reliable , byte stream services. There are Exactly Two End points communicating with each other.
- TCP does not support Multicasting and Broad casting .
- The application data is broken into what TCP consider the best sized chunks to send. The unit of information is send by TCP to IP is called Segment.
- When TCP Sends segment it maintains Timer , waiting for other End Acknowledgement ,it is not received so Retransmits the message.
- TCP receives information from other end points , then it maintains Checksum on its header data.
- TCP segments are transmitted as IP datagrams , and since IP datagrams can arrive out of order TCP segments can arrive out of order . Since IP datagrams can arrive duplicated data then it is discarded.
- TCP also provides flow control . Each end point of TCP connection has finite amount of Buffer Space.

The TCP Protocol:

- The sending and receiving TCP entities exchange data in the form of segments.
- A TCP segment consists of a fixed 20-byte header (plus an optional part) followed by zero or more data bytes.
- The TCP software decides how big segments should be. It can accumulate data from several writes into one segment or can split data from one write over multiple segments.
- 2 limits restrict the segment size.
  . First, each segment, including the TCP header,
  . Second, each link has an MTU (Maximum Transfer Unit).
- Each segment must fit in the MTU at the sender and receiver so that it can be sent and received in a single, un fragmented packet.
- However, it is still possible for IP packets carrying TCP segments to be fragmented when passing over a network path for which some link has a small MTU.

- ## The TCP Service Model:

TCP service is obtained by both the sender and the receiver creating end points, called **sockets**, as discussed in Sec. 6.1.3. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit number local to that host, called a **port**. A port is the TCP name for a TSAP. For TCP service to be obtained, a connection must be explicitly established between a socket on one machine and a socket on another machine.

All TCP connections are full duplex and point-to-point. Full duplex means that traffic can go in both directions at the same time. Point-to-point means that each connection has exactly two end points. TCP does not support multicasting or broadcasting.

A TCP connection is a byte stream, not a message stream. Message boundaries are not preserved end to end. For example, if the sending process does four 512-byte writes to a TCP stream, these data may be delivered to the receiving process as four 512-byte chunks, two 1024-byte chunks, one 2048-byte chunk (see Fig. 6-35), or some other way. There is no way for the receiver to detect the unit(s) in which the data were written, no matter how hard it tries.

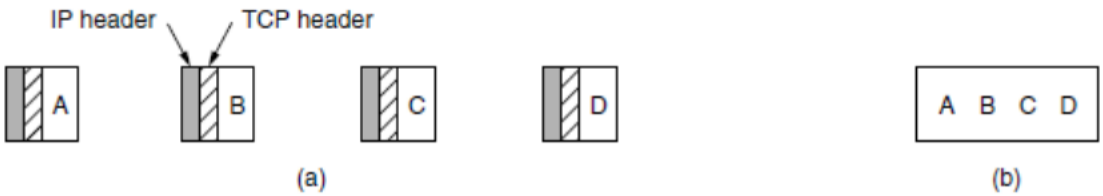
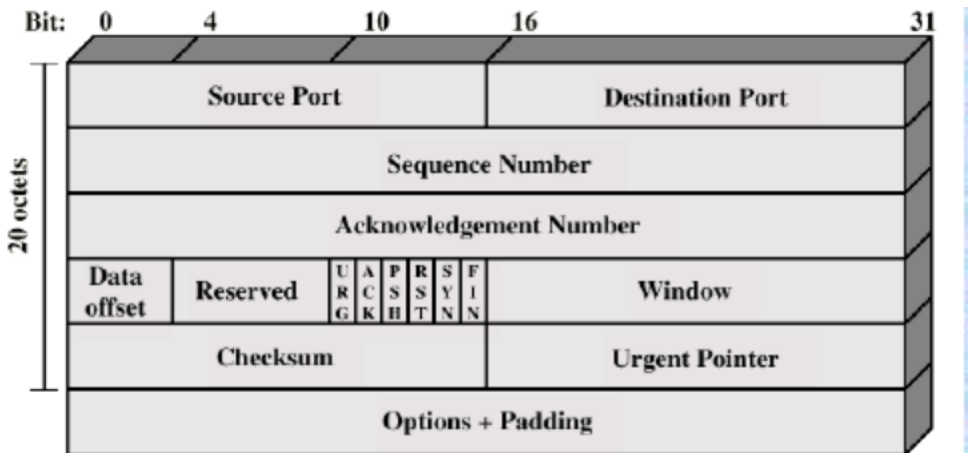


Figure 6-35. (a) Four 512-byte segments sent as separate IP datagrams. (b) The 2048 bytes of data delivered to the application in a single READ call.

- ## The TCP Segment Header:



- **Source Port (16-bits)** - It identifies source port of the application process on the sending device.
- **Destination Port (16-bits)** - It identifies destination port of the application process on the receiving device.
- **Sequence Number (32-bits)** - Sequence number of data bytes of a segment in a session.
- **Acknowledgement Number (32-bits)** - When ACK flag is set, this **number contains the next sequence number of the data byte expected and works as acknowledgement of the previous data received.**
- **Data Offset (4-bits) - This field implies** both, the size of TCP header (32-bit words) and the offset of data in current packet in the whole TCP segment.
- **Reserved (3-bits)** - Reserved for **future use and all are set zero by default.**
- **Flags (1-bit each)**
    - **URG** - It indicates that **Urgent Pointer field** has significant data and should be processed.
    - **ACK** - It indicates that Acknowledgement field has significance. **If ACK is cleared to 0, it indicates that packet does not contain any acknowledgement.**
    - **PSH** - When set, it is a request to the receiving station to PUSH data (as soon as it comes) to the receiving application without buffering it.
    - **RST** - Reset flag has the following features:
        - It is used to refuse an incoming connection.
        - It is used to reject a segment.
        - It is used to **restart a connection.**
    - **SYN** - This flag is used to **set up a connection between hosts.**
    - **FIN** - This flag is used to **release a connection and no more data is exchanged** thereafter. Because packets with SYN and FIN flags have sequence numbers, they are processed in correct order.
- **Windows Size** - This field is used for flow control between two stations and indicates the amount of buffer (in bytes) the receiver has allocated for a segment, i.e. **how much data is the receiver expecting.**
- **Checksum** - This field contains the checksum of Header, Data and Pseudo Headers.
- **Urgent Pointer** - It points to the urgent data byte if URG flag is set to 1.
- **Options** - It facilitates additional options which are not covered by the regular header. Option field is always described in 32-bit words. If this field contains data less than 32-bit, padding is used to cover the remaining bits to reach 32-bit boundary.

• TCP Connection Establishment:

- Connection establishment in a TCP Session is initiated through a three-way handshake protocol.
- To Establish the connection one side server waits for an Incoming connection by executing the LISTEN and ACCEPT primitives.

- Other side client executes a CONNECT primitive specifying the IP address and port to which it wants to connect the maximum
- TCP segment size it is waiting to accept and optionally some user data.
- Transmitter sends Connection Request (SEQ=X) to start a connection with a transmitter message ID 'X'.
- The Receiver replies connection Accepted (seq=Y , Ack=X+1) , to Ack 'x' and establish for its message the identity 'y'.
- Finally the transmitter confirms the connection with connection Accepted (Seq=X+1 , Ack=Y+1) to confirm its own identifier 'x' and
- Accpet the receiver Identity 'Y'

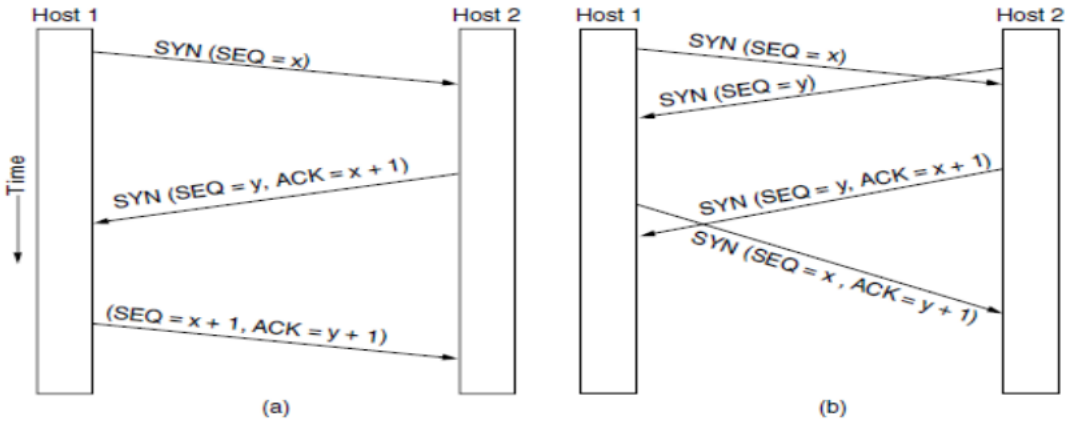


Figure 6-37. (a) TCP connection establishment in the normal case. (b) Simultaneous connection establishment on both sides.

- ## TCP Connection Release:

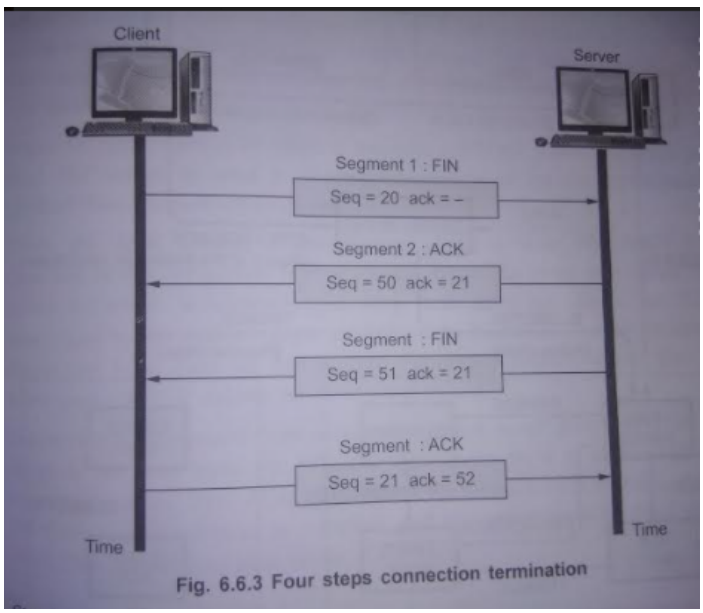


Fig. 6.6.3 Four steps connection termination

- Although TCP connections are full duplex, to understand how connections are released it is best to think of them as a pair of simplex connections.
- Each simplex connection is released independently of its sibling.
- Any two systems to involved to exchange the data can close the connection .

The following steps are followed:

1. The client RCP send the first segment , a FIN (finish) Segment.
2. The server TCP sends the second segment ,an ACK segment , to confirm the receipt of the FIN segment from the client.
3. The server TCP continue sending data in the server client direction . when it does not have any more data to send, it sends the third segment.
4. The client TCP sends the fourth segment , an ACK segment , to confirm the receipt of the FIIN segment from the TCP server.

• TCP Connection Management Modelling:

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Figure 6-38. The states used in the TCP connection management finite state machine.

One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and shoots off a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received. When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked "active close"). When the *ACK* arrives, a transition is made to the state *FIN WAIT 2* and one direction of the connection is closed. When the other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to twice the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection
record.

Now let us examine connection management from the server's viewpoint. The server does a LISTEN and settles down to see who turns up. When a SYN comes in, it is acknowledged and the server goes to the SYN RCVD state. When the server's SYN is itself acknowledged, the three-way handshake is complete and the server goes to the ESTABLISHED state. Data transfer can now occur. When the client is done transmitting its data, it does a CLOSE, which causes a FIN to arrive at the server (dashed box marked "passive close"). The server is then signalled. When it, too, does a CLOSE, a FIN is sent to the client. When the client's acknowledgement shows up, the server releases the connection and deletes the connection record.

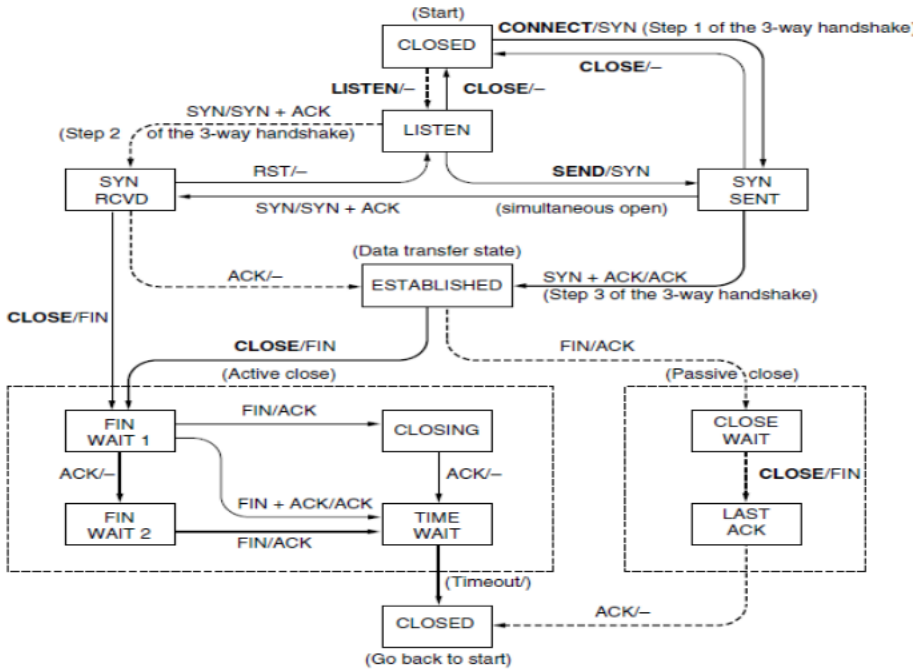


Figure 6-39. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

- ## TCP Sliding Window:

As mentioned earlier, window management in TCP decouples the issues of acknowledgement of the correct receipt of segments and receiver buffer allocation. For example, suppose the receiver has a 4096-byte buffer, as shown in Fig. 6-40. If the sender transmits a 2048-byte segment that is correctly received, the receiver will acknowledge the segment. However, since it now has only 2048 bytes of buffer space (until the application removes some data from the buffer), it will advertise a window of 2048 starting at the next byte expected.

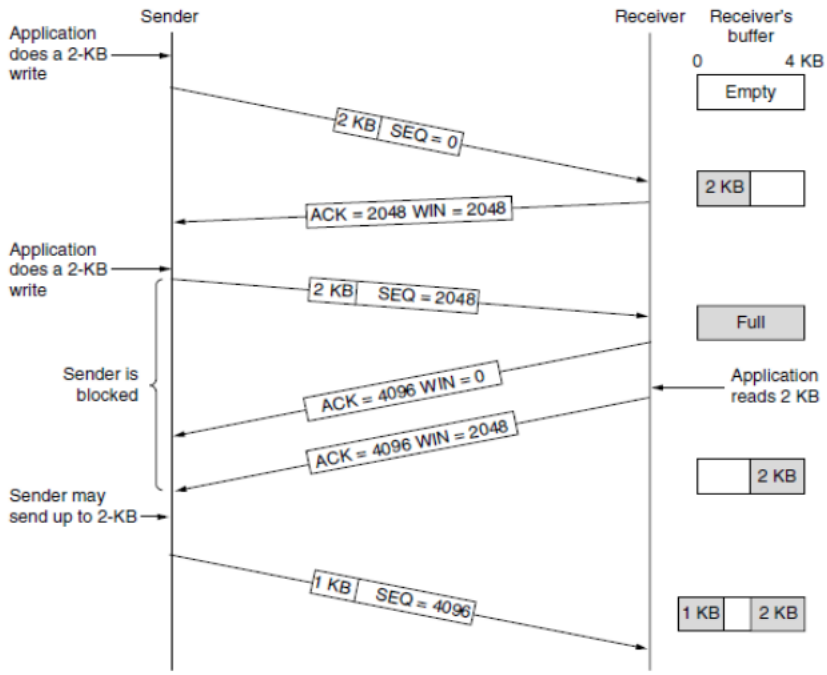


Figure 6-40. Window management in TCP.

Another problem that can degrade TCP performance is the **silly window syndrome** (Clark, 1982). This problem occurs when data are passed to the sending TCP entity in large blocks, but an interactive application on the receiving side reads data only 1 byte at a time. To see the problem, look at Fig. 6-41. Initially, the TCP buffer on the receiving side is full (i.e., it has a window of size 0) and the sender knows this. Then the interactive application reads one character from the TCP stream. This action makes the receiving TCP happy, so it sends a window update to the sender saying that it is all right to send 1 byte. The sender obliges and sends 1 byte. The buffer is now full, so the receiver acknowledges the 1-byte segment and sets the window to 0. This behavior can go on forever.

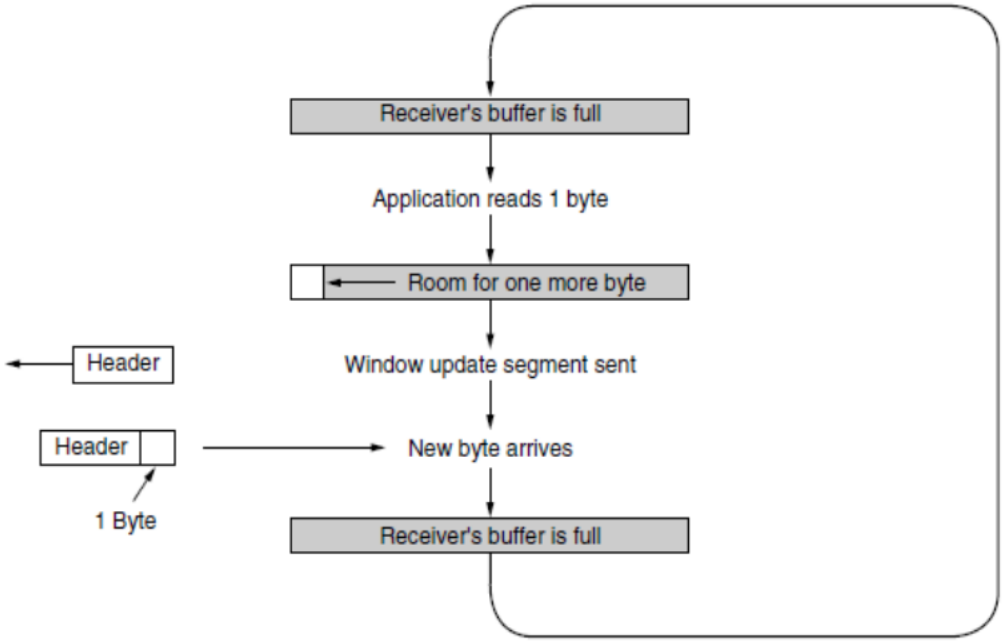


Figure 6-41. Silly window syndrome.

- ## TCP Timer Management:

TCP uses multiple timers (at least conceptually) to do its work. The most important of these is the ***RTO*** *(****Retransmission TimeOut****). When a segment is sent, a retransmission timer is started. If the segment is acknowledged before the timer expires, the timer is stopped. If, on the other hand, the timer goes off before the acknowledgement comes in, the segment is retransmitted (and the timer os started again).*

TCP Congestion Control:

We have saved one of the key functions of TCP for last: congestion control. When the load offered to any network is more than it can handle, congestion builds up. The Internet is no exception. The network layer detects congestion when queues grow large at routers and tries to manage it, if only by dropping packets. It is up to the transport layer to receive congestion feedback from the network layer and slow down the rate of traffic that it is sending into the network.
In the Internet, TCP plays the main role in controlling congestion, as well as the main role in reliable transport. That is why it is such a special protocol.One key takeaway was that a transport protocol using an AIMD (Additive Increase Multiplicative Decrease) control law in response to binary congestion signals from the network would converge to a fair and efficient bandwidth allocation. TCP congestion control is based on implementing this approach using a window and with packet loss as the binary signal. To do so, TCP maintains a ***congestion window*** *whose size is the number of bytes the sender may have in the*

network at any time.

The corresponding rate is the window size divided by the round-trip time of the connection. TCP adjusts the size of the window according to the AIMD rule. Recall that the congestion window is maintained *in addition* to the flow control window, which specifies the number of bytes that the receiver can buffer. Both windows are tracked in parallel, and the number of bytes that may be sent is the smaller of the two windows. Thus, the effective window is the smaller of what the sender thinks is all right and what the receiver thinks is all right. It takes two to tango. TCP will stop sending data if either the congestion or the flow control window is temporarily full. If the receiver says "send 64 KB" but the sender knows that bursts of more than 32 KB clog the network, it will send 32 KB. On the other hand, if the receiver says "send 64 KB" and the sender knows that bursts of up to 128 KB get through effortlessly, it will send the full 64 KB requested. The flow control window was described earlier, and in what follows we will only describe the congestion window.

However, it turns out that we can use small bursts of packets to our advantage. Fig. 6-43 shows what happens when a sender on a fast network (the 1-Gbps link) sends a small burst of four packets to a receiver on a slow network (the 1- Mbps link) that is the bottleneck or slowest part of the path. Initially the four packets travel over the link as quickly as they can be sent by the sender. At the router, they are queued while being sent because it takes longer to send a packet over the slow link than to receive the next packet over the fast link. But the queue is not large because only a small number of packets were sent at once. Note the increased length of the packets on the slow link. The same packet, of 1 KB say, is now longer because it takes more time to send it on a slow link than on a fast one.

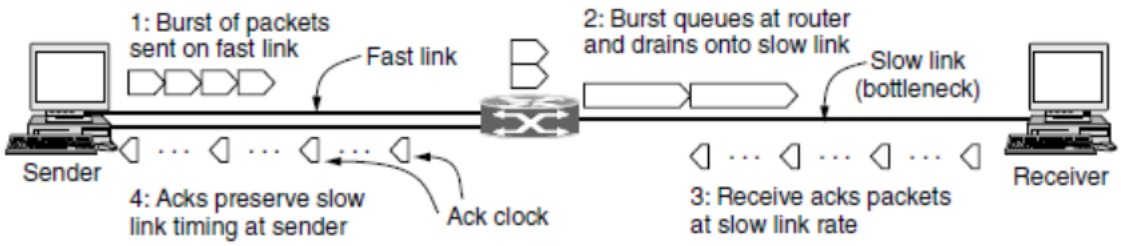


Figure 6-43. A burst of packets from a sender and the returning ack clock.

Eventually the packets get to the receiver, where they are acknowledged. The times for the acknowledgements reflect the times at which the packets arrived at the receiver after crossing the slow link. They are spread out compared to the original packets on the fast link. As these acknowledgements travel over the network and back to the sender they preserve this timing.

This algorithm is called **slow start**, but it is not slow at all—it is exponential growth—except in comparison to the previous algorithm that let an entire flow control window be sent all at once. Slow start is shown in Fig. 6-44. In the first round-trip time, the sender injects one packet into the network (and the receiver receives one packet). Two packets are sent in the next round-trip time, then four packets in the third round-trip time.

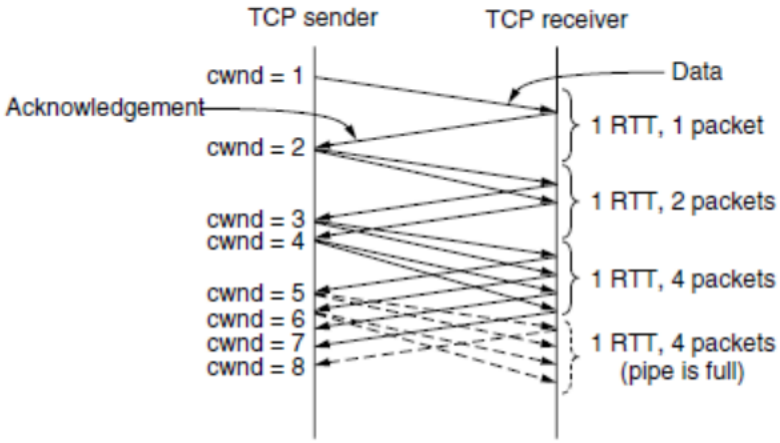


Figure 6-44. Slow start from an initial congestion window of one segment.

Simple Transport Protocol :

To make the ideas discussed so far more concrete, in this section we will study an example transport layer in detail. The abstract service primitives we will use are the connectionoriented primitives of Fig. 6 -2. The choice of these connection-oriented primitives makes the example similar to (but simpler than) the popular TCP protocol.
392

6.3.1 The Example Service Primitives

Our first problem is how to express these transport primitives concretely. CONNECT is easy: we will just have a library procedure connect that can be called with the appropriate parameters necessary to establish a connection. The parameters are the local and remote TSAPs. During the call, the caller is blocked (i.e., suspended) while the transport entity tries to set up the connection. If the connection succeeds, the caller is unblocked and can start transmitting data.
When a process wants to be able to accept incoming calls, it calls listen, specifying a particular TSAP to listen to. The process then blocks until some remote process attempts to establish a connection to its TSAP.
Note that this model is highly asymmetric. One side is passive, executing a listen and waiting until something happens. The other side is active and initiates the connection. An interesting question arises of what to do if the active side begins first. One strategy is to have the connection attempt fail if there is no listener at the remote TSAP. Another strategy is to have the initiator block (possibly forever) until a listener appears.
A compromise, used in our example, is to hold the connection request at the receiving end for a certain time interval. If a process on that host calls listen before the timer goes off, the connection is established; otherwise, it is rejected and the caller is unblocked and given an error return.
To release a connection, we will use a procedure disconnect. When both sides have disconnected, the connection is released. In other words, we are using a symmetric disconnection model.
Data transmission has precisely the same problem as connection establishment: sending is active but receiving is passive. We will use the same solution for data transmission as for connection establishment:

an active call send that transmits data and a passive call receive that blocks until a TPDU arrives. Our concrete service definition therefore consists of five primitives: CONNECT, LISTEN, DISCONNECT, SEND, and RECEIVE. Each primitive corresponds exactly to a library procedure that executes the primitive. The parameters for the service primitives and library procedures are as follows:
connum = LISTEN(local) connum = CONNECT(local, remote) status = SEND(connum, buffer, bytes) status = RECEIVE(connum, buffer, bytes) status = DISCONNECT(connum)
The LISTEN primitive announces the caller's willingness to accept connection requests directed at the indicated TSAP. The user of the primitive is blocked until an attempt is made to connect to it. There is no timeout.
The CONNECT primitive takes two parameters, a local TSAP (i.e., transport address), local, and a remote TSAP, remote, and tries to establish a transport connection between the two. If it succeeds, it returns in connum a nonnegative number used to identify the connection on subsequent calls. If it fails, the reason for failure is put in connum as a negative number. In our simple model, each TSAP may participate in only one transport connection, so a possible reason for failure is that one of the transport addresses is currently in use. Some other reasons are remote host down, illegal local address, and illegal remote address.
The SEND primitive transmits the contents of the buffer as a message on the indicated transport connection, in several units if need be. Possible errors, returned in status, are no connection, illegal buffer address, or negative count

The RECEIVE primitive indicates the caller's desire to accept data. The size of the incoming message is placed in bytes. If the remote process has released the connection or the buffer address is illegal (e.g., outside the user's program), status is set to an error code indicating the nature of the problem.
The DISCONNECT primitive terminates a transport connection. The parameter connum tells which one. Possible errors are connum belongs to another process or connum is not a valid connection identifier. The error code, or 0 for success, is returned in status.
. 6-19. Finally, we have a pointer to the data itself, and an integer giving the number of bytes of data.

Figure 6-19. The network layer packets used in our example.

Network packet	Meaning
CALL REQUEST	Sent to establish a connection
CALL ACCEPTED	Response to CALL REQUEST
CLEAR REQUEST	Sent to release a connection
CLEAR CONFIRMATION	Response to CLEAR REQUEST
DATA	Used to transport data
CREDIT	Control packet for managing the window

Fig. 6-20. Each connection is always in one of seven states, as follows:

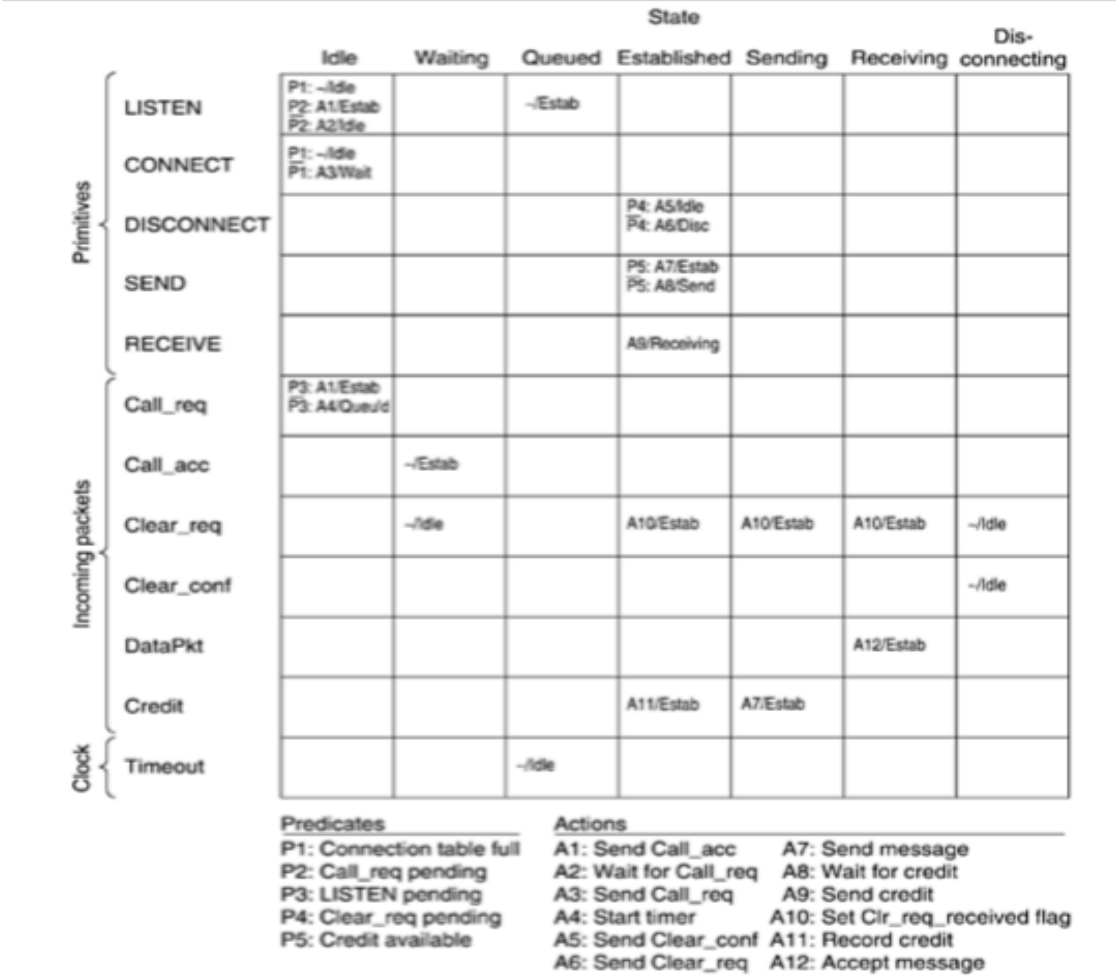
Figure 6-20. An example transport entity.

1. IDLE— Connection not established yet.
2. 2. WAITING— CONNECT has been executed and CALL REQUEST sent.
3. QUEUED— A CALL REQUEST has arrived; no LISTEN yet.
4. ESTABLISHED— The connection has been established.
5. SENDING— The user is waiting for permission to send a packet.
6. RECEIVING— A RECEIVE has been done.
7. DISCONNECTING— A DISCONNECT has been done locally.

6.3.3 The Example as a Finite State Machine

Writing a transport entity is difficult and exacting work, especially for more realistic protocols. To reduce the chance of making an error, it is often useful to represent the state of the protocol as a finite state machine.
We have already seen that our example protocol has seven states per connection. It is also possible to isolate 12 events that can move a connection from one state to another. Five of these events are the five service primitives. Another six are the arrivals of the six legal packet types. The last one is the expiration of the timer. Figure 6-21 shows the main protocol actions in matrix form. The columns are the states and the rows are the 12 events.

Figure 6-21. The example protocol as a finite state machine. Each entry has an optional predicate, an optional action, and the new state. The tilde indicates that no major action is taken. An overbar above a predicate indicates the negation of the predicate. Blank entries correspond to impossible or invalid events.



		State						
		Idle	Waiting	Queued	Established	Sending	Receiving	Dis-connecting
Primitives	LISTEN	P1: -/Idle P2: A1/Estab P2: A2/Idle		-/Estab				
	CONNECT	P1: -/Idle P1: A3/Wait						
	DISCONNECT				P4: A5/Idle P4: A6/Disc			
	SEND				P5: A7/Estab P5: A8/Send			
	RECEIVE				A9/Receiving			
Incoming packets	Call_req	P3: A1/Estab P3: A4/Queu'd						
	Call_acc		-/Estab					
	Clear_req		-/Idle		A10/Estab	A10/Estab	A10/Estab	-/Idle
	Clear_conf							-/Idle
	DataPkt						A12/Estab	
	Credit				A11/Estab	A7/Estab		
Clock	Timeout			-/Idle				

Predicates	Actions	
P1: Connection table full	A1: Send Call_acc	A7: Send message
P2: Call_req pending	A2: Wait for Call_req	A8: Wait for credit
P3: LISTEN pending	A3: Send Call_req	A9: Send credit
P4: Clear_req pending	A4: Start timer	A10: Set Clr_req_received flag
P5: Credit available	A5: Send Clear_conf	A11: Record credit
	A6: Send Clear_req	A12: Accept message

Each entry in the matrix (i.e., the finite state machine) of Fig. 6-21 has up to three fields: a predicate, an action, and a new state. The predicate indicates under which conditions the action is taken. For example, in the upper-left entry, if a LISTEN is executed and there is no more table space (predicate P1), the LISTEN fails and the state does not change. On the other hand, if a CALL REQUEST packet has already arrived for the transport address being listened to (predicate P2), the connection is established immediately. Another possibility is that P2 is false, that is, no CALL REQUEST has come in, in which case the connection remains in the IDLE state, awaiting a CALL REQUEST packet.

It is worth pointing out that the choice of states to use in the matrix is not entirely fixed by the protocol itself. In this example, there is no state LISTENING, which might have been a reasonable thing to have following a LISTEN. There is no LISTENING state because a state is associated with a connection record entry, and no connection record is created by LISTEN. Why not? Because we have decided to use the network layer virtual circuit numbers as the connection identifiers, and for a LISTEN, the virtual circuit number is ultimately chosen by the network layer when the CALL REQUEST packet arrives.

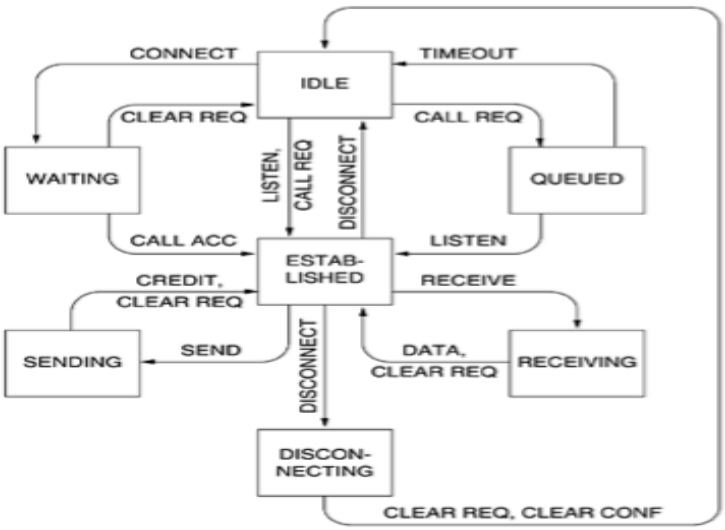
The actions A1 through A12 are the major actions, such as sending packets and starting timers.

Not all the minor actions, such as initializing the fields of a connection record, are listed. If an action involves waking up a sleeping process, the actions following the wakeup also count. For example, if a CALL REQUEST packet comes in and a process was asleep waiting for it, the transmission of the CALL ACCEPT packet following the wakeup counts as part of the action for CALL REQUEST. After each action is performed, the connection may move to a new state, as shown in Fig. 6-21.

The advantage of representing the protocol as a matrix is threefold. First, in this form it is much easier for the programmer to systematically check each combination of state and event to see if an action is required. In production implementations, some of the combinations would be used for error handling. In Fig. 6-21 no distinction is made between impossible situations and illegal ones.

.

Figure 6-22. The example protocol in graphical form. Transitions that leave the connection state unchanged have been omitted for simplicity.



6.4 The Internet Transport Protocols: UDP