UNIT – IV

THE TRANSPORT LAYER

INTRODUCTION:

- The transport layer in the TCP/IP suite is located between the application layer and the network layer. It provides services to the application layer and receives services from the network layer.
- The transport layer acts as a liaison between a client program and a server program, a process-to-process connection. The transport layer is the heart of the TCP/IP protocol suite; it is the end-to-end logical vehicle for transferring data from one point to another in the Internet.
- Communication is provided using a logical connection, which means that the two application layers, which can be located in different parts of the globe, assume that there is an imaginary direct connection through which they can send and receive messages.

I. THE TRANSPORT SERVICE:

a) 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 and/or hardware within the transport layer that does the work are called the **transport entity**.
- The transport entity can be located in the operating system kernel, in a library package bound into network applications, in a separate user process, or even on the network interface card. The first two options are most common on the Internet. The (logical) relationship of the network, transport, and application layers is illustrated in Fig. 4.1.
- Just as there are two types of network service, **connection-oriented** and **connectionless**, there are also two types of transport service. The **connection-oriented transport service** is similar to the connection-oriented network service in many ways. In both cases, connections have three phases: establishment, data transfer, and release. Addressing and flow control are also similar in both layers.
- Furthermore, the ***connectionless transport service*** is also very similar to the connectionless network service. However, note that it can be difficult to provide a connectionless transport service on top of a connection-oriented network service, since it is inefficient to set up a connection to send a single packet and then tear (*meaning run/rip/rush*) it down immediately afterwards.

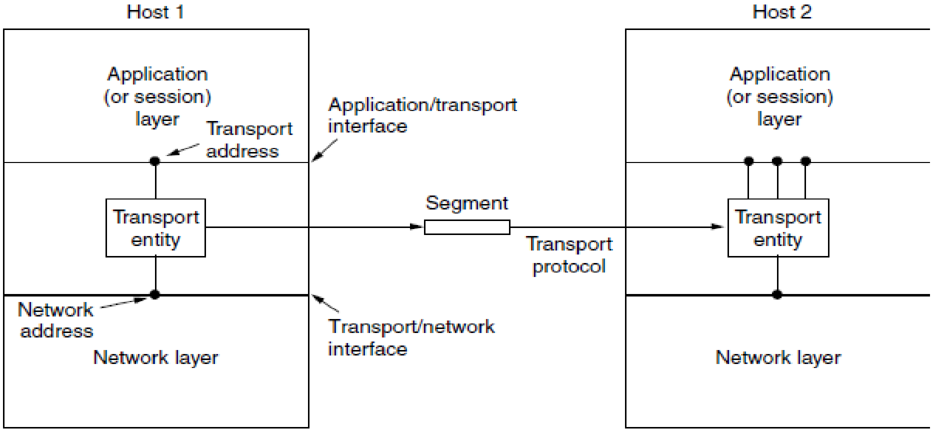
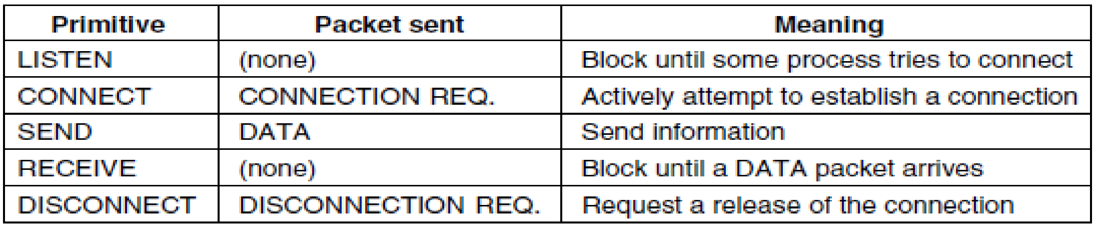


Figure 4.1: The network, transport, and application layers

b) 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 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.
- A second difference between the network service and transport service is that the services are intended for. The network service is used only by the transport entities. Few users write their own transport entities, and thus few users or programs ever (*meaning always/forever/still*) see the bare network service.
- To get an idea of what a transport service might be like, consider the five primitives listed in Fig. 4-2



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 4-2: The primitives for a simple transport service.

- **TPDU** (**Transport Protocol Data Unit**) for messages sent from transport entity to transport entity. TPDU are contained in packets. 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. 4-3

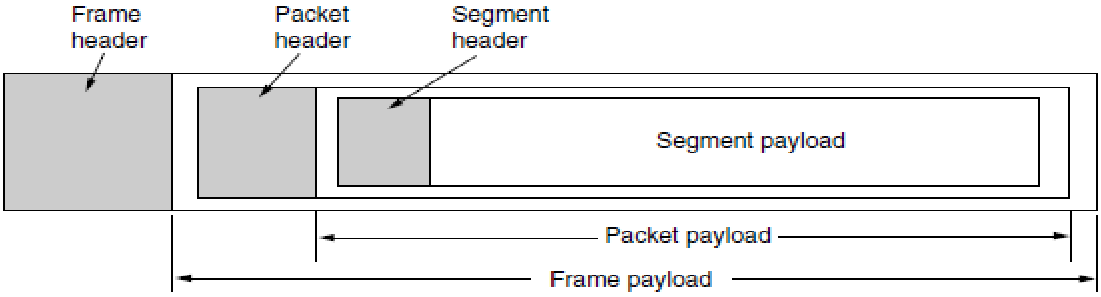


Figure 6-3: Nesting of segments, packets, and frames.

- A state diagram for connection establishment and release for these simple primitives is given in Fig. 4 -4. Each transition is triggered by some event, either a primitive executed by the local transport user or an incoming packet. For simplicity, we assume here that each segment is separately acknowledged. We also assume that a symmetric disconnection model is used, with the client going first. Please note that this model is quite unsophisticated.

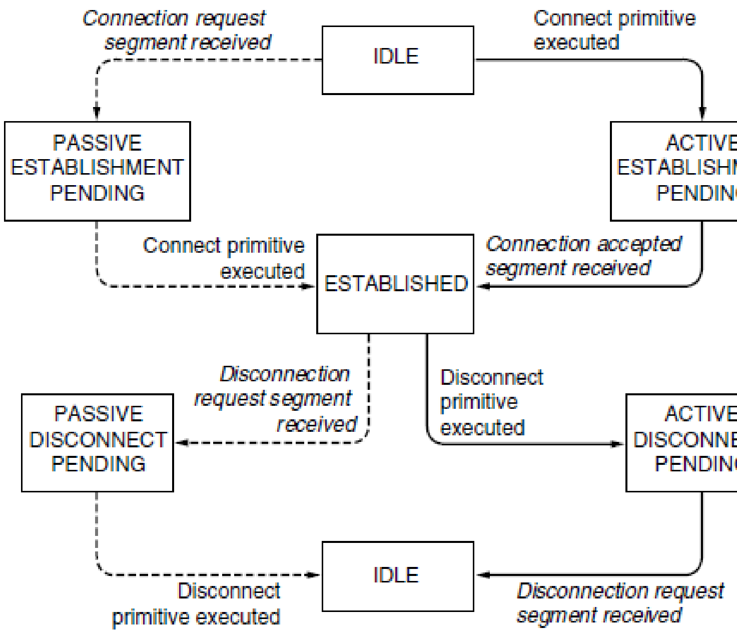
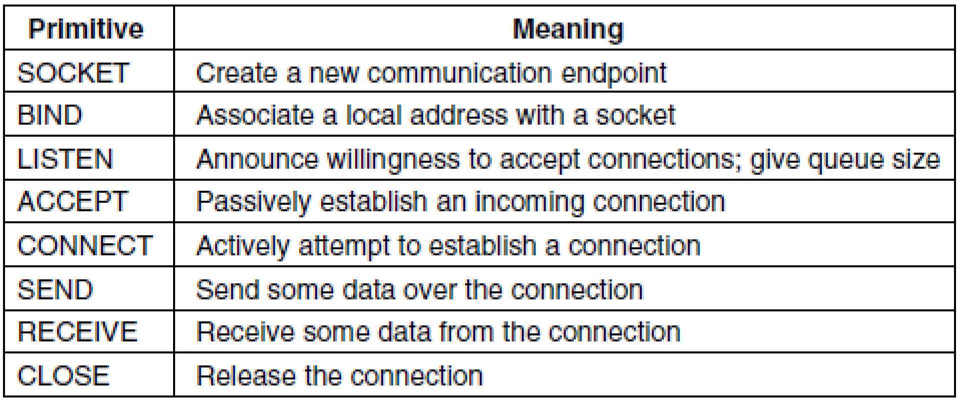

Connection request segment received
IDLE
Connect primitive executed
PASSIVE ESTABLISHMENT PENDING
ACTIVE ESTABLISHMENT PENDING
Connect primitive executed
ESTABLISHED
Connection accepted segment received
Disconnection request segment received
Disconnect primitive executed
PASSIVE DISCONNECT PENDING
ACTIVE DISCONNECT PENDING
IDLE
Disconnect primitive executed
Disconnection request segment received

Figure 4.4: 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.

c) 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.
- The primitives are now widely used for Internet programming on many operating systems, especially UNIX-based systems, and there is a socket-style API for Windows called ''winsock.'' The primitives are listed in Fig. 4.5.



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 4.5: The socket primitives for TCP

- **Note:** An Example of Socket Programming: An Internet File Server

II. ELEMENTS OF TRANSPORT PROTOCOLS:

- The transport service is implemented by a **transport protocol** used between the two transport entities. In some ways, transport protocols resemble the data link protocols. Both have to deal with error control, sequencing, and flow control, among other issues.
- However, significant differences between the two also exist. These differences are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 4.6.

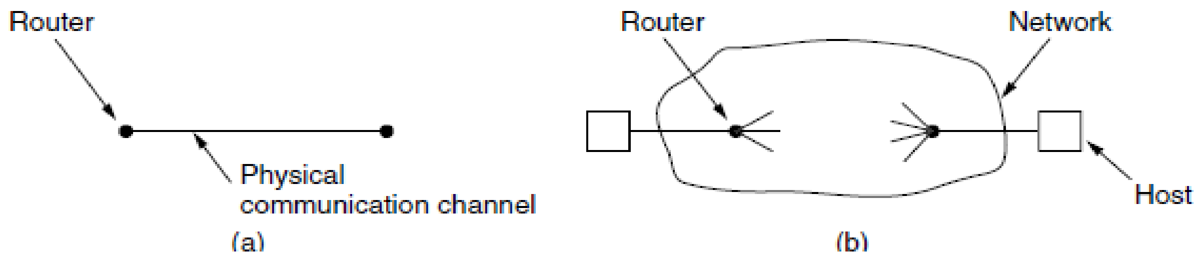


Figure 4.6: Environment of the (a) data link layer (b) transport layer

- 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.
- 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.
- For another thing, the process of establishing a connection over the wire of Fig. 4.6(a) is simple: the other end is always there (unless it has crashed, in which case it is not there). Either way, there is not much to do.
- Even on wireless links, the process is not much different. Just sending a message is sufficient to have it reach all other destinations. If the message is not acknowledged due to an error, it can be resent. In the transport layer, initial connection establishment is complicated.

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 naturally called **NSAPs** (**Network Service Access Points**). IP addresses are examples of NSAPs.
- Figure 4.7 illustrates the relationship between the NSAPs, the TSAPs, and a transport connection.

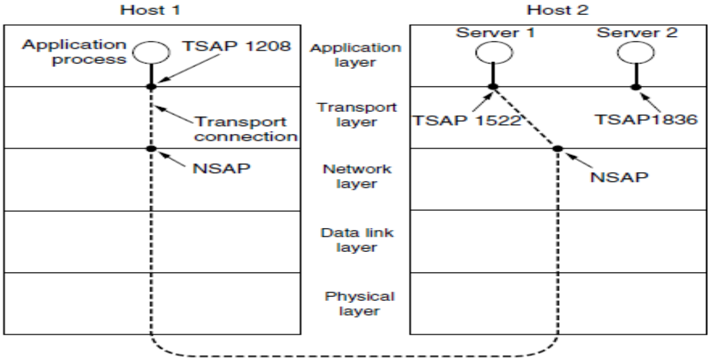


Figure 4.7: TSAPs, NSAPs, and Transport connections

- 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 in figure 4.7.
- A possible scenario for a transport connection is as follows:

  1. A mail server process attaches itself to TSAP 1522 on host 2 to wait for an incoming call. A call such as our LISTEN might be used, for example.

  2. An application process on host 1 wants to send an email message, so it attaches itself to TSAP 1208 and issues a CONNECT request. The request specifies TSAP 1208 on host 1 as the source and TSAP 1522 on host 2 as the destination. This action ultimately results in a transport connection being established between the application process and the server.

  3. The application process sends over the mail message.

  4. The mail server responds to say that it will deliver the message.

  5. The transport connection is released.

<u>2. Connection Establishment:</u>

- 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 behavior causes serious complications.
- Imagine a network that is so congested that acknowledgements hardly ever get back in time and each packet times out and is retransmitted two or three times. Suppose that the network uses datagram inside and that every packet follows a different route.
- Some of the packets might get stuck in a traffic jam inside the network and take a long time to arrive. That is, they may be delayed in the network and pop out much later, when the sender thought that they had been lost.
- The worst possible nightmare is as follows. "A user establishes a connection with a bank, sends messages telling the bank to transfer a large amount of money to the account of a not-entirely-trustworthy person ". Unfortunately, the packets decide to take the scenic route to the destination and go off exploring a remote corner of the network.
- The sender then times out and sends them all again. This time the packets take the shortest route and are delivered quickly so the sender releases the connection.
- Unfortunately, eventually the initial batch of packets finally come out of hiding and arrives at the destination in order, asking the bank to establish a new connection and transfer money (again). The bank has no way of telling that these are duplicates. It must assume that this is a second, independent transaction, and transfers the money again.
- The crux (*meaning root*) of the problem is that the delayed duplicates are thought to be new packets. We cannot prevent packets from being duplicated and delayed. But if and when this happens, the packets must be rejected as duplicates and not processed as fresh packets.
- The problem can be attacked in various ways, none of them very satisfactory. One way is to use throwaway transport addresses. In this approach, each time a transport address is needed, a new one is generated. When a connection is released, the address is discarded and never used again. Delayed duplicate packets then never find their way to a transport process and can do no damage.

  Note: However, this approach makes it more difficult to connect with a process in the first place.
- Another possibility is to give each connection a unique identifier (i.e., a sequence number incremented for each connection established) chosen by the initiating party and put in each segment, including the one requesting the connection.

- After each connection is released, each transport entity can update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request comes in, it can be checked against the table to see if it belongs to a previously released connection.
- Unfortunately, this scheme has a basic flaw: it requires each transport entity to maintain a certain amount of history information indefinitely. This history must persist at both the source and destination machines. Otherwise, if a machine crashes and loses its memory, it will no longer know which connection identifiers have already been used by its peers.
- Instead, we need to take a different tack to simplify the problem. Rather than allowing packets to live forever within the network, we devise a mechanism to kill off aged packets that are still hobbling about.
- Packet lifetime can be restricted to a known maximum using one (or more) of the following techniques:

1. Restricted network design.
2. Putting a hop counter in each packet.
3. Time stamping each packet.

- TCP uses three-way handshake to establish connections in the presence of delayed duplicate control segments as shown in figure 4.8.

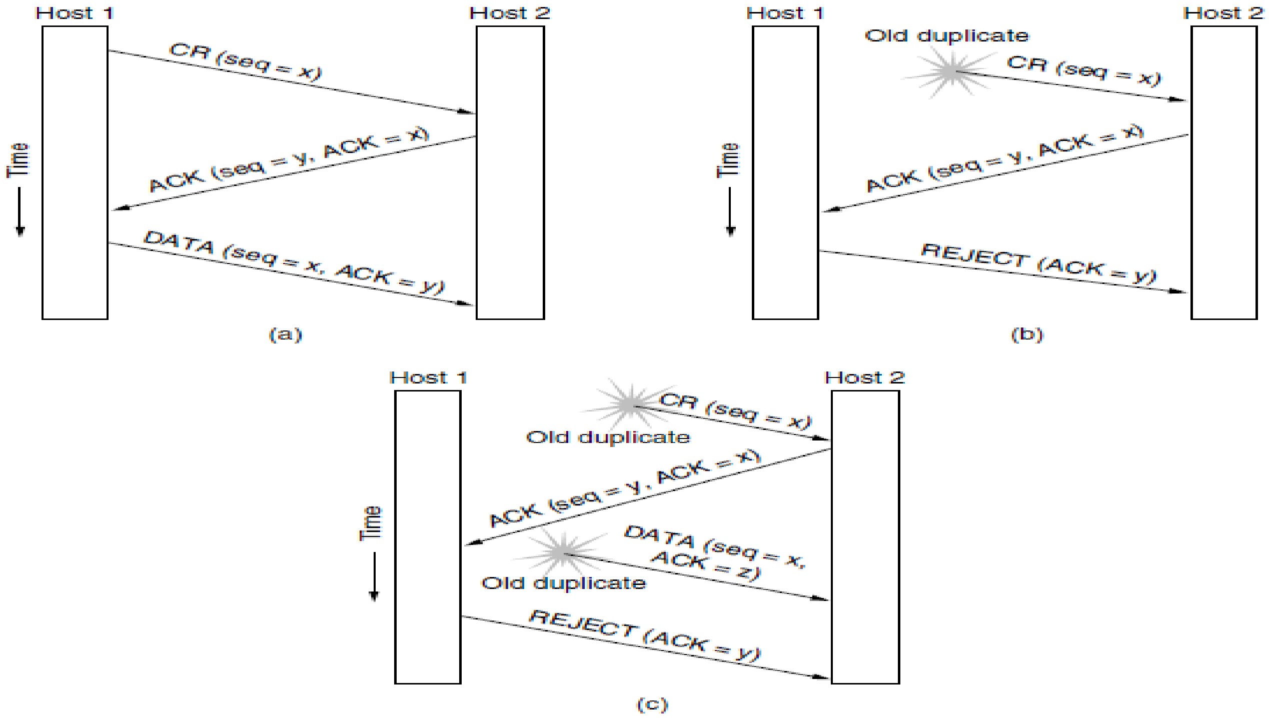


Figure 4.5: Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes Connection Request. (a) Normal operation. (b) Old duplicate connection request appearing out of nowhere. (c) Duplicate connection request and duplicate ack.

3. Connection Release:

- Releasing a connection is easier than establishing one. 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. 4.9. 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.
- 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.’’
- In practice, we can avoid this quandary (*meaning dilemma/difficulty*) by foregoing the need for agreement and pushing the problem up to the transport user, letting each side independently decide when it is done. This is an easier problem to solve.

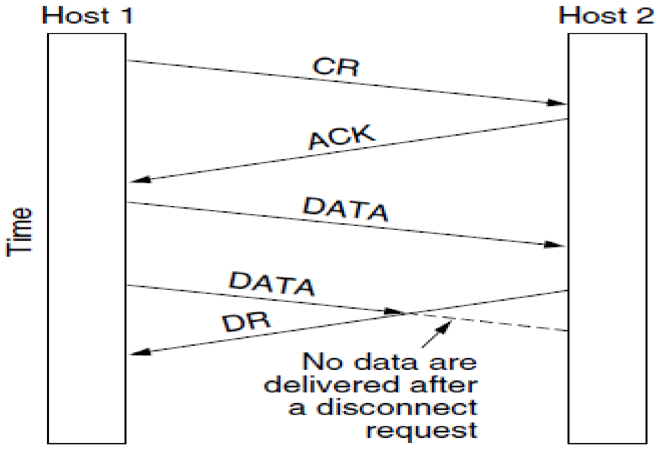


Figure 4.6: Abrupt disconnection with loss of data

- Figure 4.10 illustrates four scenarios of releasing using a three-way handshake. While this protocol is not infallible, it is usually adequate. In Fig. 4.10(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.
- If the final ACK segment is lost, as shown in Fig. 4.10(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. 4.10(c), we see how this works, assuming that the second time no segments is lost and all segments are delivered correctly and on time.
- Our last scenario, Fig. 4.10(d), is the same as Fig. 4.10(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.

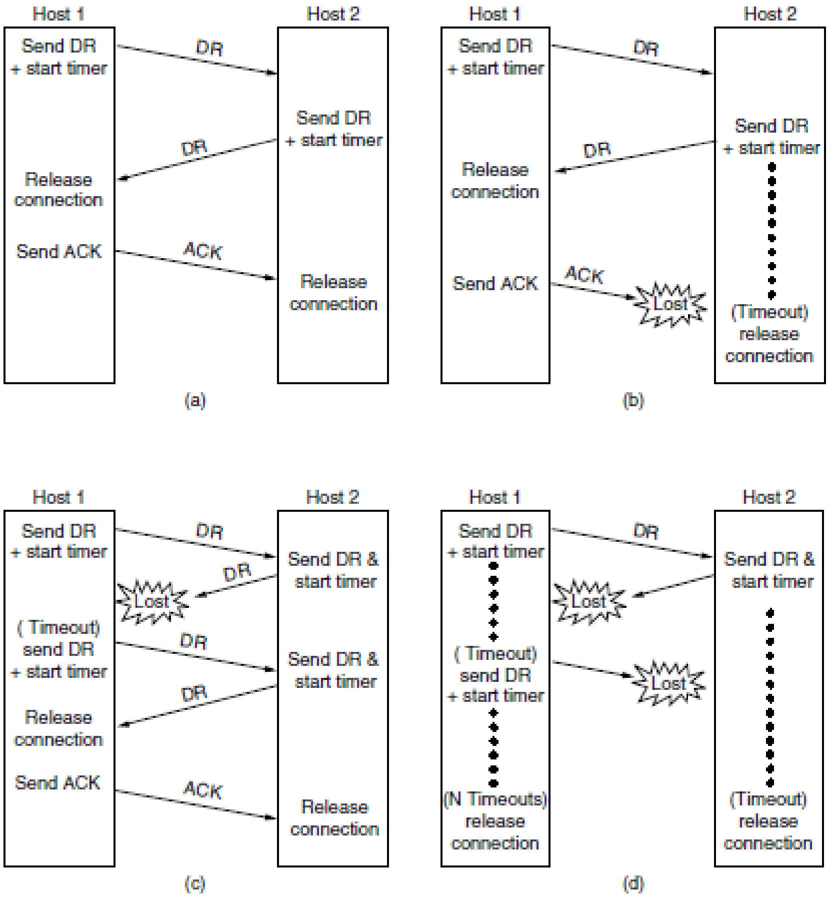


Figure 4.10: Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

4. 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.
- In some ways the flow control problem in the transport layer is the same as in the data link layer, but in other ways it is different.
- The basic similarity is that in both layers a sliding window or other scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver.
- The main difference is that a router usually has relatively few lines, whereas a host may have numerous connections.
- This difference makes it impractical to implement the data link buffering strategy in the transport layer.
- If the network service is unreliable, the sender must buffer all TPDUs sent, just as in the data link layer.
- With reliable network service, other trade-offs become possible.
- In particular, if the sender knows that the receiver always has buffer space, it need not retain copies of the TPDUs it sends.
- If the receiver cannot guarantee that every incoming TPDU will be accepted, the sender will have to buffer anyway.
- In the latter case, the sender cannot trust the network layer's acknowledgement, because the acknowledgement means only that the TPDU arrived, not that it was accepted.

- Even if the receiver has agreed to do the buffering, there still remains the question of the buffer size.
- If most TPDUs are nearly the same size, it is natural to organize the buffers as a pool of identically-sized buffers, with one TPDU per buffer, as in Fig. 4.11(a).
- If there is wide variation in TPDU size, a pool of fixed-sized buffers presents problems.
- If the buffer size is chosen equal to the largest possible TPDU, space will be wasted whenever a short TPDU arrives.
- If the buffer size is chosen less than the maximum TPDU size, multiple buffers will be needed for long TPDUs, with the attendant complexity.

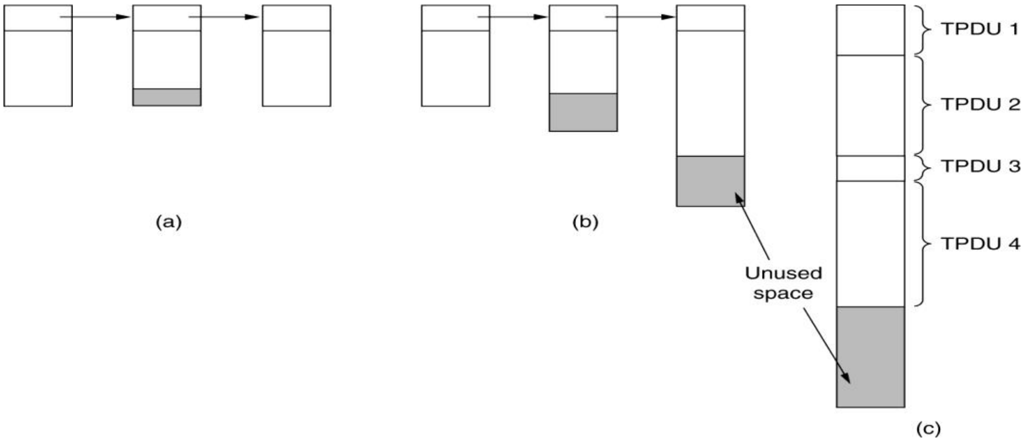


Fig. 4.11: (a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connection.

- Another approach to the buffer size problem is to use variable-sized buffers, as in Fig. 4.11(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. 4.11(c).
- This system also makes good use of memory, provided that all connections are heavily loaded, but is poor if some connections are lightly loaded.

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. 4.12(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. 4.12(b).

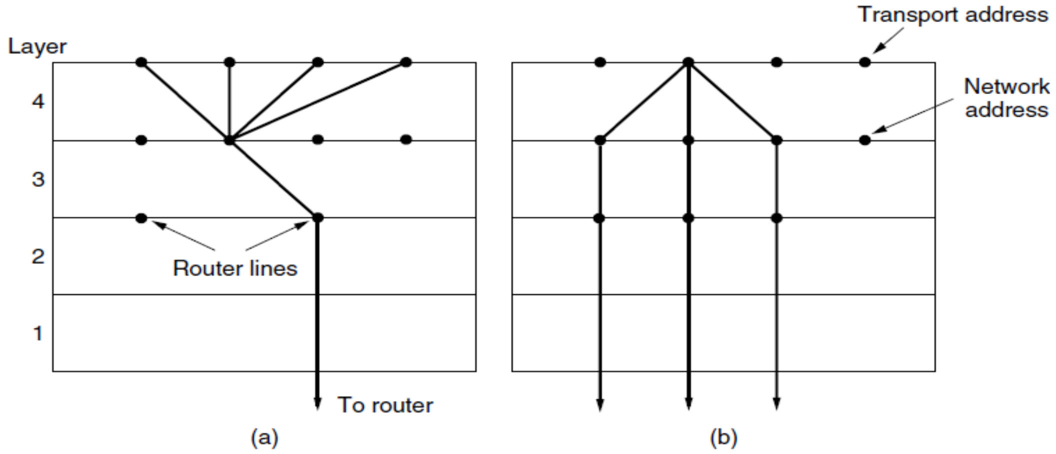


Figure 4.12: (a) Multiplexing (b) Inverse Multiplexing

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 network and 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.

III. CONGESTION CONTROL:

- If the transport entities on many machines send too many packets into the network too quickly, the network will become congested, with performance degraded as packets are delayed and lost.
- Controlling congestion to avoid this problem is the combined responsibility of the network and transport layers. Congestion occurs at routers, so it is detected at the network layer.
- However, congestion is ultimately caused by traffic sent into the network by the transport layer. The only effective way to control congestion is for the transport protocols to send packets into the network more slowly.

1. Desirable Bandwidth Allocation:

- Before we describe how to regulate traffic, we must understand what we are trying to achieve by running a congestion control algorithm. That is, we must specify the state in which a good congestion control algorithm will operate the network.
- The goal is more than to simply avoid congestion. It is to find a good allocation of bandwidth to the transport entities that are using the network. A good allocation will deliver good performance because it uses all the available bandwidth but avoids congestion, it will be fair across competing transport entities, and it will quickly track changes in traffic demands.

i) Efficiency and Power:

- An efficient allocation of bandwidth across transport entities will use all of the network capacity that is available. However, it is not quite right to think that if there is a 100-Mbps link, five transport entities should get 20 Mbps each. They should usually get less than 20 Mbps for good performance.
- This curve and a matching curve for the delay as a function of the offered load are given in Fig. 4.13.

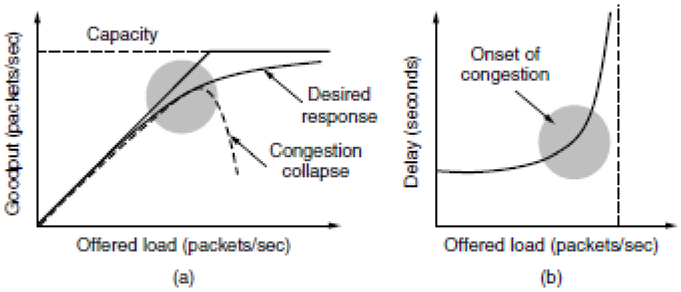


Figure 4.13. (a) Goodput and (b) delay as a function of offered load.

- As the load increases in Fig. 6-19(a) goodput initially increases at the same rate, but as the load approaches the capacity, goodput rises more gradually. This falloff is because bursts of traffic can occasionally mount up and cause some losses at buffers inside the network. If the transport protocol is poorly designed and retransmits packets that have been delayed but not lost, the network can enter congestion collapse. In this state, senders are furiously sending packets, but increasingly little useful work is being accomplished.
- The corresponding delay is given in Fig. 4.13(b) initially the delay is fixed, representing the propagation delay across the network. As the load approaches the capacity, the delay rises, slowly at first and then much more rapidly. This is again because of bursts of traffic that tend to mound up at high load. The delay cannot really go to infinity, except in a model in which the routers have infinite buffers. Instead, packets will be lost after experiencing the maximum buffering delay.
- For both goodput and delay, performance begins to degrade at the onset of congestion.
- To identify it, Klein rock (1979) proposed the metric of **power**, where

$$power = \frac{load}{delay}$$

- Power will initially rise with offered load, as delay remains small and roughly constant, but will reach a maximum and fall as delay grows rapidly.

ii) Max-Min Fairness:

- In the preceding discussion, we did not talk about how to divide bandwidth between different transport senders. This sound like a simple question to answer—give all the senders an equal fraction of the bandwidth—but it involves several considerations.
- Perhaps the first consideration is to ask what this problem has to do with congestion control.
- A second consideration is what a fair portion means for flows in a network. It is simple enough if *N* flows use a single link, in which case they can all have 1/*N* of the bandwidth (although efficiency will dictate that they use slightly less if the traffic is bursty).
- But what happens if the flows have different, but overlapping, network paths? For example, one flow may cross three links, and the other flows may cross one link. The three-link flow consumes more network resources. It might be fairer in some sense to give it less bandwidth than the one-link flows.
- The form of fairness that is often desired for network usage is **max-min fairness**. An allocation is max-min fair if the bandwidth given to one flow cannot be increased without decreasing the bandwidth given to another flow with an allocation that is no larger.

iii) Convergence:

- A final criterion is that the congestion control algorithm converges quickly to a fair and efficient allocation of bandwidth. The discussion of the desirable operating point above assumes a static network environment.
- However, connections are always coming and going in a network, and the bandwidth needed by a given connection will vary over time too. Because of the variation in demand, the ideal operating point for the network varies over time.
- A good congestion control algorithm should rapidly converge to the ideal operating point, and it should track that point as it changes over time. If the convergence is too slow, the algorithm will never be close to the changing operating point. If the algorithm is not stable, it may fail to converge to the right point in some cases, or even oscillate around the right point.

An example of a bandwidth allocation that changes over time and converges quickly is shown in Fig. 4.14. Initially, flow 1 has all of the bandwidth. One second later, flow 2 starts. It needs bandwidth as well. The allocation quickly changes to give each of these flows half the bandwidth. At 4 seconds, a third flow Joins. However, this flow uses only 20% of the bandwidth, which is less than its fair share (which is a third). Flows 1 and 2 quickly adjust, dividing the available bandwidth to each have 40% of the bandwidth. At 9 seconds, the second flow leaves, and the third flow remains unchanged. The first flow quickly captures 80% of the bandwidth. At all times, the total allocated bandwidth is approximately 100%, so that the network is fully used, and competing flows get equal treatment (but do not have to use more bandwidth than they need).

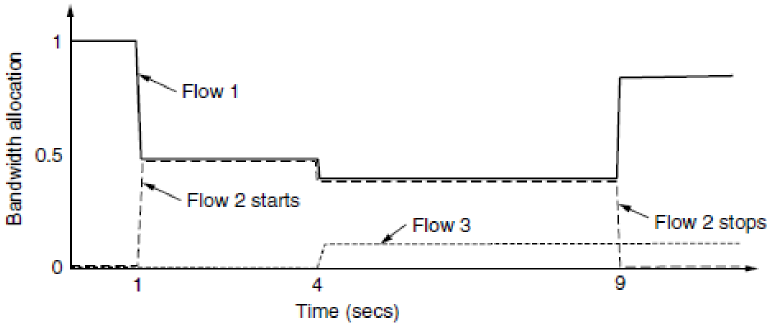


Figure 4.14: Changing bandwidth allocation over time.

2. Regulating the sending rate:

- Now it is time to regulate the sending rates to obtain a desirable bandwidth allocation. The sending rate may be limited by two factors.

A) The first is flow control, in the case that there is insufficient buffering at the receiver.

B) The second is congestion, in the case that there is insufficient capacity in the network.

- In Fig. 4.15, we see this problem illustrated hydraulically. In Fig. 4.15(a), we see a thick pipe leading to a small-capacity receiver. This is a flow-control limited situation. As long as the sender does not send more water than the bucket can contain, no water will be lost.
- In Fig. 4.15(b), the limiting factor is not the bucket capacity, but the internal carrying capacity of the network. If too much water comes in too fast, it will back up and some will be lost (in this case, by overflowing the funnel).
- The way that a transport protocol should regulate the sending rate depends on the form of the feedback returned by the network. Different network layers may return different kinds of feedback. The feedback may be explicit or implicit, and it may be precise or imprecise.

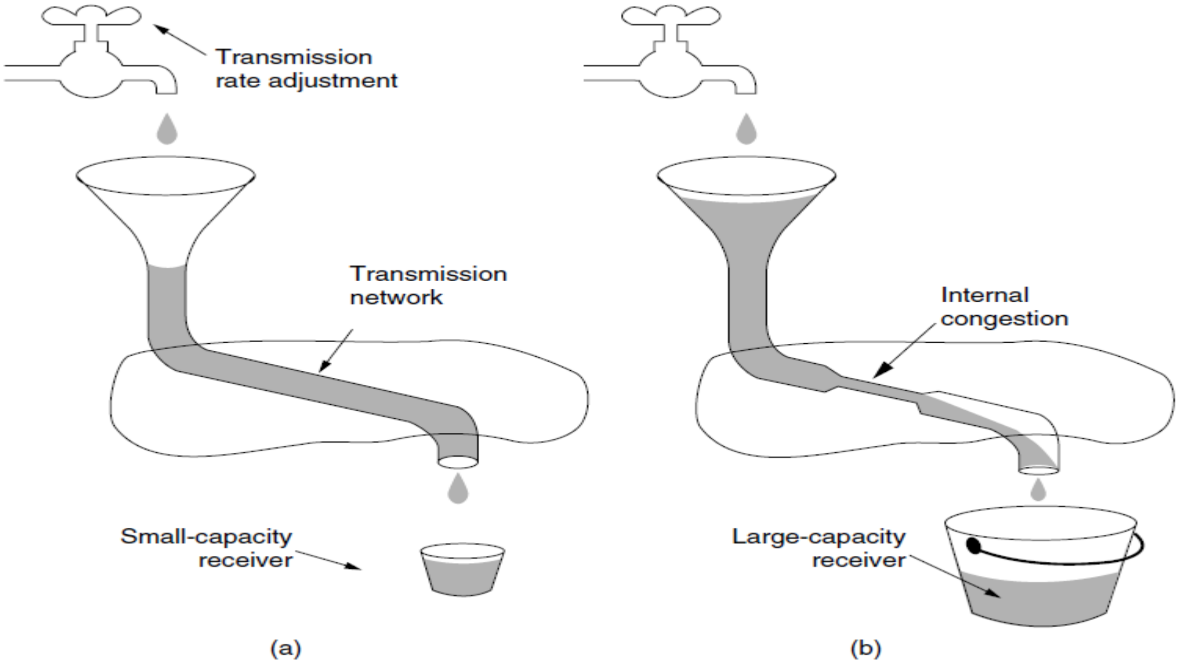


Figure 4.15: (a) a fast network feeding a low-capacity receiver. (b) a slow network feeding a high-capacity receiver.

3. Wireless issues:

- Transport protocols such as TCP that implement congestion control should be independent of the underlying network and link layer technologies. That is a good theory, but in practice there are issues with wireless networks. The main issue is that packet loss is often used as a congestion signal, including by TCP.
- Wireless networks lose packets all the time due to transmission errors. To function well, the only packet losses that the congestion control algorithm should observe are losses due to insufficient bandwidth, not losses due to transmission errors. One solution to this problem is to mask the wireless losses by using retransmissions over the wireless link.

THE INTERNET TRANSPORT PROTOCOLS

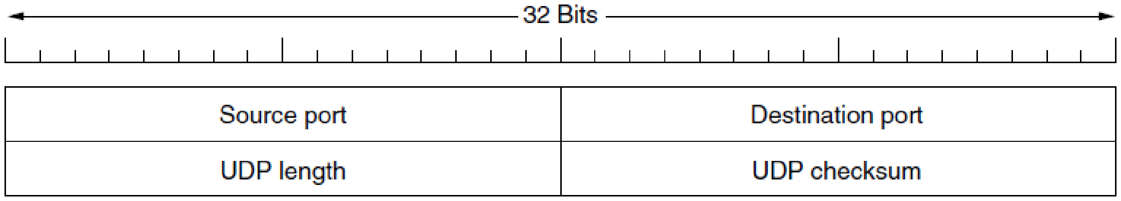
I. UDP (User Datagram Protocol):

- 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.

1. 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 is described in RFC 768.

- UDP transmits **segments** consisting of an 8-byte header followed by the payload. The header is shown in Fig. 4.16. 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. This attachment occurs when the BIND primitive or something similar is used.



Source port	Destination port
UDP length	UDP checksum

Figure 4.16: the UDP header

- Think of ports as mailboxes that applications can rent to receive packets. In fact, the main value of UDP over just using raw IP is the addition of the source and destination ports.
- Without the port fields, the transport layer would not know what to do with each incoming packet. With them, it delivers the embedded segment to the correct application.
- 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. The minimum length is 8 bytes, to cover the header. The maximum length is 65,515 bytes, which is lower than the largest number that will fit in 16 bits because of the size limit on IP packets.
- An optional *Checksum* is also provided for extra reliability. It checksums the header, the data, and a conceptual IP pseudo header. 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.

- The checksum algorithm is simply to add up all the 16-bit words in one's complement and to take the one's complement of the sum.

2. Remote procedure call:

- In a certain sense, sending a message to a remote host and getting a reply back is a lot like making a function call in a programming language. 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. The actual steps in making an RPC are shown in Fig. 4.17.

  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 **marshaling**.

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

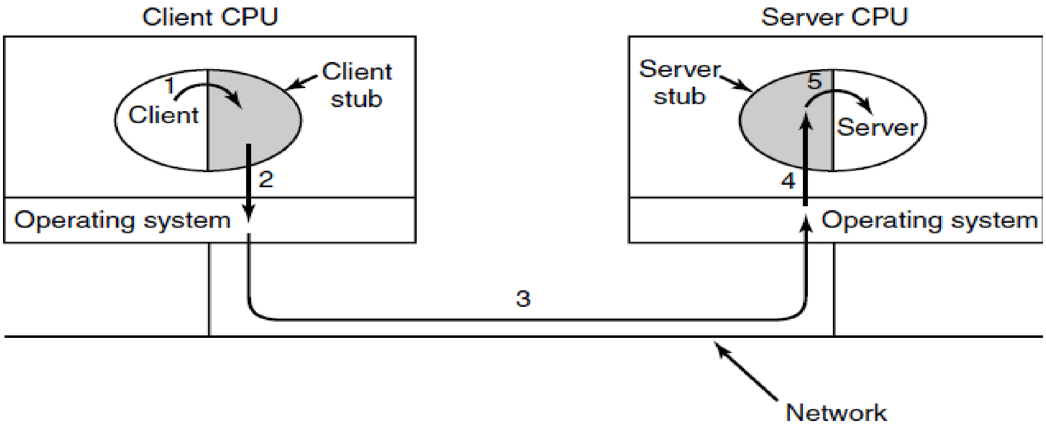


Figure 4.17: Steps in making a remote procedure call, the stubs are shaded

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 unmarshaled parameters.

- The reply traces the same path in the other direction.
- The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way.
- Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual.

3. Real-Time Transport Protocols

- 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. Thus was **RTP** (**Real-time Transport Protocol**) born?
- It is described in RFC 3550 and is now in widespread use for multimedia applications. There are 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.

4. RTP—the Real-Time Transport Protocol:

- The basic function of RTP is to multiplex several real-time data streams onto single stream of UDP packets. The UDP stream can be sent to a single destination (unicasting) or to multiple destinations (multicasting).
- Because RTP just uses normal UDP, its packets are not treated specially by the routers unless some normal IP quality-of-service features are enabled. In particular, there are no special guarantees about delivery, and packets may be lost, delayed, corrupted, etc.
- The RTP format contains several features to help receivers work with multimedia information. The RTP header is illustrated in Fig. 4.18. It consists of three 32-bit words and potentially some extensions.

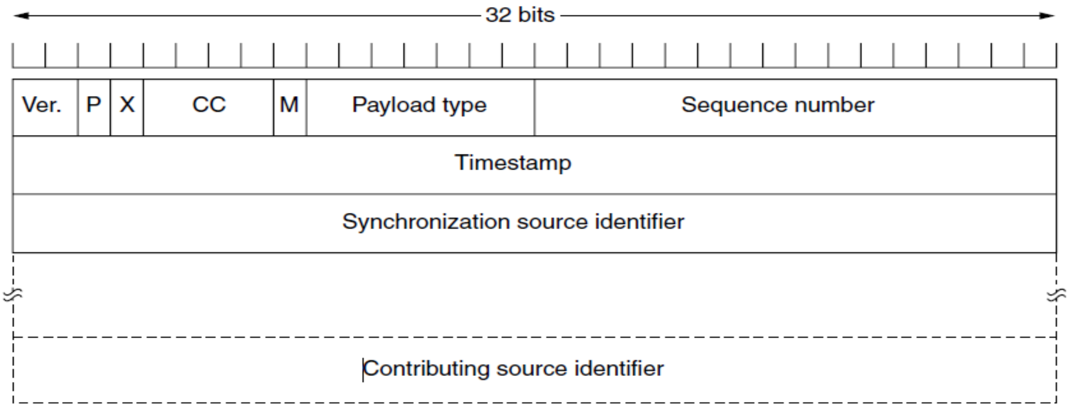

32 bits
Ver.
P
X
CC
M
Payload type
Sequence number
Timestamp
Synchronization source identifier
Contributing source identifier

Figure 4.18: The RTP header

- The first word contains the Version field, which is already at 2.
- The *P* bit indicates that the packet has been padded to a multiple of 4 bytes.
- The X bit indicates that an extension header is present.
- The CC field tells how many contributing sources are present, from 0 to 15.
- The M bit is an application-specific marker bit. It can be used to mark the start of a video frame, the start of a word in an audio channel, or something else that the application understands.
- The Payload type field tells which encoding algorithm has been used (e.g., uncompressed 8-bit audio, MP3, etc.).
- The Sequence number is just a counter that is incremented on each RTP packet sent. It is used to detect lost packets.
- The Timestamp is produced by the stream's source to note when the first sample in the packet was made.
- The Synchronization source identifier tells which stream the packet belongs to. It is the method used to multiplex and demultiplex multiple data streams onto single stream of UDP packets.
- Finally, the Contributing source identifiers, if any, are used when mixers are present.

5. RTCP—the Real-time Transport Control Protocol

- RTP has a little sister protocol (little sibling protocol?) called **RTCP** (**Realtime Transport Control Protocol**). It is defined along with RTP in RFC 3550 and handles feedback, synchronization, and the user interface. It does not transport any media samples.
- The first function can be used to provide feedback on delay, variation in delay or jitter, bandwidth, congestion, and other network properties to the sources. This information can be used by the encoding process to increase the data rate (and give better quality) when the network is functioning well and to cut back the data rate when there is trouble in the network. By providing continuous feedback, the encoding algorithms can be continuously adapted to provide the best quality possible under the current circumstances.
- RTCP also handles interstream synchronization. The problem is that different streams may use different clocks, with different granularities and different drift rates. RTCP can be used to keep them in sync.
- Finally, RTCP provides a way for naming the various sources (e.g., in ASCII text). This information can be displayed on the receiver's screen to indicate who is talking at the moment.

II. TCP (Transmission Control Protocol):

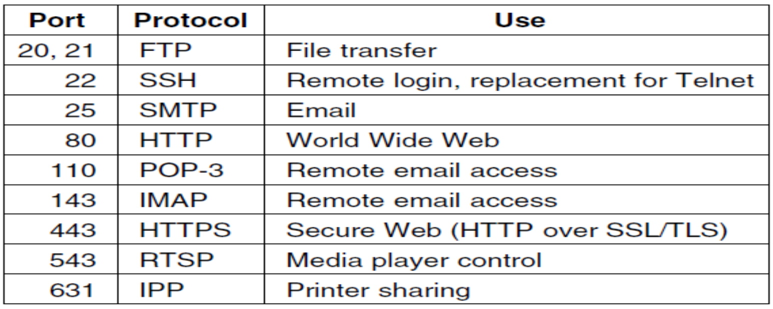
- UDP is a simple protocol and it has some very important uses, such as client server interactions and multimedia, but for most Internet applications, reliable, sequenced delivery is needed. UDP cannot provide this, so another protocol is required. It is called TCP and is the main workhorse of the Internet.

1. Introduction to TCP:

- **TCP** (**Transmission Control Protocol**) was specifically designed to provide a reliable end-to-end byte stream over an unreliable internetwork. An internetwork differs from a single network because different parts may have wildly different topologies, bandwidths, delays, packet sizes, and other parameters.
- TCP was designed to dynamically adapt to properties of the internetwork and to be robust in the face of many kinds of failures. TCP was formally defined in RFC 793 in September 1981.
- As time went on, many improvements have been made, and various errors and inconsistencies have been fixed. To give you a sense of the extent of TCP, the important RFCs are now RFC 793 plus: clarifications and bug fixes in RFC 1122; extensions for high-performance in RFC 1323.
- Selective acknowledgements in RFC 2018; congestion control in RFC 2581; repurposing of header fields for quality of service in RFC 2873; improved retransmission timers in RFC 2988; and explicit congestion notification in RFC 3168. The IP layer gives no guarantee that datagrams will be delivered properly, nor any indication of how fast datagrams may be sent.
- It is up to TCP to send datagrams fast enough to make use of the capacity but not cause congestion, and to time out and retransmit any datagrams that are not delivered. Datagrams that do arrive may well do so in the wrong order; it is also up to TCP to reassemble them into messages in the proper sequence.

2. The TCP Service Model:

- TCP service is obtained by both the sender and the receiver creating end points, called **sockets.** 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. A socket may be used for multiple connections at the same time. In other words, two or more connections may terminate at the same socket.
- Port numbers below 1024 are reserved for standard services that can usually only be started by privileged users (e.g., root in UNIX systems). They are called **well-known ports**.
- For example, any process wishing to remotely retrieve mail from a host can connect to the destination host's port 143 to contact its IMAP daemon. The list of well-known ports is given at *www.iana.org*. Over 700 have been assigned. A few of the better-known ones are listed in Fig. 4.19



Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

.

Figure 4.19: Some assigned ports

- 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.

3. The TCP Protocol:

- A key feature of TCP, and one that dominates the protocol design, is that every byte on a TCP connection has its own 32-bit sequence number. When the Internet began, the lines between routers were mostly 56-kbps leased lines, so a host blasting away at full speed took over 1 week to cycle through the sequence numbers.
- 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. Two limits restrict the segment size. First, each segment, including the TCP header, must fit in the 65,515- byte IP payload. 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, unfragmented 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.
- If this happens, it degrades performance and causes other problems. Instead, modern TCP implementations perform **path MTU discovery** by using the technique outlined in RFC 1191. This technique uses ICMP error messages to find the smallest MTU for any link on the path. TCP then adjusts the segment size downwards to avoid fragmentation.
- The basic protocol used by TCP entities is the sliding window protocol with a dynamic window size. When a sender transmits a segment, it also starts a timer. When the segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exist, and otherwise without) bearing an acknowledgement number equal to the next sequence number it expects to receive and the remaining window size.
- If the sender’s timer goes off before the acknowledgement is received, the sender transmits the segment again

4. The TCP Segment Header:

- Figure 4.20 shows the layout of a TCP segment. Every segment begins with a fixed-format, 20-byte header. The fixed header may be followed by header options. After the options, if any, up to 65,535 − 20 − 20 = 65,495 data bytes may follow, where the first 20 refer to the IP header and the second to the TCP header.
- Segments without any data are legal and are commonly used for acknowledgements and control messages.

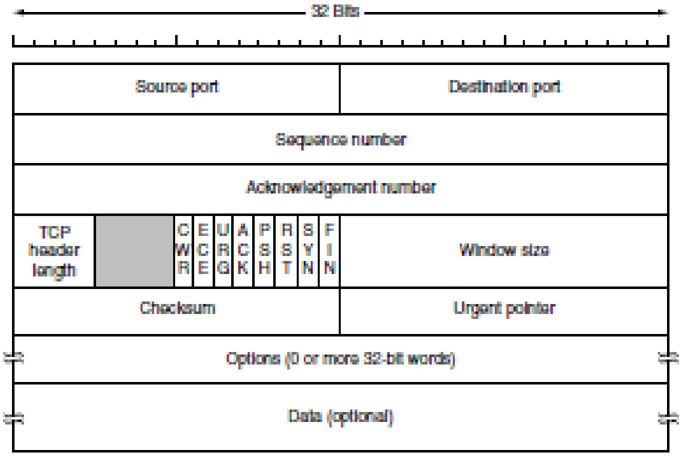


Figure 4.20: The TCP Header

- The **Source port and Destination port** fields identify the local end points of the connection. The source and destination end points together identify the connection. This connection identifier is called a **5 tuple** because it consists of five pieces of information: the protocol (TCP), source IP and source port, and destination IP and destination port.
- The **Sequence number and Acknowledgement number** fields perform their usual functions.
- The **TCP header length** tells how many 32-bit words are contained in the TCP header. This information is needed because the Options field is of variable length, so the header is, too.
- Now come **eight 1-bit flags**. CWR and ECE are used to signal congestion when ECN (Explicit Congestion Notification) is used. CWR is set to signal Congestion Window Reduced from the TCP sender to the TCP receiver so that it knows the sender has slowed down and can stop sending the ECN-Echo.
- URG is set to 1 if the **Urgent pointer** is in use. The Urgent pointer is used to indicate a byte offset from the current sequence number at which urgent data are to be found.
- The **ACK** bit is set to 1 to indicate that the Acknowledgement number is valid. This is the case for nearly all packets. If ACK is 0, the segment does not contain an acknowledgement, so the Acknowledgement number field is ignored.
- The **PSH** bit indicates PUSHed data. The receiver is hereby kindly requested to deliver the data to the application upon arrival and not buffer it until a full buffer has been received (which it might otherwise do for efficiency).
- The **RST** bit is used to abruptly reset a connection that has become confused due to a host crash or some other reason.
- The **SYN** bit is used to establish connections. The FIN bit is used to release a connection.
- The Window size field tells how many bytes may be sent starting at the byte acknowledged.
- A Checksum is also provided for extra reliability. The Options field provides a way to add extra facilities not covered by the regular header.

5. TCP Connection Establishment:

- Connections are established in TCP by means of the three-way handshake. To establish a connection, one side, say, the server passively waits for an incoming connection by executing the LISTEN and ACCEPTS primitives in that order, either specifying a specific source or nobody in particular.

- The other side, say, the client, executes a CONNECT primitive, specifying the IP address and port to which it wants to connect, the maximum TCP segment size it is willing to accept, and optionally some user data (e.g., a password). The CONNECT primitive sends a TCP segment with the *SYN* bit on and *ACK* bit off and waits for a response.
- When this segment arrives at the destination, the TCP entity their checks to see if there is a process that has done a LISTEN on the port given in the *Destination port* field. If not, it sends a reply with the *RST* bit on to reject the connection.
- If some process is listening to the port, that process is given the incoming TCP segment. It can either accept or reject the connection. If it accepts, an acknowledgement segment is sent back. The sequence of TCP segments sent in the normal case is shown in Fig. 4.21(a). Note that a *SYN* segment consumes 1 byte of sequence space so that it can be acknowledged unambiguously.

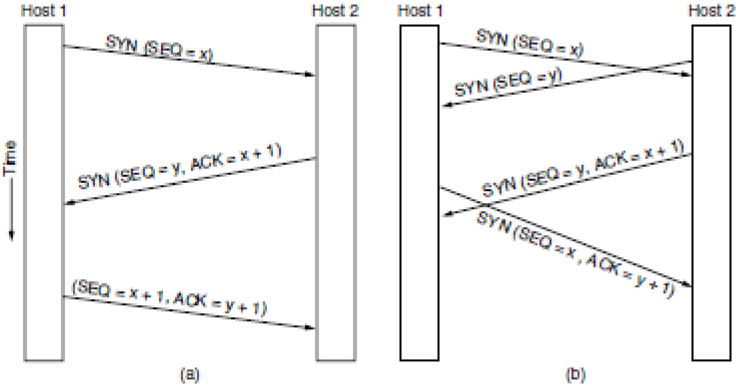

Host 1
Host 2
SYN (SEQ = x)
Time
SYN (SEQ = y, ACK = x + 1)
(SEQ = x + 1, ACK = y + 1)
(a)
Host 1
Host 2
SYN (SEQ = x)
SYN (SEQ = y)
SYN (SEQ = y, ACK = x + 1)
SYN (SEQ = x, ACK = y + 1)
(b)

Figure 4.21 (a) TCP connection establishment in the normal case. (b) Simultaneous Connection establishment on both sides.

- In the event that two hosts simultaneously attempt to establish a connection between the same two sockets, the sequence of events is as illustrated in Fig.4.21 (b). The result of these events is that just one connection is established, not two, because connections are identified by their end points. If the first setup results in a connection identified by (*x*, *y*) and the second one does too, only one table entry is made, namely, for (*x*, *y*).

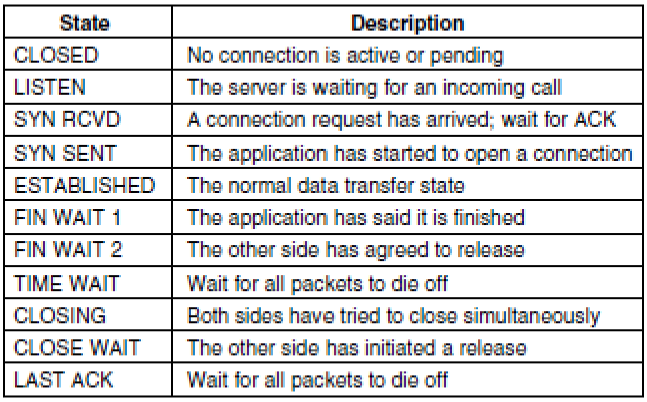
6. TCP Connection Release:

- 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.
- To release a connection, either party can send a TCP segment with the *FIN* bit set, which means that it has no more data to transmit. When the *FIN* is acknowledged, that direction is shut down for new data.
- Data may continue to flow indefinitely in the other direction, however. When both directions have been shut down, the connection is released.

7. TCP Connection Management Modeling:

- The steps required establishing and release connections can be represented in a finite state machine with the 11 states listed in Fig. 4.22. In each state, certain events are legal. When a legal event happens, some action may be taken. If some other event happens, an error is reported.

- Each connection starts in the CLOSED state. It leaves that state when it does either a passive open (LISTEN) or an active open (CONNECT). If the other side does the opposite one, a connection is established and the state becomes ESTABLISHED. Connection release can be initiated by either side. When it is complete, the state returns to CLOSED.



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 4.22: The states used in the TCP connection management finite state machine.

- TCP connection management finite state machine is shown in Fig. 4.23. The common case of a client actively connecting to a passive server is shown with heavy lines—solid for the client, dotted for the server. The lightface lines are unusual event sequences.

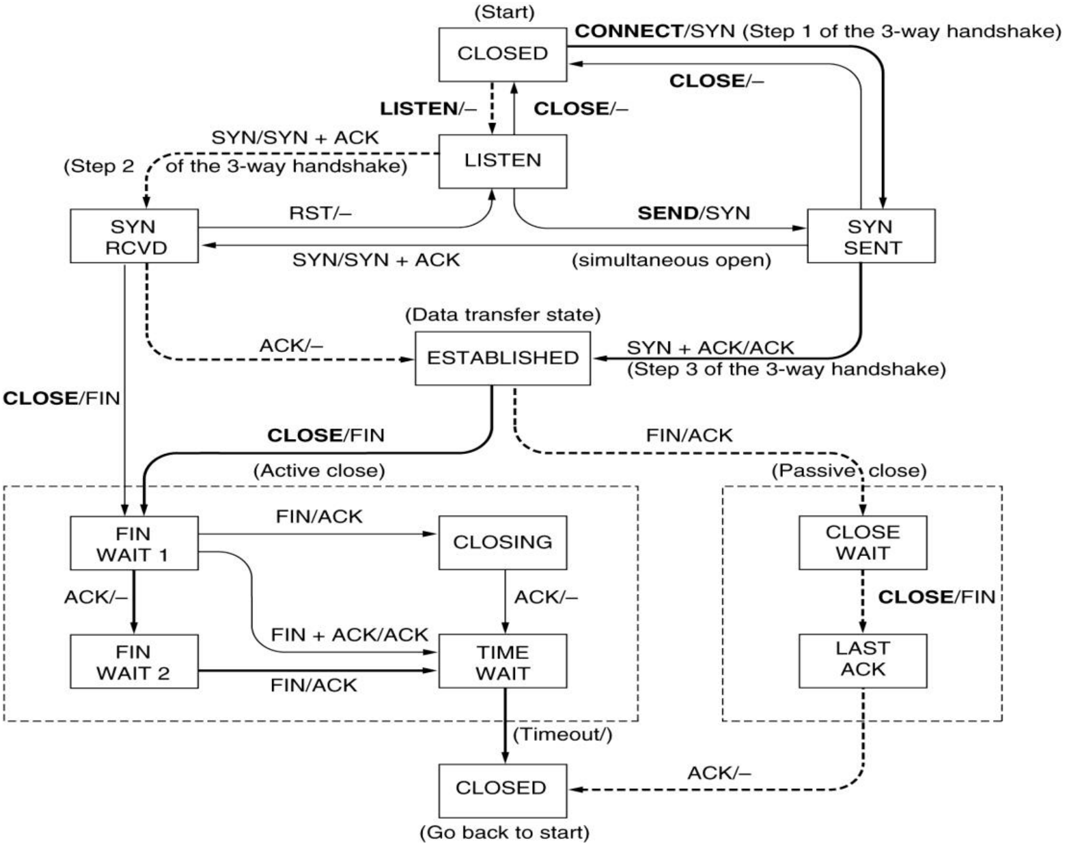


Figure 4.23. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labeled with the event causing it and the action resulting from it, separated by a slash.

8. TCP Sliding Window:

- **Window probe** is a packet sent by the sender, who can send a 1-byte segment to force the receiver to reannounce the next byte expected and the window size.
- **Delayed acknowledgements** is an optimization, where the idea is to delay acknowledgements and window updates for up to 500 msec in the hope of acquiring some data on which to hitch a free ride.

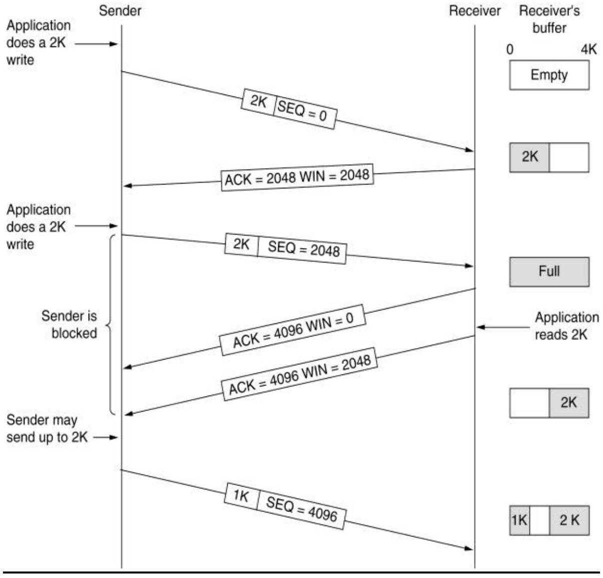


Figure 4.24: Window management in TCP.

- **Nagle's algorithm** is a way to reduce the bandwidth wastage by a sender that sends multiple short packets (e.g., 41-byte packets containing 1 byte of data).
- When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged. Then send all the buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
- **Silly window** syndrome is a problem that 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.

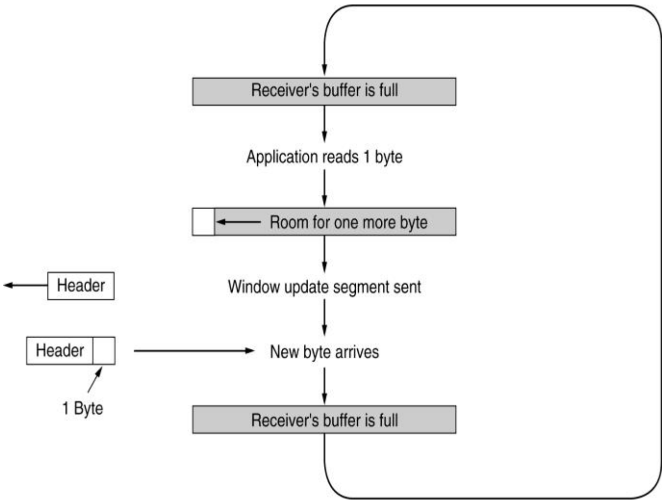


Figure 4.25: Silly window syndrome.

- Clark's solution is to prevent the receiver from sending a window update for 1 byte. Instead, it is forced to wait until it has a decent amount of space available and advertise that instead.

- Nagle's algorithm and Clark's solution to the silly window syndrome are complementary. Nagle was trying to solve the problem caused by the sending application delivering data to TCP a byte at a time. Clark was trying to solve the problem of the receiving application sucking the data up from TCP a byte at a time.
- Both solutions are valid and can work together. The goal is for the sender not to send small segments and the receiver not to ask for them.

9. TCP Timer Management:

- **Retransmission timer:** 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 started again).
- **Persistence timer** is designed to prevent a deadlock situation where, the sender keeps waiting for a window update from the receiver, which is lost. When the persistence timer goes off, the sender transmits a probe to the receiver. The response to the probe gives the window size.
- **Keep alive timer:** When a connection has been idle for a long time, the keep alive timer may go off to cause one side to check whether the other side is still there. If it fails to respond, the connection is terminated.

10. TCP Congestion Control:

- To deal with the two problems of receiver's capacity and network capacity, each sender maintains two windows: the window the receiver has granted and a second window, the congestion window.
- Each reflects the number of bytes the sender may transmit. The number of bytes that may be sent is the minimum of the two windows.
- When a connection is established, the sender initializes the congestion window to the size of the maximum segment in use on the connection. It then sends one maximum segment. Each burst acknowledged doubles the congestion window.
- The congestion window keeps growing exponentially until either a timeout occurs or the receiver's window is reached. This algorithm is called slow start.
- Internet congestion control algorithm uses a third parameter, the threshold, initially 64 KB, in addition to the receiver and congestion windows. When a timeout occurs, the threshold is set to half of the current congestion window, and the congestion window is reset to one maximum segment.

11. The Future of TCP:

- Another proposal is SCTP (Stream Control Transmission Protocol). Its features include message boundary preservation, multiple delivery modes (e.g., unordered delivery), multihoming (backup destinations), and selective acknowledgements

III. PERFORMANCE PROBLEMS IN COMPUTER NETWORKS:

- Some performance problems, such as congestion, are caused by temporary resource overloads. If more traffic suddenly arrives at a router than the router can handle, congestion will build up and performance will suffer.
- Performance also degrades when there is a structural resource imbalance. For example, if a gigabit communication line is attached to a low-end PC, the poor host will not be able to process the incoming packets fast enough and some will be lost. These packets will eventually be retransmitted, adding delay, wasting bandwidth, and generally reducing performance.
- Overloads can also be synchronously triggered. As an example, if a segment contains a bad parameter, in many cases the receiver will thoughtfully send back an error notification.
- Another tuning issue is setting timeouts. When a segment is sent, a timer is set to guard against loss of the segment. If the timeout is set too short, unnecessary retransmissions will occur, clogging the wires. If the timeout is set too long, unnecessary delays will occur after a segment is lost.

IV. NETWORK PERFORMANCE MEASUREMENT:

- When a network performs poorly, its users often complain to the folks running it, demanding improvements. To improve the performance, the operators must first determine exactly what is going on. To find out what is really happening, the operators must make measurements.
- Measurements can be made in different ways and at many locations (both in the protocol stack and physically). The most basic kind of measurement is to start a timer when beginning some activity and see how long that activity takes.
- Other measurements are made with counters that record how often some event has happened (e.g., number of lost segments).
- Measuring network performance and parameters has many potential pitfalls. We list a few of them here. Any systematic attempt to measure network performance should be careful to avoid these.

1) Make Sure That the Sample Size Is Large Enough

- Do not measure the time to send one segment, but repeat the measurement, says, one million times and takes the average.

2) Make Sure That the Samples Are Representative

- Ideally, the whole sequence of one million measurements should be repeated at different times of the day and the week to see the effect of different network conditions on the measured quantity.

3) Caching Can Wreak Havoc with Measurements

- Repeating a measurement many times will return an unexpectedly fast answer if the protocols use caching mechanisms.

4) Be Sure That Nothing Unexpected Is Going On during Your Tests

- Making measurements at the same time that some user has decided to run a video conference over your network will often give different results than if there is no video conference.

5) Be Careful When Using a Coarse-Grained Clock

- Computer clocks function by incrementing some counter at regular intervals.

6) Be Careful about Extrapolating the Results:

- Suppose that you make measurements with simulated network loads running from 0 (idle) to 0.4 (40% of capacity).