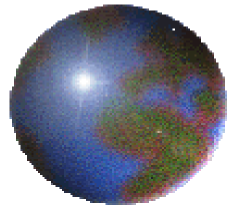


CEN445
Network Protocols & Algorithms

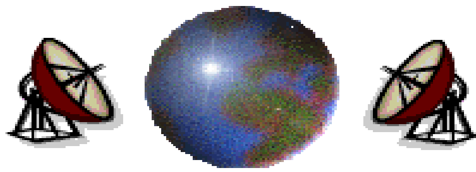
Chapter 6

Transport Layer

Prepared by

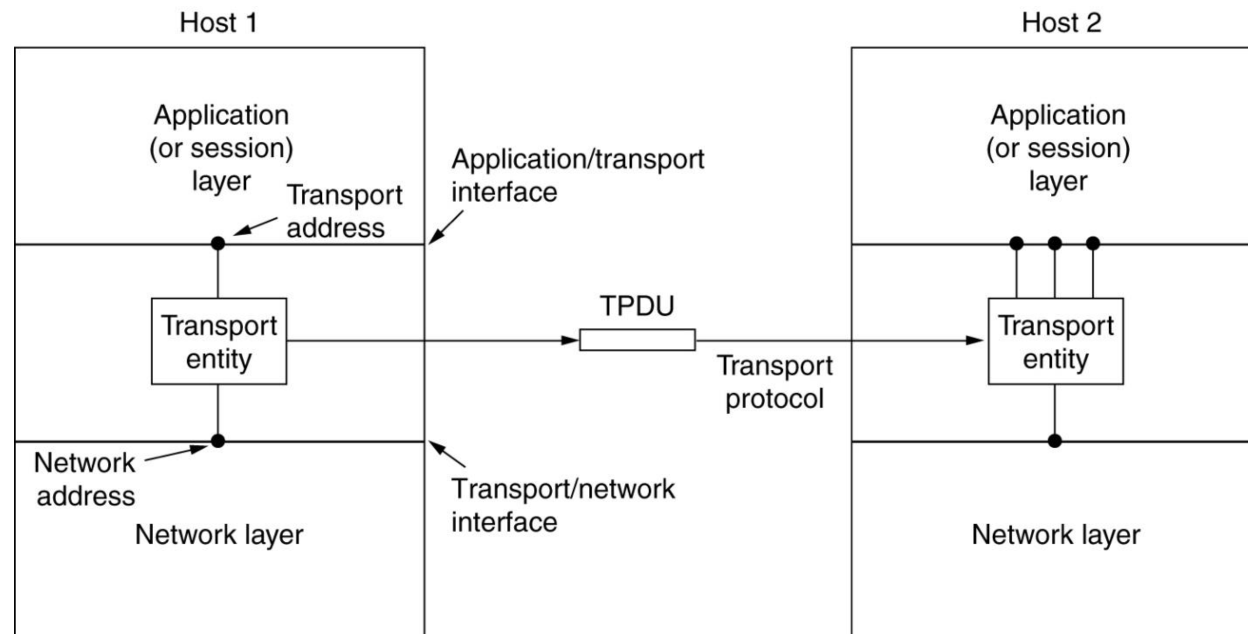
Dr. Mohammed Amer Arafah

Summer 2010

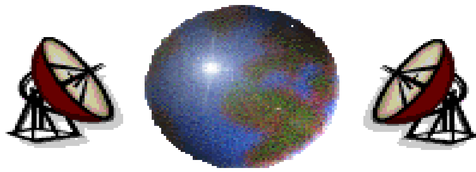


Services Provided to the Upper Layers

- The ultimate goal of the transport layer is to provide efficient, reliable, and cost effective service to its processes in the application layer.
- The hardware/software within the transport layer that does the work is called the **transport entity**.

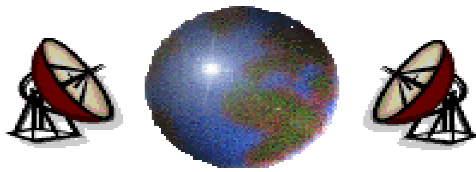


The network, transport, and application layers.



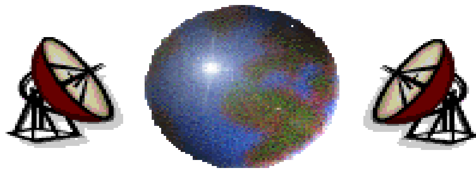
Services Provided to the Upper Layers

- Just as there are two types of network service, **connection-oriented** and **connectionless**, there are also the same two types of transport service.
- If the transport layer is so similar to the network layer service, why are there two distinct layers?
- The network layer is part of communication subnet. What happens if the network layer offers connection-oriented service but is unreliable? Suppose that it frequently loses packets, what happens if routers crash from time to time?
- The users have no control over the subnet, so they cannot solve the problem of poor service by using better routers or putting more error handling in the data link layer. The only possibility is to put another layer on the top of the network layer that improves the quality of service.



Services Provided to the Upper Layers

- If a transport layer is informed through a long transmission that its network connection has been abruptly terminated, it can set up a new connection to the remote transport entity. Then it can send a query to its peer asking which data arrived and which did not, and then pick up from where it left off.
- The existence of the transport layer makes it possible for the transport service to be more reliable than the underlying network service.
- The transport service primitives can be designed to be independent of the network service primitives which may vary from network to network.
- Because of the transport layer, it is possible for the application programs to be written using a standard set of primitives, and to have these programs work on a wide variety of networks.

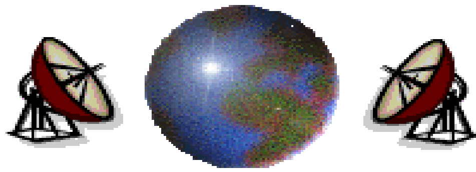


Quality of Service (QoS)

- QoS can be characterized by a number of specific parameters.
- The transport service may allow the user to specify preferred, acceptable, and minimum values for various service parameters at the time a connection is set up.
- It is up the transport layer to examine these parameters, and depending on the kind of network service(s), determine whether it can provide the required service.

Connection Establishment Delay
Connection Establishment Failure Probability
Throughput
Transit Delay
Residual Error Ratio
Protection
Priority
Resilience

Typical Transport Layer quality of Service Parameter



Services Provided to the Upper Layers

Connection Establishment Delay:

- It is the amount of time elapsing between a transport connection being requested and the confirmation being received by the user of the transport service.

Connection Establishment Failure probability:

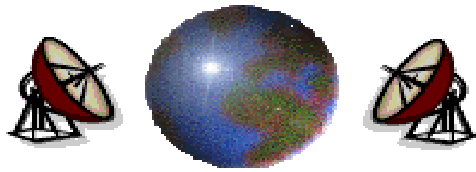
- It is the chance of the connection not being established within the maximum establishment delay.

Throughput:

- It measures the number of bytes of user data transferred per second.

Transit delay:

- It measures the time between a message being sent by a transport user on the source machine and its being received by the transport user on the destination machine.



Services Provided to the Upper Layers

Residual Error Ratio:

- It measures the number of lost or garbled messages as a fraction of the total sent.

Protection:

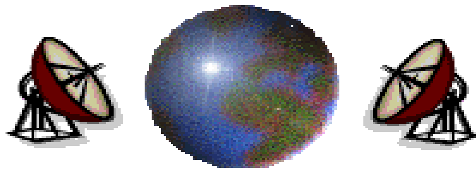
- It provides a way for the transport user to specify the interest of having the transport layer provide protection against unauthorized parties.

Priority:

- It provides a way for a transport user to indicate that some of its connections are more important than other ones, and in the event of congestion, to make sure that the high-priority connections get serviced before the low-priority ones.

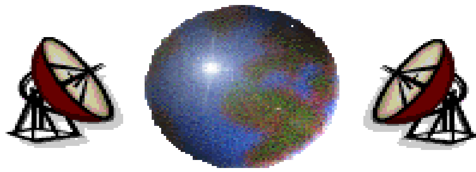
Resilience:

- It gives the probability of the transport layer itself spontaneously terminating a connection due to internal problems or congestion.



Transport Service Primitives

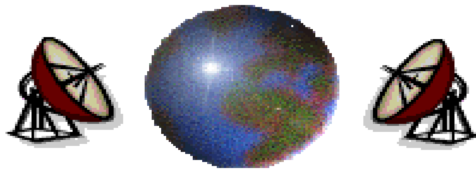
- The transport service primitives allow transport users (e.g., application programs) to access the transport service. Each transport service has its own access primitives.
- The purpose of the transport layer is to provide a reliable service on the top of an unreliable network. Therefore, it hides the imperfections of the network service so the user processes can just assume the existence of an error-free bit stream.
- The following transport service primitives allow application programs to establish, use, and release connection.



Transport Service Primitives

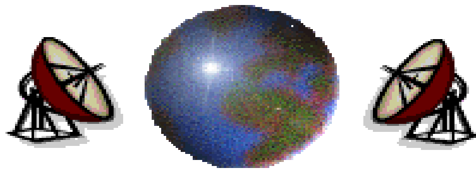
Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

The primitives for a simple transport service.

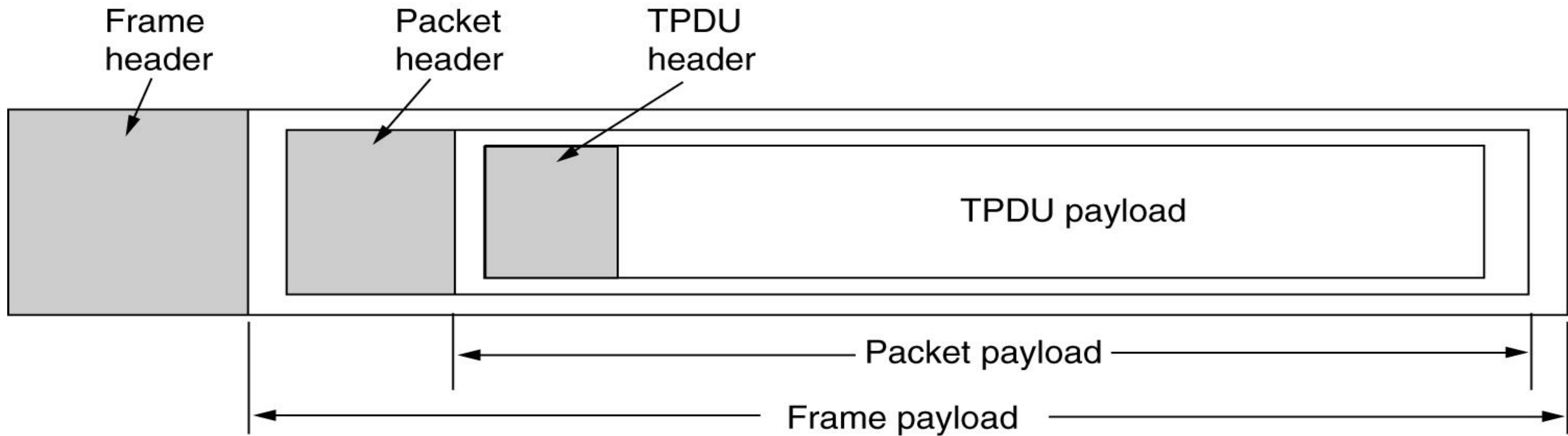


Transport Service Primitives

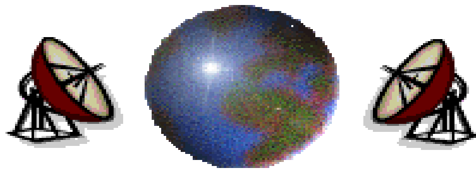
- A message, sent from transport entity to transport entity, is called **Transport Protocol Data Unit** (**TPDU**).
- TPDUs (exchanged by the transport layer) are contained in packets (exchanged by the neteork layer). In turns, packets are contained in frames (exchanged by the data link layer).
- When a frame arrives, the data link layer processes the frame header and passes the contents of the frame payload field up to the network entity. The network entity processes the packet header and passes the contents of the packet payload up to the transport entity.



Transport Service Primitives

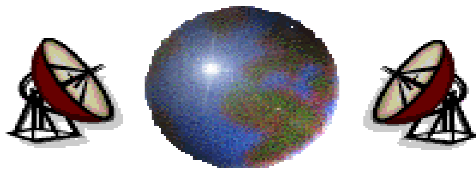


The nesting of TPDUs, packets, and frames

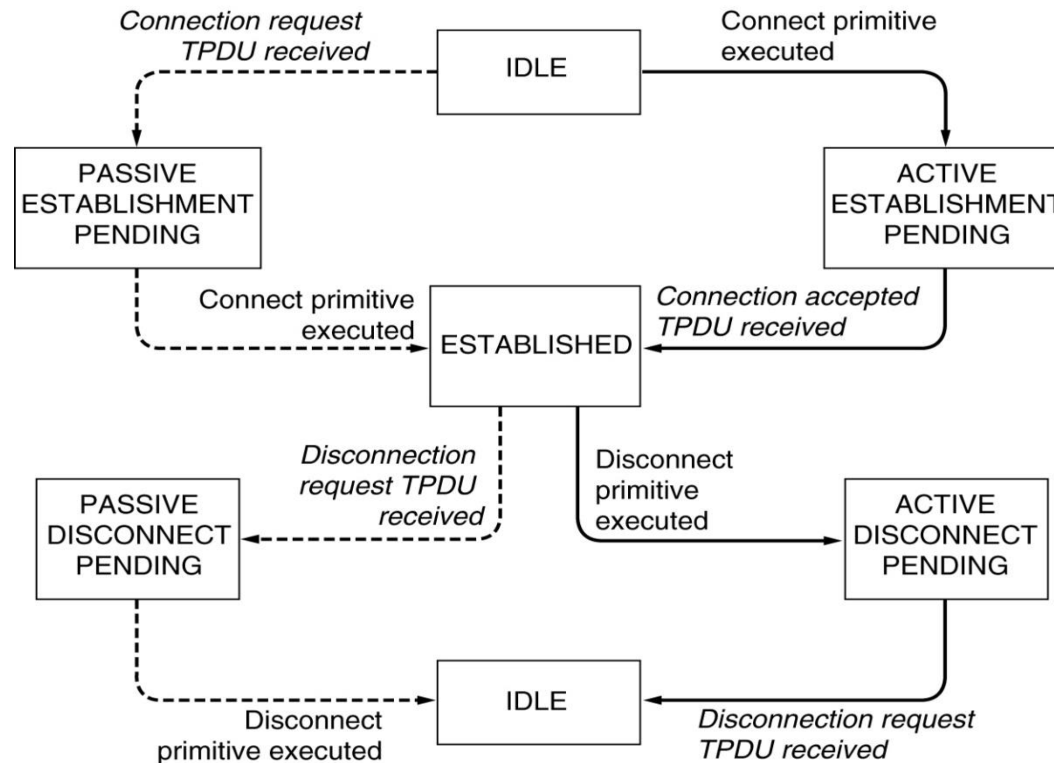


Transport Service Primitives

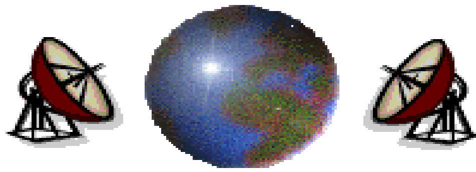
- Consider an application with a server and a number of remote clients.
- The server executes a **LISTEN** primitive to block the server (i.e., is interested in handling requests) until a client turns up.
- When a client wants to talk to a server, it executes a **CONNECT** primitive. This causes to block the client and to send a **CONNECTION REQUEST TPDU** to the server.
- When it arrives, the transport entity checks to see that server is blocked on a **LISTEN**. It then unblock the server and sends a **CONNECTION ACCEPTED TPDU** back to the client. When this TPDU arrives, the client is unblocked and the connection is established.
- When connection is no longer needed, it must be released by issuing a **DISCONNECT** primitive.



Transport Service Primitives

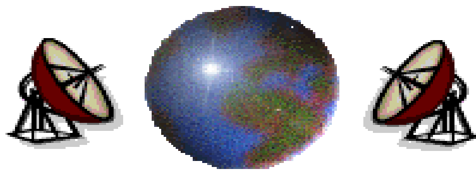


A state diagram for a simple connection management scheme. Transitions labeled in italics are caused by packet arrivals. The solid lines show the client's state sequence. The dashed lines show the server's state sequence.



Berkeley Sockets

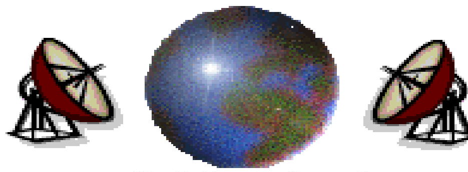
Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection



Socket Programming Example: Internet File Server

Client code using sockets.

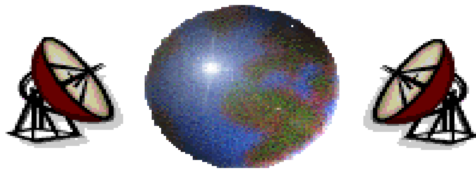
```
/* This page contains a client program that can request a file from the server program
 * on the next page. The server responds by sending the whole file.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define SERVER_PORT 12345                    /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                        /* block transfer size */

int main(int argc, char **argv)
{
  int c, s, bytes;
  char buf[BUF_SIZE];                        /* buffer for incoming file */
  struct hostent *h;                         /* info about server */
  struct sockaddr_in channel;                /* holds IP address */

  if (argc != 3) fatal("Usage: client server-name file-name");
  h = gethostbyname(argv[1]);                /* look up host's IP address */
  if (!h) fatal("gethostbyname failed");

  s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (s <0) fatal("socket");
  memset(&channel, 0, sizeof(channel));
  channel.sin_family= AF_INET;
  memcpy(&channel.sin  addr.s  addr, h->h  addr, h->h  length);
  channel.sin_port= htons(SERVER_PORT);

  c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
  if (c < 0) fatal("connect failed");

  /* Connection is now established. Send file name including 0 byte at end. */
  write(s, argv[2], strlen(argv[2])+1);

  /* Go get the file and write it to standard output. */
  while (1) {
       bytes = read(s, buf, BUF_SIZE);       /* read from socket */
       if (bytes <= 0) exit(0);              /* check for end of file */
       write(1, buf, bytes);                 /* write to standard output */
  }
}

fatal(char *string)
{
  printf("%s\n", string);
  exit(1);
}
```



Socket Programming Example: Internet File Server

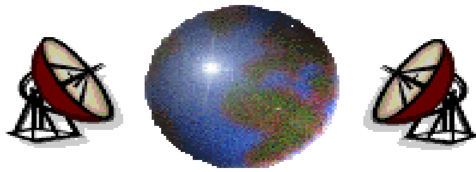
```
#include <sys/types.h>                    /* This is the server code */
#include <sys/fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define SERVER_PORT 12345                 /* arbitrary, but client & server must agree */
#define BUF_SIZE 4096                     /* block transfer size */
#define QUEUE_SIZE 10
int main(int argc, char *argv[])
{
 int s, b, l, fd, sa, bytes, on = 1;
 char buf[BUF_SIZE];                      /* buffer for outgoing file */
 struct sockaddr_in channel;              /* hold's IP address */
 /* Build address structure to bind to socket. */
 memset(&channel, 0, sizeof(channel));    /* zero channel */
 channel.sin_family = AF_INET;
 channel.sin_addr.s_addr = htonl(INADDR_ANY);
 channel.sin_port = htons(SERVER_PORT);
 /* Passive open. Wait for connection. */
 s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);   /* create socket */
 if (s < 0) fatal("socket failed");
 setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));
 b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
 if (b < 0) fatal("bind failed");
 l = listen(s, QUEUE_SIZE);               /* specify queue size */
 if (l < 0) fatal("listen failed");
 /* Socket is now set up and bound. Wait for connection and process it. */
 while (1) {
     sa = accept(s, 0, 0);                /* block for connection request */
     if (sa < 0) fatal("accept failed");
     read(sa, buf, BUF_SIZE);             /* read file name from socket */
     /* Get and return the file. */
     fd = open(buf, O_RDONLY);            /* open the file to be sent back */
     if (fd < 0) fatal("open failed");
     while (1) {
          bytes = read(fd, buf, BUF_SIZE); /* read from file */
          if (bytes <= 0) break;           /* check for end of file */
          write(sa, buf, bytes);           /* write bytes to socket */
     }
     close(fd);                           /* close file */
     close(sa);                           /* close connection */
 }
}
```

Client code using sockets.



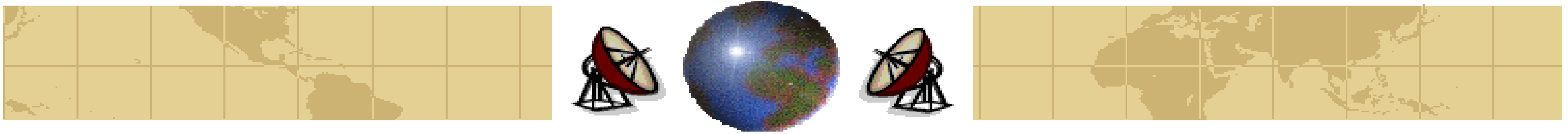
Elements of Transport Protocols

- Addressing
- Connection Establishment.
- Connection Release
- Flow Control and Buffering
- Multiplexing
- Crash Recovery

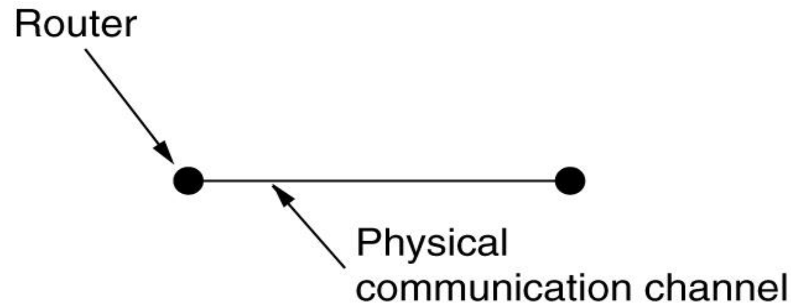


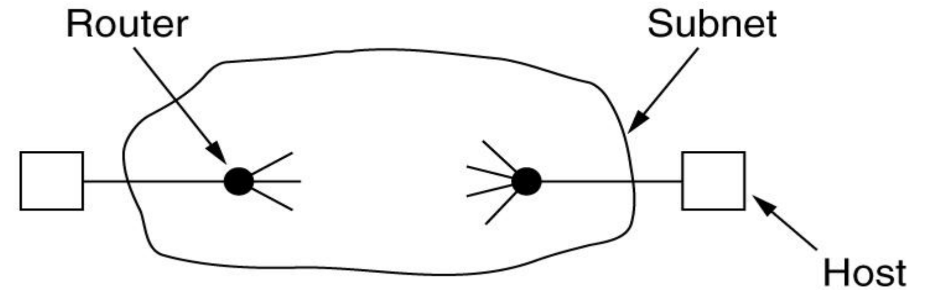
Elements of Transport Protocols

- The transport service is implemented by a **transport protocol** used between the two transport entities.
- Both transport protocols and data link protocols have to deal with error control, sequencing, and flow control.
- However, significant differences between the two also exist. At data link layer, two routers communicate directly, via a physical channel, whereas at the transport layer, this physical channel is replaced by the entire subnet.

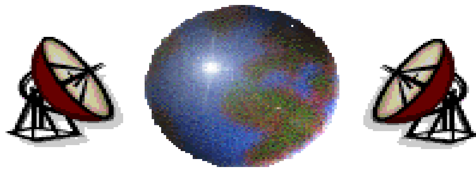


Transport Protocol





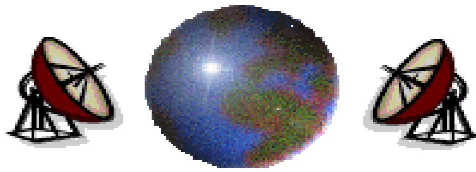
(a) Environment of the data link layer.
(b) Environment of the transport layer.



Elements of Transport Protocols

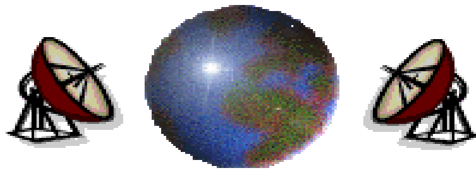
Differences between the Data Link Layer and Transport Layer:

- **First:** In data link layer, each outgoing line uniquely specifies a particular router. However, in the transport layer, explicit addressing of destinations is required.
- **Second:** The process of establishing a connection over a wire is simple. However, in transport layer, initial connection establishment is more complicated.
- **Third:** The existence of storage capacity in the subnet. In data link layer, the frame may arrive or be lost, but it cannot bounce around for a while. However, in transport layer, there is a nonnegligible probability that a packet may be stored for a number of seconds and then delivered later.
- **Fourth:** Buffering and flow control are needed in both layers, but in the transport layer may require a different approach than we used in the data link layer.

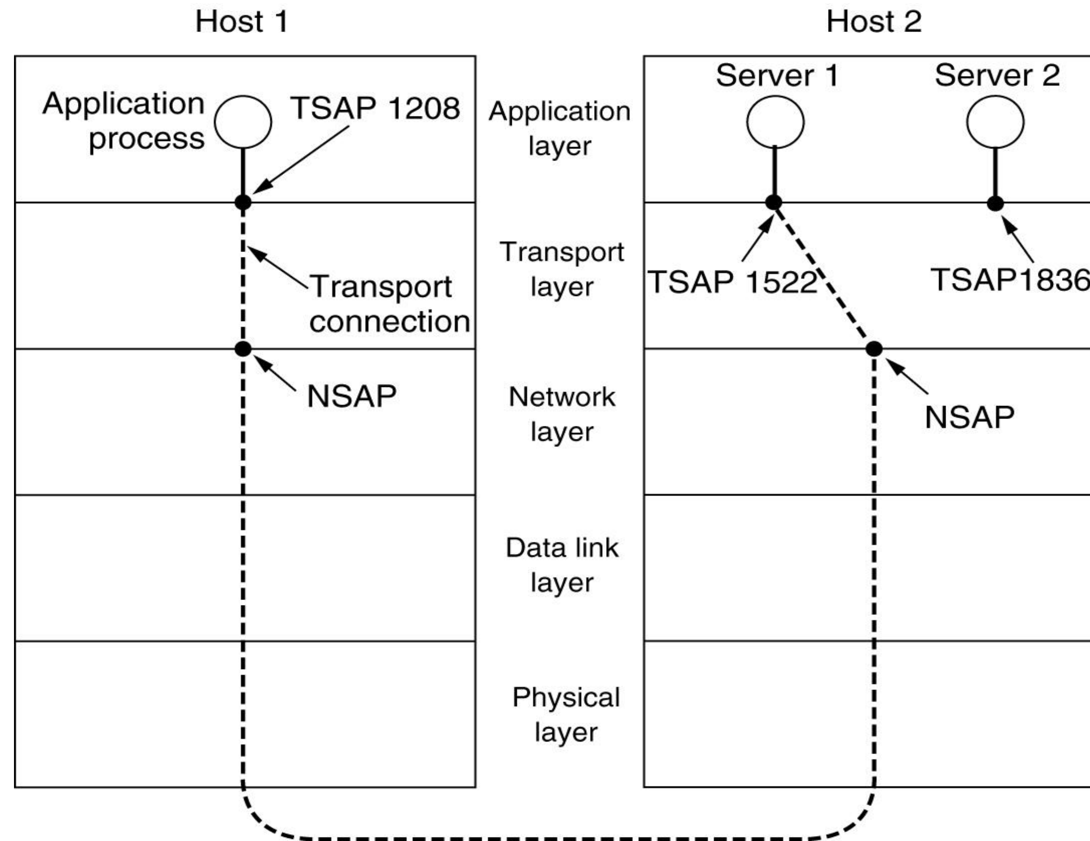


Addressing

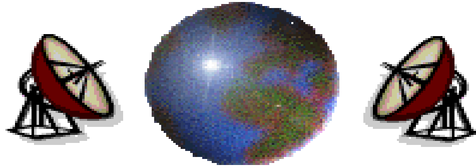
- When an application process wishes to set up a connection to a remote application process, it must specify which one to connect to.
- The method is to define transport addresses to which processes can listen for connection request. These are called **Transport Service Access Point (TSAP)**. In the Internet, these points are (IP address, local port) pairs. Similarly, the end points in the network layer are called **Network Service Access Point (NSAP).**



Addressing



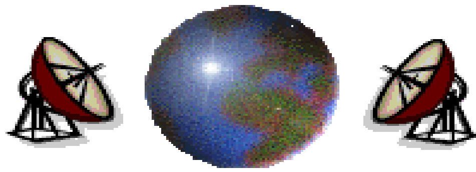
TSAPs, NSAPs and transport connections



Addressing

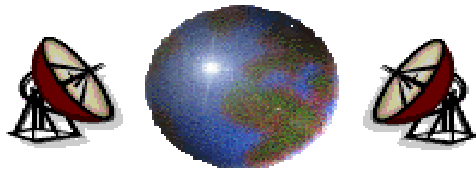
Example:

- A possible connection scenario for a transport connection over a connection oriented a network layer is as follows.
  1. A time-of-day server on host 2 attaches itself to TSAP 122 to wait for an incoming call. A call such as **LISTEN** might me used.
  2. An application process on host 1 wants to find out the time-of-day, so it issues a **CONNECT** request specifying TSAP 6 as the source and TSAP 122 as the destination.
  3. The transport entity on host 1 selects a network address on its machine (if it has more than one) and sets up a network connection to make host 1's transport entity to talk to the transport entity on host 2.
  4. The first thing the transport entity on 1 says to its peer on 2 is: "Good morning. I would like to establish a transport connection between my TSAP 6 and your TSAP 122. What do you say?"
  5. The transport entity on 2 then asks the time-of-day server at TSAP 122 if it is willing to accept a new connection.



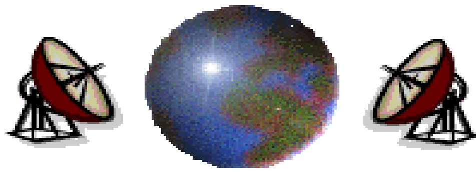
Addressing

- How does the user process on host 1 know that the time-of-day server is attached to TSAP 122?
- One possibility is stable TSAP addresses, i.e., the time-of-day server has been attaching itself to TSAP 122 for years, and gradually all the network users have learned this.
- Stable TSAP addresses might work for a small number of key services. However, in general, user processes often talk to other user processes for a short time and do not have a TSAP address that is known in advance. Furthermore, if there are many server processes, most of which are rarely used, it is wasteful to have each of them active and listening to a stable TSAP address all day long. Therefore, a better scheme is needed.



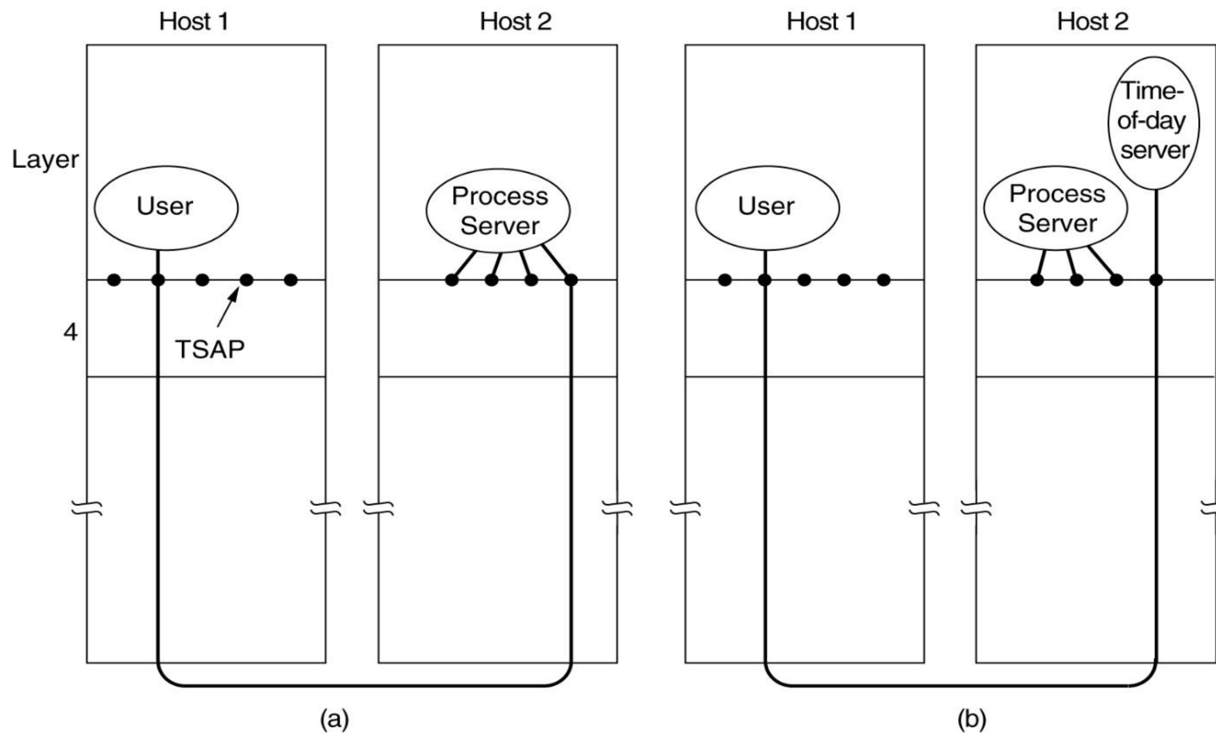
Addressing

- One scheme, used by UNIX on the Internet, is known as the **initial connection protocol**, which is as follows.
- Each machine that wishes to offer service to remote users has a special **process server**, which listens to a set of ports at the same time, waiting for a TCP connection request, specifying a TCP address (TCP port) of the service they want. If no server is waiting for them, they get a connection to the process server.
- After it gets the incoming request, the process server produces the requested server, which then does the requested work, while the process server goes back to listening for new requests.

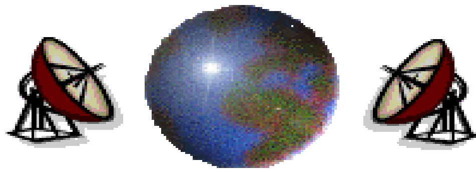


Addressing

Example for Initial Connection Protocol:



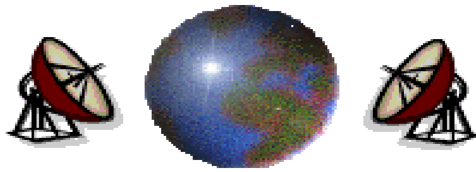
How a user process in host 1 establishes a connection with a time-of-day server in host 2.



Addressing

- For a **file server**, it needs to run a special hardware (a machine with a disk).
- There exists a special process called a **name server** or sometimes a **directory server**.
- To find the TSAP address corresponding to a given service name, such as "time-of-day", a user sets up a connection to the **name server**. The user then sends a message specifying the **service name**. The name server sends back the TSAP address. Then the user releases the connection with the name server and establishes a new one with the desired service.
- TSAP addresses can be either **hierarchical addresses** or **flat addresses**.
- **Hierarchical Address** =

  <galaxy><star><planet><country><network><host><port>.

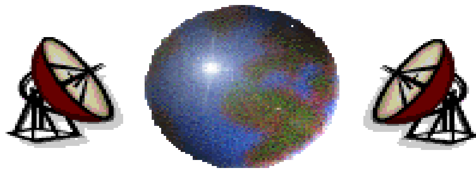


Establishing a Connection

- Establishing a connection sounds easy. It would seem sufficient for one entity to just send a **CONNECTION REQUEST** TPDU to the destination and wait for a **CONNECTION ACCEPTED** reply. The problem occurs when the network can lose, store, and duplicate packets.

- To solve this problem, one way is to use throw away transport addresses. Each time a transport address is needed, a new one is generated. When a connection is released, the address is discarded.

- Another possibility is to give each connection identifier (i.e., a sequence number incremented for each connection established), chosen by the initiating party, and put in each TPDU, including the one requesting the connection. After each connection is released, each transport entity could update a table listing obsolete connections as (peer transport entity, connection identifier) pairs. Whenever a connection request came in, it could be checked against the table to see if it belong to previously released connection.

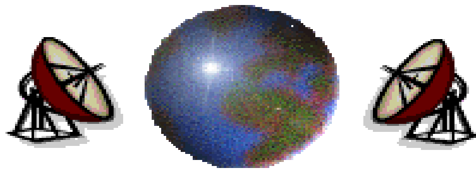


Establishing a Connection

- The drawback of this scheme is that when a machine crashes and loses its history, it will no longer know which connection identifiers have already been used.

- Therefore, we have to devise a mechanism to kill of the aged packets using one of the following techniques:
  1. **Restricted subnet design:** It prevents packet from looping.
  2. **Putting a hop limit in each packet.**
  3. **Timestamping each packet:** The router clocks need to be synchronized.

- In practice, we will need to guarantee not only that packet is dead, but also that all acknowledgements to it are also dead. If we wait a time *T* after a packet has been sent, we can sure that all traces of it are now gone and that neither it nor its acknowledgements will suddenly appear.

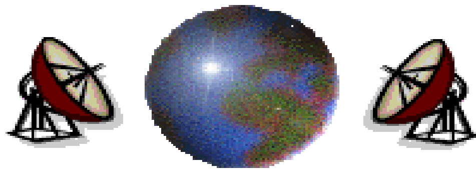


Establishing a Connection

- To get around a machine losing all memory, **Tomlinson** proposed to equip each host with a time-of-day clock. The clocks at different hosts need not to be synchronized. Furthermore, the number of bits in the counter must equal or exceed the number of bits in the sequence number. Also, the clock is assumed to continue running even if the host goes down.

- The basic idea is to ensure that two identically numbered TDPU are never outstanding at the same time. When a connection is set up, the low-order k bits of the clock are used as the initial sequence number. Therefore, each connection starts numbering its TDPUs with a different sequence number.

- The sequence space should be so large that by the time sequence number wrap around, old TPDUs with the same sequence number are long gone.



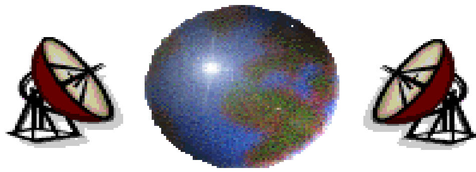
Establishing a Connection

- To establish a connection, there is a potential problem in getting both sides to agree on the initial sequence number.

- For example, host 1 establishes a connection by sending a **CONNECT REQUEST TPDU** containing the proposed initial sequence number and destination port number to a remote peer, host 2. The receiver, host 2, then acknowledges this request by sending a **CONNECTION ACCEPT TPDU** back.

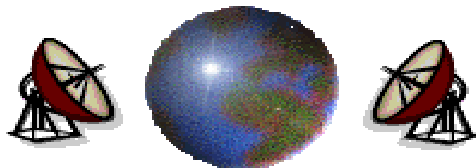
- If the **CONNECTION REQUEST TPDU** is lost but a delayed duplicate **CONNECTION REQUEST** suddenly shows up at host 2, the connection will be established incorrectly.

- To solve this problem, **Tomlison** introduced the **three-way handshake**.

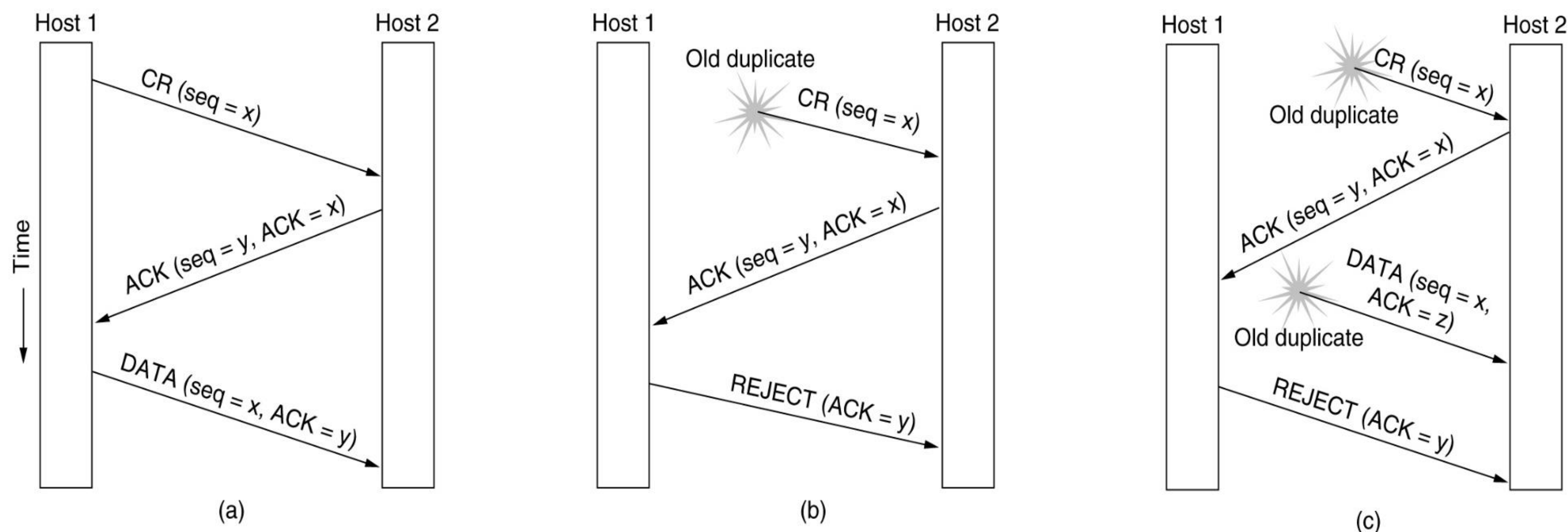


Establishing a Connection

- The **three-way handshake** protocol does not require both sides to begin sending with the same sequence number.
- Host 1 chooses a sequence number, *x*, and sends **a CONNECTION REQUEST TPDU** containing *x* to host 2. Host 2 replies with a **CONNECTION ACCEPTED TDPU** acknowledging *x* and announcing its own initial sequence number, *y*. Finally, host 1 acknowledges host 2 in the first data TPDU that it sends.
- The three-way handshake works in the presence of delayed duplicate control TPDUs.



Connection Establishment

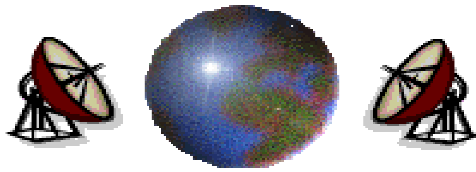


Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST.

(a) Normal operation.

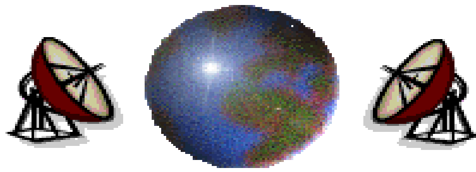
(b) Old CONNECTION REQUEST appearing out of nowhere.

(c) Duplicate CONNECTION REQUEST and duplicate ACK.

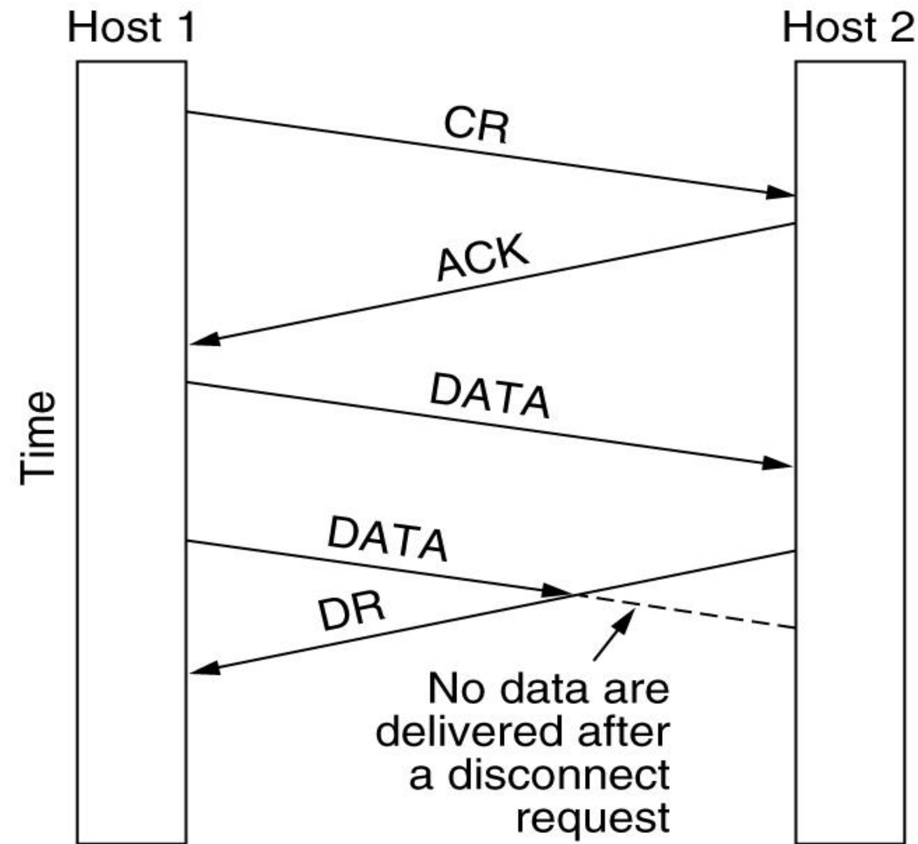


Releasing a Connection

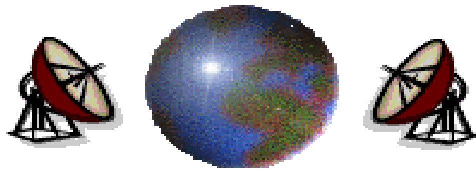
- There are two styles of terminating a connection: **asymmetric release** and **symmetric release**.
- **Asymmetric release** is the way the telephone works: when one party hangs up, the connection is broken.
- Asymmetric release may result in a data loss. If a connection is established, host 1 sends a TDPU to host 2. Then host 1 sends another TPDU. However, host 2 issues a **DISCONNECT** before the second TPDU arrives. The result is that the connection is released and the data are lost.



Connection Release



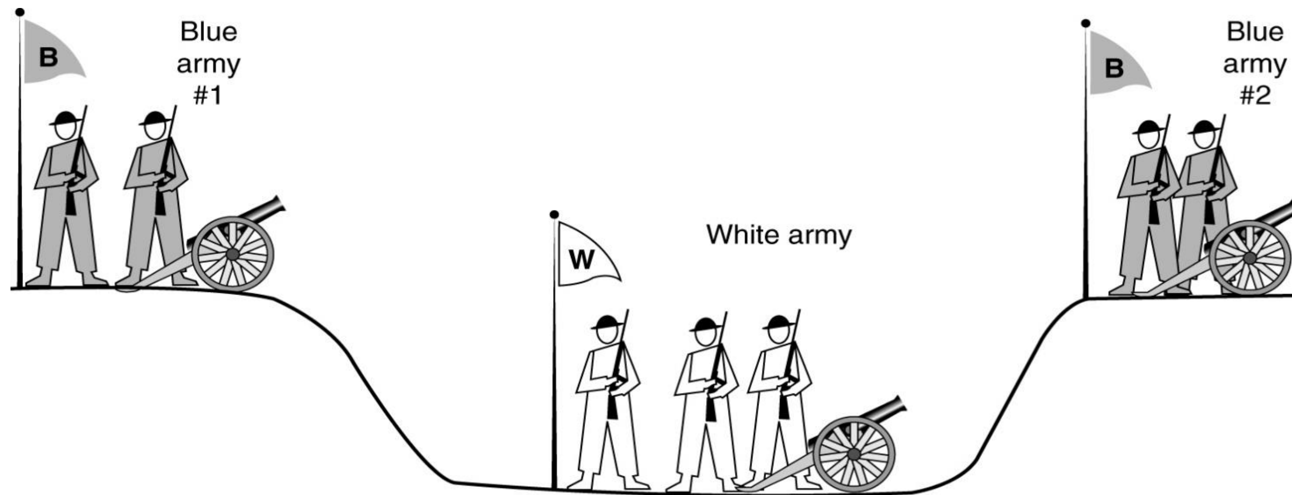
Abrupt disconnection with loss of data.



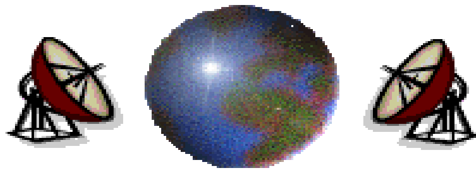
Releasing a Connection

- **Symmetric release** treats the connection as two separate unidirectional connections and requires each one to be released separately.

- Although symmetric release is more sophisticated protocol that avoids data loss, it does not always work. There is a famous problem that deals with this issue. It is called the **two-army problem**.



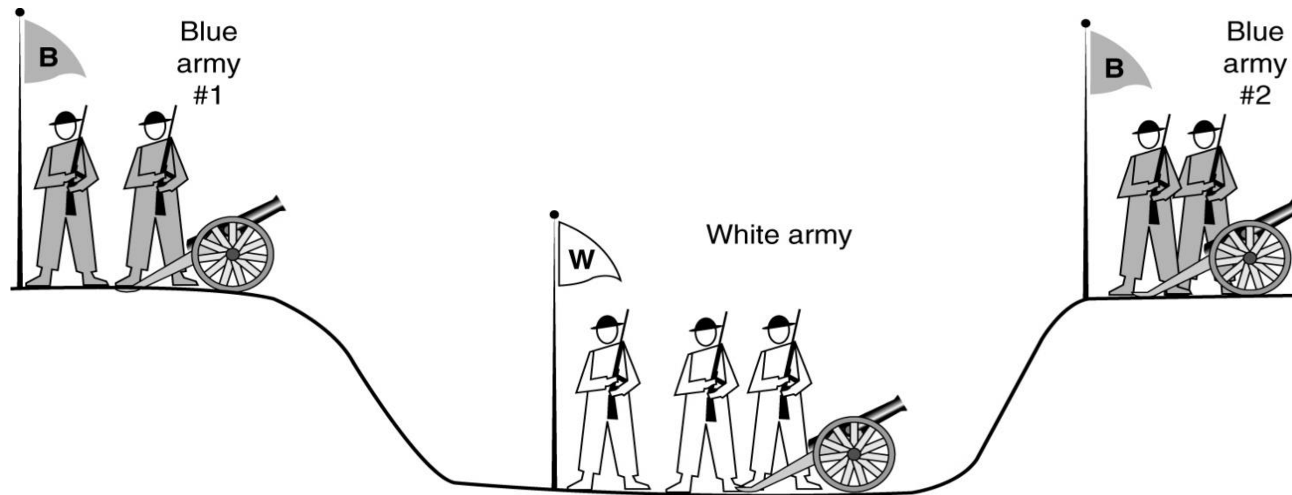
The two-army problem.



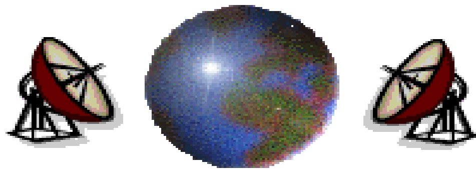
Releasing a Connection

- **Symmetric release** treats the connection as two separate unidirectional connections and requires each one to be released separately.

- Although symmetric release is more sophisticated protocol that avoids data loss, it does not always work. There is a famous problem that deals with this issue. It is called the **two-army problem**.



The two-army problem.



Connection Release

Host 1 | Host 2 | Host 1 | Host 2

Send DR + start timer
DR
Send DR + start timer
DR
Release connection
Send ACK
ACK
Release connection

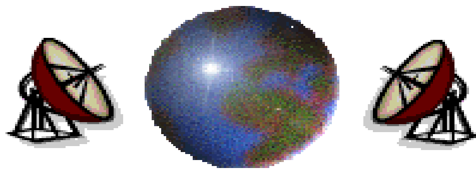
(a)

Send DR + start timer
DR
Send DR + start timer
DR
Release connection
Send ACK
ACK
Lost
(Timeout) release connection

(b)

Four protocol scenarios for releasing a connection.

(a) Normal case of a three-way handshake. (b) final ACK lost.



Connection Release

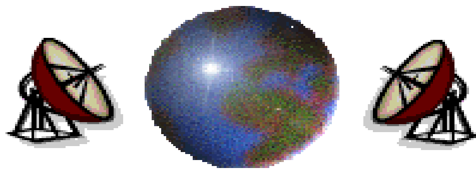
Host 1
Host 2
Send DR + start timer
DR
Send DR & start timer
DR
Lost
(Timeout) send DR + start timer
DR
Send DR & start timer
DR
Release connection
Send ACK
ACK
Release connection

(c)

Host 1
Host 2
Send DR + start timer
DR
Send DR & start timer
Lost
(Timeout) send DR + start timer
Lost
(N Timeouts) release connection
(Timeout) release connection

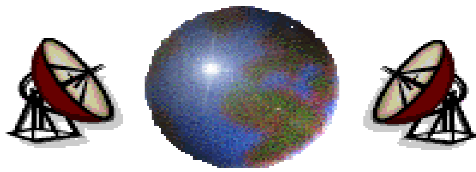
(d)

(c) Response lost. (d) Response lost and subsequent DRs lost.



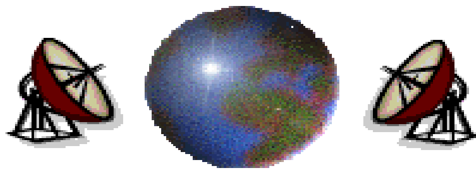
Flow Control and Buffering

- Flow Control scheme is needed on each connection to keep a fast transmitter from overrunning a slow receiver.
- If the subnet provides a datagram service, the sending transport entity must buffer outgoing frames because they might be retransmitted.
- If the receiver knows that the sender buffers all TPDUs until they are acknowledged, the receiver may or may not dedicate specific buffers to specific connections. The receiver may maintain a single buffer pool shared by all connections.
- When a TPDU comes in, an attempt is made to acquire a new buffer. If one is available, the TPDU is accepted, otherwise, it is discarded. Since the sender is prepared to retransmit TPDUs lost by the subnet, no harm is done by having the receiver drop TPDUs. The sender just keeps trying until it gets an acknowledgement.



Flow Control and Buffering

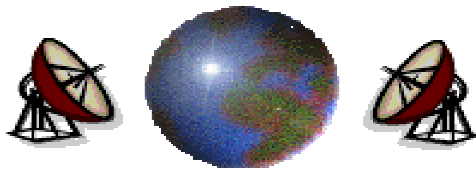
- In summary, If the network service is **unreliable**, the sender must buffer all TPDUs sent. However, with **reliable** network service, if the sender knows that the receiver always has a buffer space, it need not retain copies of the TPDUs it sends. However, if the receiver cannot guarantee that every incoming TPDU will be accepted, the sender will have to buffer anyway.



Flow Control and Buffering

Buffer Size

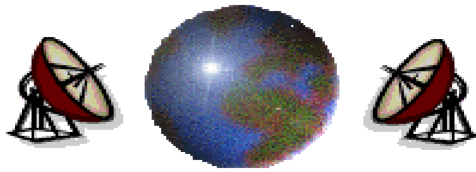
- If most TPDUs are nearly the same size, we can organize the buffers as **a pool of identical size buffers**, with one TPDU per buffer.
- If there is a wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, a pool of fixed-sized buffers 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.
- Another approach to the buffer size problem is to use **variable-size buffers**. The advantage 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**. This is good approach when all connections are heavily loaded, but is poor if some connections are lightly loaded.



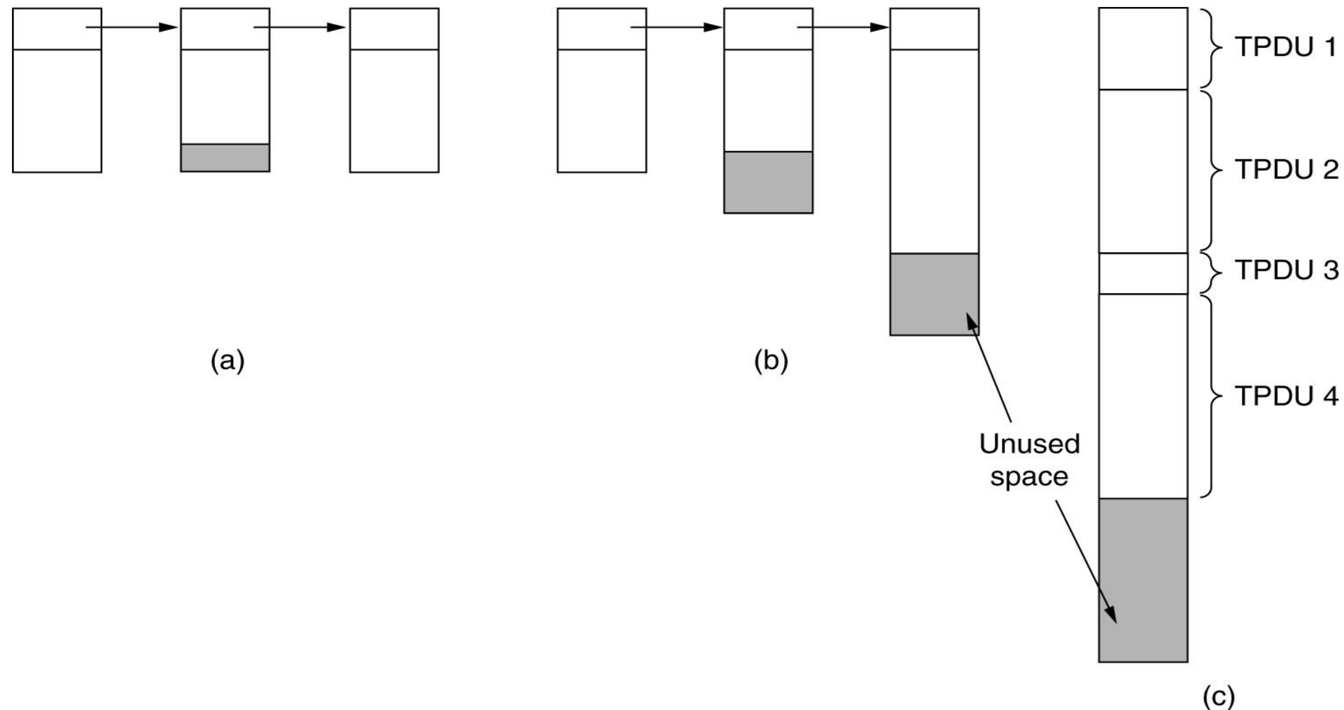
Flow Control and Buffering

Buffer Size

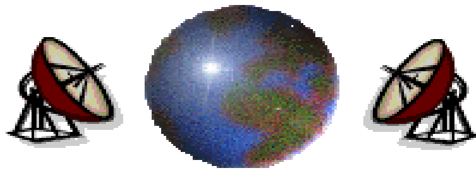
- If most TPDUs are nearly the same size, we can organize the buffers as **a pool of identical size buffers**, with one TPDU per buffer.
- If there is a wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, a pool of fixed-sized buffers 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.
- Another approach to the buffer size problem is to use **variable-size buffers**. The advantage 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**. This is good approach when all connections are heavily loaded, but is poor if some connections are lightly loaded.



Flow Control and Buffering



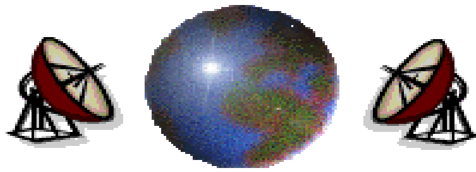
(a) Chained fixed-size buffers. (b) Chained variable-sized buffers.
(c) One large circular buffer per connection.



Flow Control and Buffering

Dynamic Buffer Allocation:

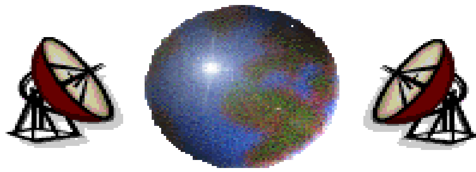
- The sender requests a certain number of buffers. The receiver then grants as many of these as it can afford. Every time the sender transmits a TPDU, it must decrements its allocation, and stops when the allocation reaches zero. The receiver then separately piggybacks both acknowledgements and buffer allocations onto the reverse traffic.



Flow Control and Buffering

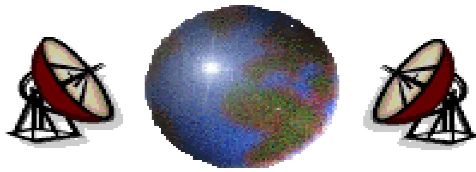
	A	Message	B	Comments
1	→	< request 8 buffers>	→	A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	A may now send 5
12	←	<ack = 4, buf = 2>	←	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	A is now blocked again
15	←	<ack = 6, buf = 0>	←	A is still blocked
16	• • •	<ack = 6, buf = 4>	←	Potential deadlock

Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (…) indicates a lost TPDU.

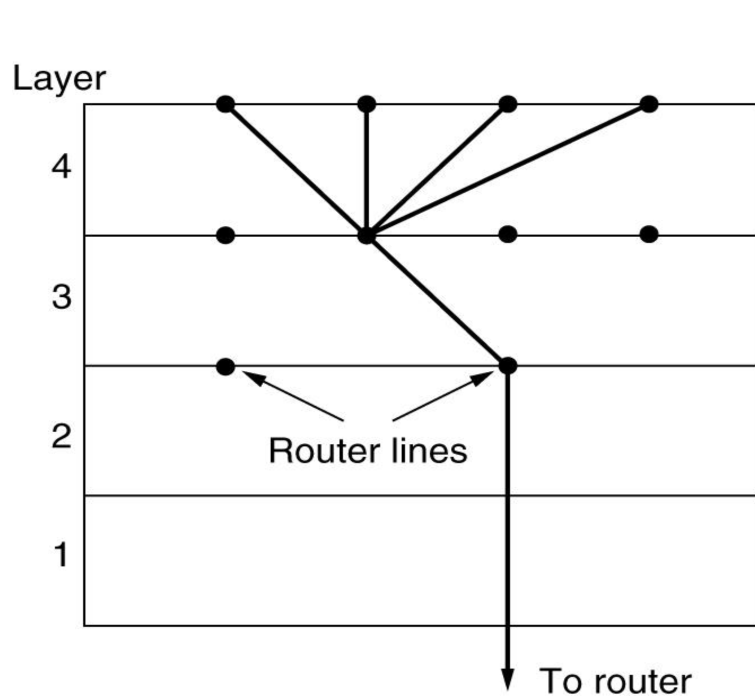


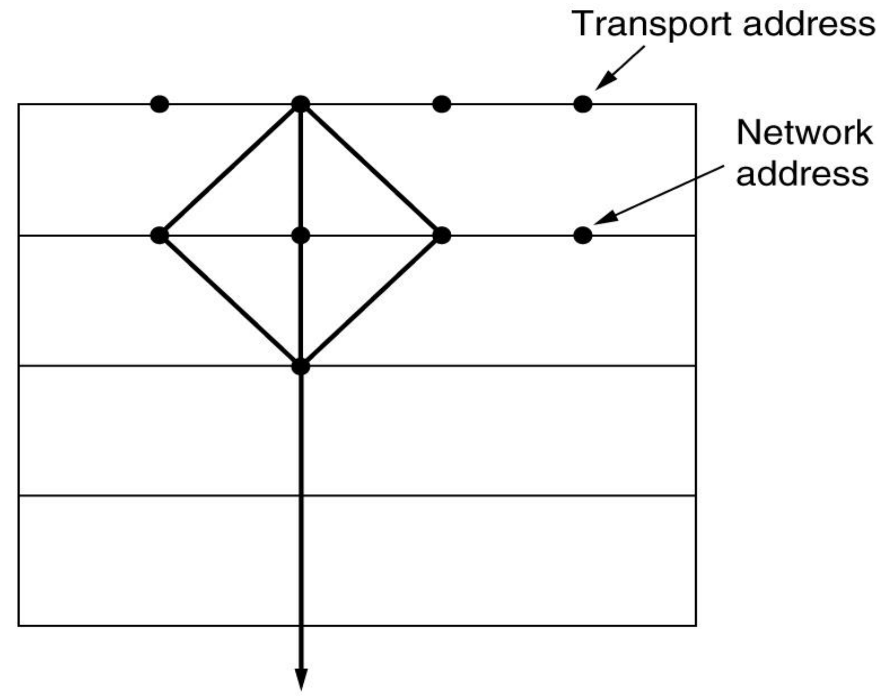
Multiplexing

- In networks that use virtual circuits within the subnet, each open connection consumes some table space in the routers for the entire duration of the connection. If buffers are dedicated to the virtual circuit in each router, a user who left a terminal logged into a remote machine for a period is consuming expensive resources.
- The consequence of billing a user based on the amount of data sent, not he connection time, is to have many virtual circuits open for long periods of time. This makes multiplexing of different transport connections onto the same network connection attractive.
- It is up to the transport layer to group transport connections according to their destination and map each group onto the minimum number of network connections.

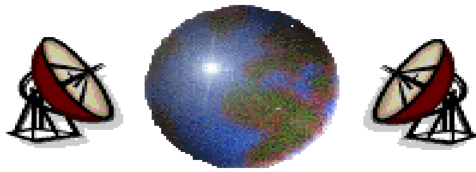


Multiplexing



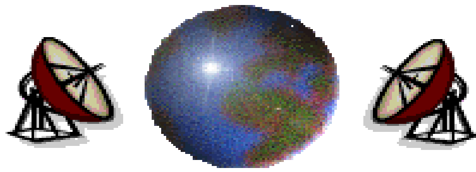


(a) Upward multiplexing. (b) Downward multiplexing.



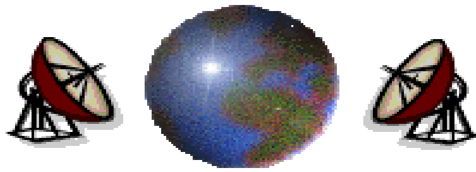
The Internet Transport Protocols: TCP

- Introduction to TCP
- The TCP Service Model
- The TCP Protocol
- The TCP Segment Header
- TCP Connection Establishment
- TCP Connection Release
- TCP Connection Management Modeling
- TCP Transmission Policy
- TCP Congestion Control
- TCP Timer Management
- Wireless TCP and UDP
- Transactional TCP



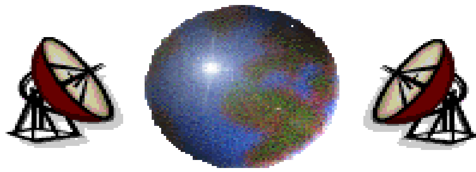
The Internet Transport Protocols (TCP and UDP)

- The Internet has two main protocols in the transport layer, a **connection-oriented protocol (TCP) and a connectionless protocol (UDP)**.
- **TCP (Transmission Control Protocol)** was designed to provide a reliable end-to end connection over an unreliable internetwork.
- TCP was designed to dynamically adapt to the properties of the internetwork.
- A TCP entity accepts user data streams from local processes, breaks them up into pieces not exceeding 64K bytes (in practice, usually 1500 bytes), and sends each piece as a separate IP datagram. When IP datagrams containing TCP data arrive at a machine, they are given to the TCP entity, which reconstruct the original byte streams.
- The IP layer gives no guarantee that datagram will be delivered properly, so it is up to TCP to time out and retransmit them. Also, it is up to TCP to reassemble the datagrams into messages in the proper sequence. Therefore, TCP provides the reliability the most users want and that IP does not provide.



The TCP Service Model

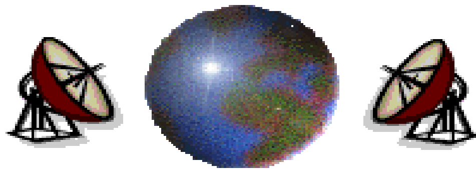
- TCP service is obtained by having both the sender and receiver create end points, called sockets. Each socket has a socket number (address) consisting of the IP address of the host and a 16-bit port. A port is the TCP name for a TSAP.
- To obtain TCP service, a connection must be explicitly established between a **socket on the sending machine** and a **socket on the receiving machine** using the socket calls.
- Port numbers below 1024 are called well-known ports and are reserved for standard services. For example, port 21 is for **FTP**, port 23 for **TELNET**, port 79 for **finger**, and port 119 for **USENET** news.
- All TCP connections are full-duplex and point-to-point.



The TCP Service Model

Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial File Transfer Protocol
79	Finger	Lookup info about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

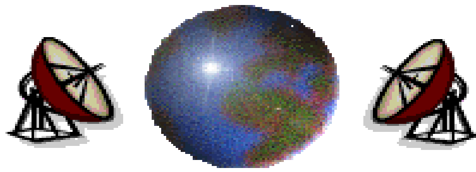
Some assigned ports.



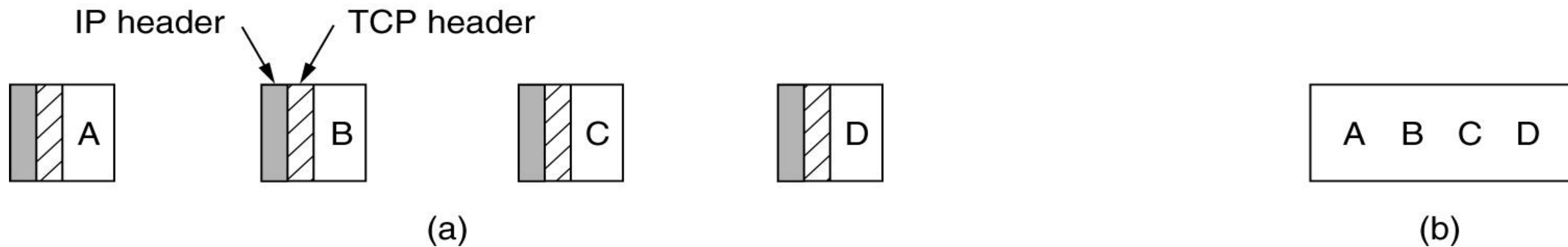
The TCP Service Model

- 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 write to a TCP stream, these data may delivered to the receiving processes as four 512-byte pieces, two 1024-byte pieces, one 2048-byte piece, or some other way.

- When an application passes data to TCP, TCP may send it immediately or buffer it (in order to collect a larger amount to send at once). However, sometimes, the application really wants the data to be sent immediately. For example, suppose a user is logged into a remote machine. After a command line has been finished and the carriage return typed, it is essential that the line be shipped off to the remote machine immediately and not buffered until the next line comes in.

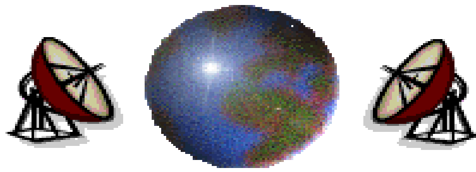


The TCP Service Model



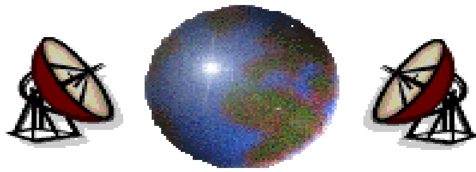
(a) Four 512-byte segments sent as separate IP datagrams.

(b) The 2048 bytes of data delivered to the application in a single READ CALL.



The TCP Protocol

- A sending and receiving TCP entities exchange data in form of **segments**.
- A **segment** consists of a 20-byte header (plus an optional part) followed by zero or more data bytes.
- Two limits restrict the segment size. **First**, each segment, including the TCP header, must fit in the 65,535-byte IP payload. **Second,** each network has a **maximum transfer unit** (**MTU**). If a segment passes through sequence of networks without being fragmented and then hits one whose MTU is smaller than the segment, the router at the boundary fragments the segment into two or more smaller segments. Each new segment gets its own TCP and IP headers, so fragmentation by routers increases the total overhead.

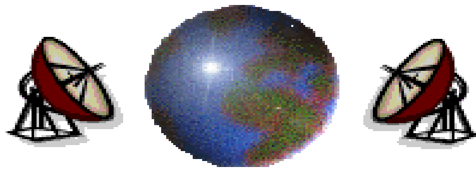


The TCP Protocol

- The basic protocol used by TCP entities is the **sliding window protocol**. When a sender transmits a segment, it also starts a timer. When a segment arrives at the destination, the receiving TCP entity sends back a segment (with data if any exists, otherwise without data) bearing an **acknowledgement** number equal to the next sequence number it expects to receive. If a sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

Challenges:

- Segments can be fragmented, it is possible that part of the transmitted segment arrives and acknowledged by the receiving TSP entity, but the rest is lost.
- Segments can arrive out of order.
- Segments can also be delayed and the sender times out and retransmits them.
- If a retransmitted segment takes a different route than the original, and is fragmented differently.
- It is possible that a segment may occasionally hit a congested (or broken) network along its path.



The TCP Segment Header

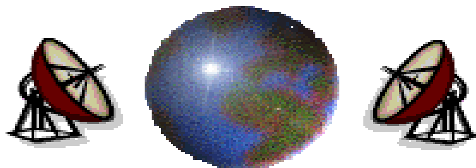
TCP Header	Data

TCP Segment

Bit positions: 1, 8, 16, 24, 32

(Source Port) بوابة المرسل							(Destination Port) بوابة المستقبل	
(Sequence Number) الرقم التسلسلي								
(Acknowledgement Number) رقم الإقرار								
طول الرأس		URG	ACK	PSH	RST	SYN	FIN	(Window Size) حجم النافذة
(Checksum) اختبار الصحة							(Urgent Pointer) مؤشر الأهمية	
(Options) خيارات								

TCP Header.

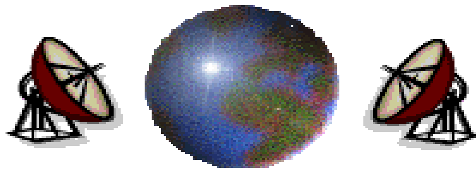


The TCP Segment Header

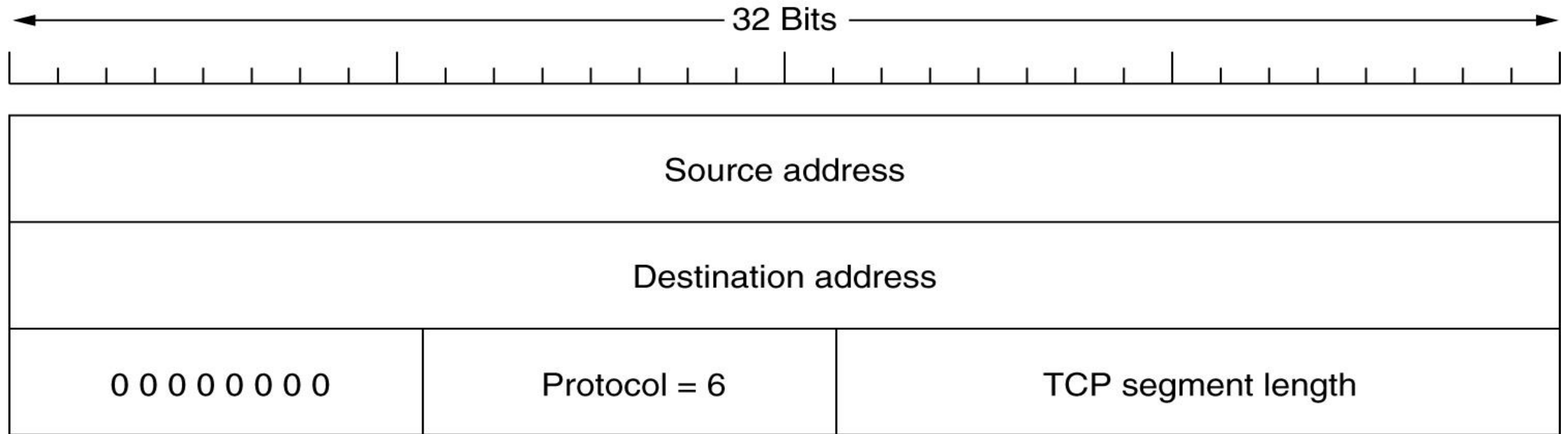
32 Bits

Source port								Destination port
Sequence number								
Acknowledgement number								
TCP header length		URG	ACK	PSH	RST	SYN	FIN	Window size
Checksum								Urgent pointer
Options (0 or more 32-bit words)								
Data (optional)								

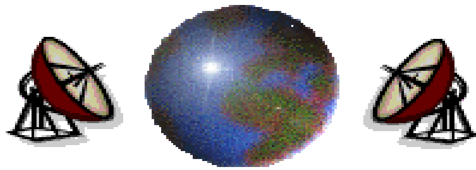
TCP Header.



The TCP Segment Header



The pseudoheader included in the TCP checksum.



The TCP Segment Header

Source Port and Destination Port Fields (16 bits each):

- The *source port* and *destination port* identify the local end points of the connection. A port plus its host's IP address forms a 48-bit unique TSAP.

Sequence Number and Acknowledgement Number Fields (32 bits each):

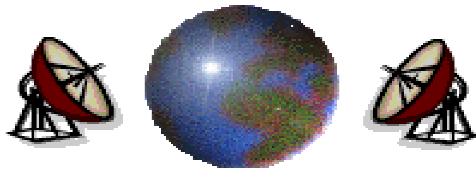
- The *sequence number* and *acknowledgement number* fields perform their usual function. The acknowledgement number field specifies the **next byte expected**. Both are 32 bits long because every byte of data is numbered in a TCP stream.

TCP Header Length Field (4 bits):

- The *TCP header length* field tells the number of 32-bit words in the TCP header.

Reserved Field (6-bit):

- The *reserved* field is reserved for future use.



The TCP Segment Header

URG Bit:

- The *URG* bit is set to 1 to indicate that the *urgent pointer* is valid. The *urgent pointer* indicates amount of urgent (expedited) data in the segment.

ACK Bit:

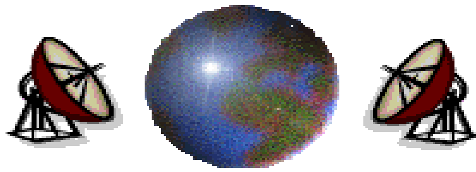
- The *ACK* bit set to 1 to indicate that the *acknowledgement number* is valid. If *ACK* is 0, the segment does not contain an acknowledgement so the *acknowledgement number* field is ignored.

PSH Bit:

- If the *PSH* bit is set, the receiver is requested to deliver the data to the application upon arrival and not buffered until a full buffer has been received.

RST Bit:

- The *RST* bit is used to reset a connection that has become confused due to host crash or some other reason. It is also used to reject an invalid segment or to refuse an attempt to open a connection.



The TCP Segment Header

SYN Bit:

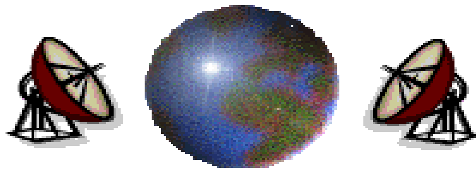
- The *SYN* bit is used to establish connections. The connection request has *SYN*=1 and *ACK*=0 to indicate that the piggyback acknowledgement field is not in use. The connection reply does bear an acknowledgement, so it has *SYN*=1 and *ACK*=1. Therefore, the *SYN* bit is used to denote **CONNECTION REQUESTED** and **CONNECTION ACCEPTED**, with *ACK* bit used to distinguish between those two possibilities.

FIN Bit:

- The *FIN* bit is used to release connection. It specifies that the sender has no more data to transmit.

Window Size Field (16 bits):

- The *window size* field relates to the **sliding window flow control** scheme. It indicates the number of bytes may be sent starting at the byte acknowledged.

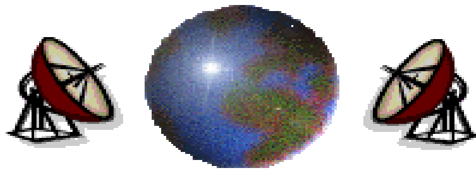


The TCP Segment Header

Checksum Field (16 bit):

- A *checksum* is provided for extreme reliability. It checksums the header, the data, and the conceptual pseudoheader. The checksum is the 1's complement of the sum of all the 16-bit words in the segment added together using 1's complement arithmetic. As consequence, when the receiver performs the calculation on the entire segment including the *checksum* field, the result should be zero.

- The pseudoheader contains the 32-bit IP addresses if the source and destination machines, the protocol number for TCP (6), and the byte count for the TCP segment (including the header).



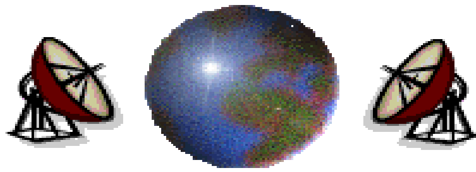
The TCP Segment Header

Urgent Pointer Field (16 bits):

- The urgent pointer indicates the amount of urgent (expedited) data in the segment. Normally this is delivered by the receiving TCP entity immediately it is received.

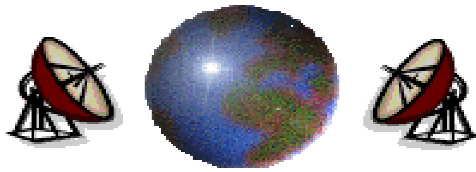
Option Field:

- The *option* field was designed to provide a way to add extra facilities not covered by the regular header. The most important option is the one that allows each host to specify the maximum TCP payload it is willing to accept. During connection setup, each side can announce its maximum, and the smaller of the two numbers wins. If a host does not use this option, it defaults to a 536-byte payload.



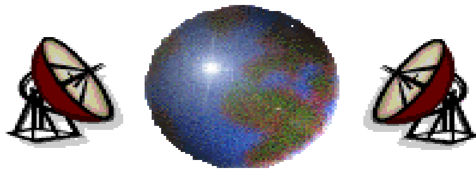
The TCP Segment Header

- For lines with high bandwidth, high delay, or both, the 64 KByte window is often a problem.
- On a T3 line (44.736 Mbps), it take only 12 msec to output a full 64 KByte window. If the round trip propagation delay is 50 msec (typical for **transcontinental fiber**), the sender will be idle 3/4 of the time waiting for acknowledgement.
- On a satellite connection, the situation is even worse. A larger window size would allow the sender to keep pumping data out, but using 16-bit *window size* field, there is no way to express such a size. Therefore, a window scale option is proposed, allowing the sender and receiver to negotiate a *window scale* option. This number allows both sides to shift the window size field up to 16 bits to the left, allowing windows up to 232 bytes.
- Another option proposed is the use of the **selective repeat** instead of **go back n** protocol.



TCP Connection Management

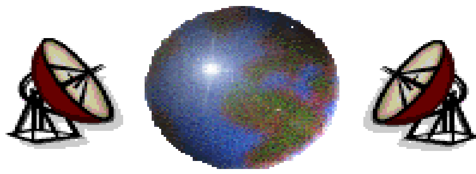
- Connections are established using the three-way handshake.
- To establish a connection, one side, say the **server**, passively waits for an incoming connection by executing the **LISTEN** and **ACCEPT** primitives.
- The other side, say the **client**, executes a **CONNECT** primitive, specifying the IP address and port to which it wants to connect and the maximum TCP segment size it is willing to accept. The **CONNECT** primitive sends a TCP segment with the *SYN* bit **on** and *ACK* bit **off** and waits for response.
- When the segment arrives at the destination (server), the TCP entity 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.



TCP Connection Management

- 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. Note that a *SYN* segment consumes 1 byte of sequence space so it can be acknowledged unambiguously.

- If two hosts simultaneously attempt to establish a connection between the same two sockets, only one connection is established.



TCP Connection Management

Host 1

Host 2

الرغبة في انشاء الاتصال
من جهاز ١ إلى جهاز ٢

Segment1 (SYN=1 , ACK=0)

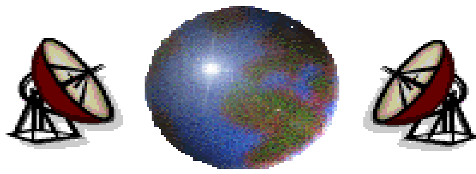
الرغبة في انشاء
الاتصال من جهاز ٢
إلى جهاز ١

اقرار بأنه قد تم انشاء
الاتصال من جهاز ١
إلى جهاز ٢

Segment2 (SYN=1 , ACK=1)

اقرار بأنه قد تم انشاء الاتصال
من جهاز ٢ إلى جهاز ١

Segment3 (SYN=0 , ACK=1)



TCP Connection Management

الرغبة في قطع الاتصال من جهاز ١ إلى جهاز ٢

Segment1 (FIN=1 , ACK=0)

الرغبة في قطع الاتصال من جهاز ٢ إلى جهاز ١

اقرار بأنه قد تم قطع الاتصال من جهاز ١ إلى جهاز ٢

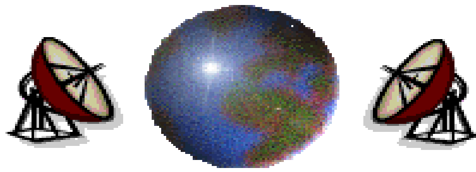
Host 1

Host 2

Segment2 (FIN=1 , ACK=1)

اقرار بأنه قد تم قطع الاتصال من جهاز ٢ إلى جهاز ١

Segment3 (FIN=0 , ACK=1)



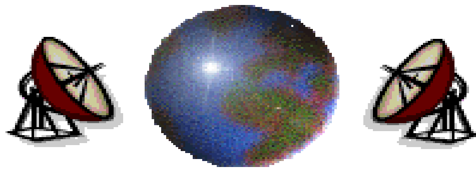
TCP Connection Establishment

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

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

(a) TCP connection establishment in the normal case.

(b) Call collision.

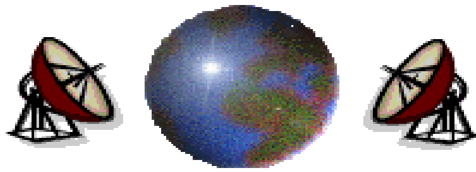


TCP Connection Management

- The initial sequence number on a connection is not 0. A **clock-based** scheme is used, with a clock tick every 4 msec. For additional safety, when a host crashes, it may not reboot for the maximum packet lifetime (120 sec) to make sure that no packets from previous connections are still roaming around the Internet somewhere.

- Although TCP connections are full duplex, connections are released as a pair of simplex connections. Each simplex party can send a TCP segment with the *FIN* bit **set**, which mean that it has no more data to transmit. When a *FIN* is acknowledged, the direction is shut down. Data may continue to flow in the other direction. When both directions have been shut down, the connection is released.

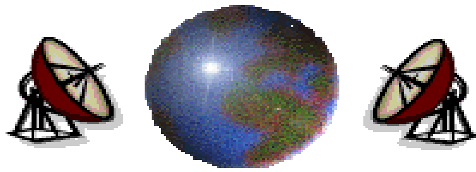
- Normally, four TCP segments are needed to release a connection, one *FIN* and one *ACK* for each direction. However, it is possible for the first *ACK* and the second *FIN* to be contained in the same segment, reducing the total count to three.



TCP Connection Management

- To avoid the two-army problem, timers are used. If a response to a *FIN* is not forthcoming within two maximum packet lifetimes, the sender of the *FIN* releases the connection. The other side will eventually notice that nobody seems to be listening to it any more, and times out as well.

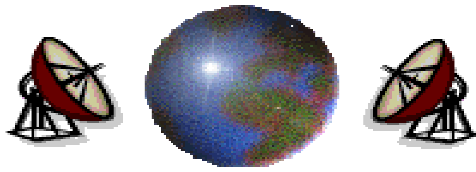
- The steps required to establish and release a connections can be represented in a finite state machine with 11 states.



TCP Connection Management

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
TIMED 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

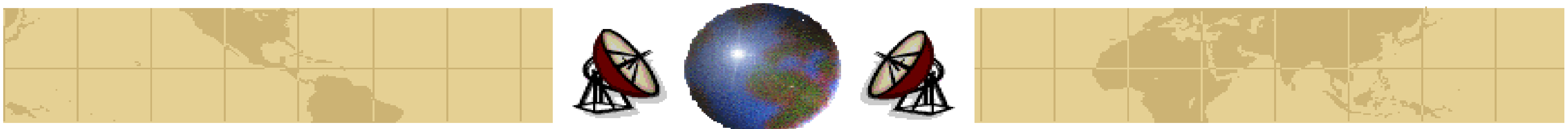
The states used in the TCP connection management finite state machine.



TCP Connection Management

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 by the event causing it and the action resulting from it, separated by a slash.

(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way handshake)
CLOSE/–
LISTEN/–
CLOSE/–
SYN/SYN + ACK
(Step 2 of the 3-way handshake)
LISTEN
RST/–
SEND/SYN
SYN RCVD
SYN SENT
SYN/SYN + ACK
(simultaneous open)
(Data transfer state)
ACK/–
ESTABLISHED
SYN + ACK/ACK
(Step 3 of the 3-way handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
(Active close)
(Passive close)
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
ACK/–
ACK/–
CLOSE/FIN
FIN + ACK/ACK
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
(Timeout/)
ACK/–
CLOSED
(Go back to start)



TCP Transmission Policy

Sender

Receiver

Application does a 2K Write

2K | SEQ = 0

0 4K-1

Empty

Receiver's Buffer

2K

ACK=2048 , WIN=2048

Application does a 2K Write

2K | SEQ=2048

Full

Sender is blocked

ACK=4096 , WIN=0

Application reads 2K

ACK=4096 , WIN=2048

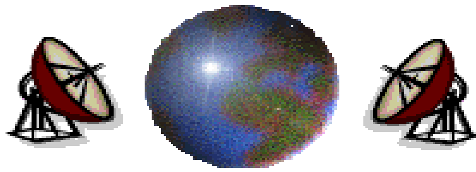
2K

Sender may send up to 2K

1K | SEQ=4096

1K 2K

Window management in TCP.



TCP Transmission Policy

Receiver's buffer is full

Application reads 1 byte

Room for one more byte

Header

Window update segment sent

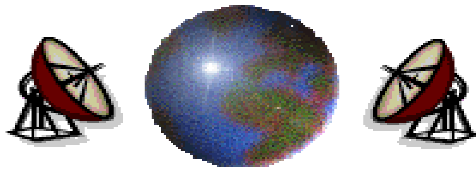
Header

New byte arrives

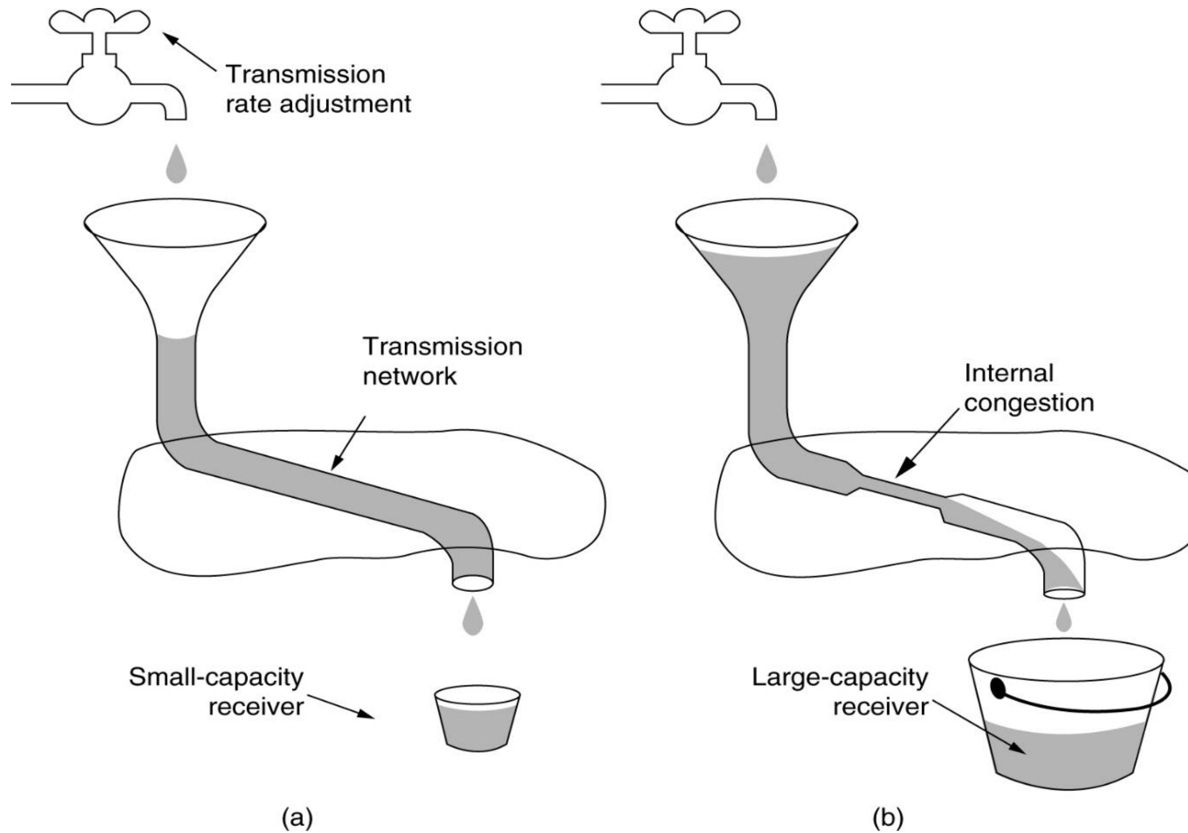
1 Byte

Receiver's buffer is full

Silly window syndrome.

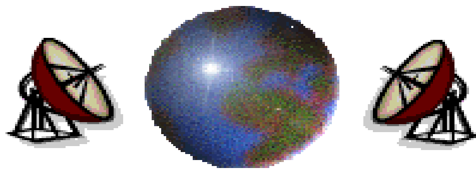


TCP Congestion Control



(a) A fast network feeding a low capacity receiver.

(b) A slow network feeding a high-capacity receiver.



TCP Congestion Control

Timeout
Threshold
Threshold
Congestion window (kilobytes)
0 4 8 12 16 20 24 28 32 36 40 44
0 2 4 6 8 10 12 14 16 18 20 22 24
Transmission number

An example of the Internet congestion algorithm.