SEMESTER – 5

COMPUTER NETWORKS – 21CS52

Module 4

THE TRANSPORT LAYER

1.THE TRANSPORT SERVICE

1. Services Provided to the Upper Layers
2. Transport Service Primitives
3. Berkeley Sockets
4. An Example of Socket Programming: An Internet File Server

2. ELEMENTS OF TRANSPORT PROTOCOLS

1. Addressing
2. Connection Release
3. Error Control and Flow Control
4. Multiplexing
5. Crash Recovery

3. CONGESTION CONTROL

1. Desirable Bandwidth Allocation
2. Regulating the Sending Rate
3. Wireless Issues

4. THE INTERNET TRANSPORT PROTOCOLS: UDP

1. Introduction to UDP
2. Remote Procedure Call
3. Real-Time Transport Protocols

5 THE INTERNET TRANSPORT PROTOCOLS: TCP

1. The TCP Service Model
2. The TCP Protocol
3. The TCP Segment Header
4. The Connection Management
5. TCP Transmission Policy
6. TCP Congestion Control
7. TCP Timer Management.

THE TRANSPORT LAYER

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 is 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. 1.

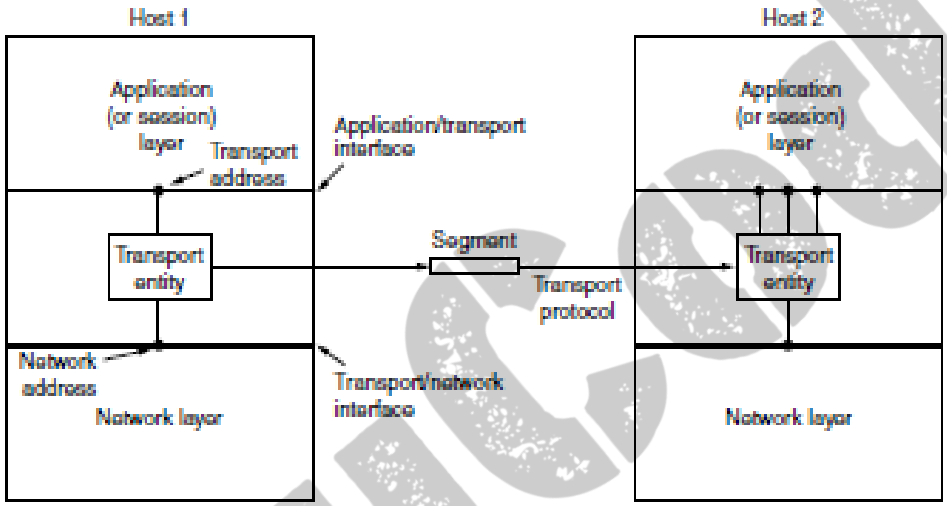


Fig. 1. The network, transport, and application layers.

- **Connection-oriented communication:** It is quite easy for the application to interpret the connection as a data stream instead of having to cope up with the connectionless models that underlie it. For example, internet protocol (IP) and the UDP's datagram protocol.
- **Byte orientation:** Processing the data stream is quite easy when compared with using the communication system format for processing the messages. Because of such simplification, it becomes possible for the applications to work up on message formats that underlie.
- **Same order delivery:** Usually, it is not guaranteed by the transport layer that the data packets will be received in the same order in which they were sent. But this is one of the desired features of the transport layer. Segment numbering is used for incorporating this feature. The data packets are thus passed on to the receiver in order. Head of line blocking is a consequence of implementing this.
- **Reliability:** During transportation some data packets might be lost because of errors and problems such as network congestion. By using error detection mechanism such as CRC (cyclic redundancy check), the data might be checked by the transport protocol for any corruption and for the verification whether the correct reception of the data by either sending

a NACK or an ACK signal to the sending host. Some schemes such as the ARR (automatic repeat request) are sometimes used for the retransmission of the corrupted or the lost data.

- **Flow control:** The rate at which the data is transmitted between two nodes is managed for preventing a sending host with a fast speed from the transmission of data more than what the receiver's data buffer can take at a time. Otherwise, it might cause a buffer overrun.

- **Congestion avoidance:** Traffic entry into the network can be controlled by means of congestion control by avoiding congestive collapse. The network might be kept in a state of congestive collapse by automatic repeat requests.

Transport Service Primitives

A service is specified by a set of primitives. A primitive means operation. To access the service a user process can access these primitives. These primitives are different for connection-oriented service and connectionless service.

There are five types of service primitives:

1. LISTEN: When a server is ready to accept an incoming connection, it executes the LISTEN primitive. It blocks waiting for an incoming connection.
2. CONNECT: It connects the server by establishing a connection. Response is awaited.
3. RECIEVE: Then the RECIEVE call blocks the server.
4. SEND: Then the client executes SEND primitive to transmit its request followed by the execution of RECIEVE to get the reply. Send the message.
5. DISCONNECT: This primitive is used for terminating the connection. After this primitive one can't send any message. When the client sends DISCONNECT packet then the server also sends the DISCONNECT packet to acknowledge the client. When the server package is received by client then the process is terminated.

Connection Oriented Service Primitives

- There are 4 types of primitives for Connection Oriented Service:

CONNECT	**This primitive makes a connection**
DATA, DATA-ACKNOWLEDGE, EXPEDITED-DATA	Data and information is sent using thus primitive
CONNECT	Primitive for closing the connection
RESET	Primitive for resetting the connection

Connectionless Oriented Service Primitives

- There are 2 types of primitives for Connectionless Oriented Service:

UNIDATA	**This primitive sends a packet of data**
FACILITY, REPORT	Primitive for enquiring about the performance of the network, like delivery statistics.

Consider an application with a server and several remote clients.

- To start with, the server executes a LISTEN primitive, typically by calling a library procedure that makes a system call to block the server until a client turns up.
- For lack of a better term, we will reluctantly use the somewhat ungainly acronym TPDU(Transport Protocol Data Unit) for message sent from transport entity to transport entity.
- Thus, TPDUs (exchanged by the transport layer) are contained in packets (exchanged by the network layer).
- In turn, 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.
- When a client wants to talk to the server, it executes a CONNECT primitive.
- The transport entity carries out this primitive by blocking the caller and sending a packet to the server.

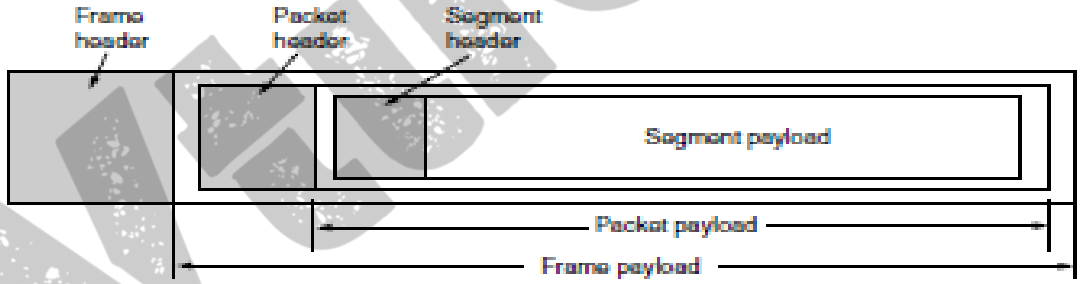


Fig. 2. Nesting of segments, packets and frames

- Encapsulated in the payload of this packet is a transport layer message for the server's transport entity.
- The client's CONNECT call causes a CONNECTION REQUEST TPDU to be sent to the server.
- When it arrives, the transport entity checks to see that the server is blocked on a LISTEN.
- k. It then unblocks 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. Data can now be exchanged using the SEND and RECEIVE primitives.

- In the simplest form, either party can do a(blocking)RECEIVE to wait for the other party to do a SEND. When the TPDU arrives, the receiver is unblocked.
- It can then process the TPDU and send a reply. As long as both sides can keep track of whose turn it is to send, this scheme works fine.
- When a connection is no longer needed, it must be released to free up table space within the two transport entities.

A state diagram for connection establishment and release for these simple primitives is given in Fig.3. 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.

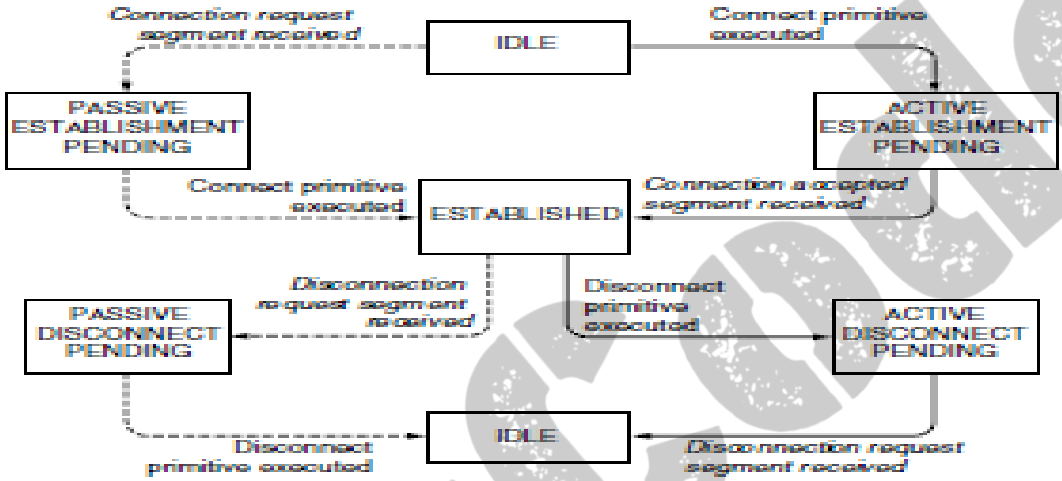


Fig.3. A state diagram for a simple connection management scheme.
Transitions labelled in italics are caused by packet arrivals. The solid lines show.
the client's state sequence. The dashed lines show the server's state sequence.

As an example of the nitty-gritty of how real socket calls are made, consider the client and server code of Fig. 6-6. Here we have a very primitive Internet fileserver along with an example client that uses it. The code has many limitations (discussed below), but in principle the server code can be compiled and run on any UNIX system connected to the Internet. The client code can be compiled and run on any other UNIX machine on the Internet, anywhere in the world. The client code can be executed with appropriate parameters to fetch any file to which the server has access on its machine. The file is written to standard output, which, of course, can be redirected to a file or pipe. Let us look at the server code first. It starts out by including some standard headers, the last three of which contain the main Internet-related definitions and data structures. Next comes a definition of *SERVER PORT* as 12345. This number was chosen arbitrarily. Any number between 1024 and 65535 will work just as well, as long as it is not in use by some other process; ports below 1023 are reserved for privileged users. The next two lines in the server define two constants needed. The first one determines the chunk size in bytes used for the file transfer. The second one determines how many pending connections can be held before additional ones are discarded upon arrival.

Berkeley Sockets

Berkeley sockets is an application programming interface (API) for Internet sockets and Unix domain sockets, used for inter-process communication (IPC). It is commonly implemented as a library of linkable modules. It originated with the 4.2BSD Unix operating system, which was released in 1983.

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

- *socket()* creates a new socket of a certain type, identified by an integer number, and allocates system resources to it.
- *bind()* is typically used on the server side, and associates a socket with a socket address structure, i.e. a specified local IP address and a port number.
- *listen()* is used on the server side, and causes a bound TCP socket to enter listening state.
- *connect()* is used on the client side, and assigns a free local port number to a socket. In case of a TCP socket, it causes an attempt to establish a new TCP connection.
- *accept()* is used on the server side. It accepts a received incoming attempt to create a new TCP connection from the remote client, and creates a new socket associated with the socket address pair of this connection.
- *send(), recv(), sendto(),* and *recvfrom()* are used for sending and receiving data. The standard functions *write()* and *read()* may also be used.
- *close()* causes the system to release resources allocated to a socket. In case of TCP, the connection is terminated.
- *gethostbyname()* and *gethostbyaddr()* are used to resolve host names and addresses. IPv4 only.
- *getaddrinfo()* and *freeaddrinfo()* are used to resolve host names and addresses. IPv4, IPv6.
- *select()* is used to suspend, waiting for one or more of a provided list of sockets to be ready to read, ready to write, or that have errors.
- *poll()* is used to check on the state of a socket in a set of sockets. The set can be tested to see if any socket can be written to, read from or if an error occurred.
- *getsockopt()* is used to retrieve the current value of a particular socket option for the specified socket.
- *setsockopt()* is used to set a particular socket option for the specified socket.

An Example of Socket Programming: An Internet File Server

As an example of the nitty-gritty of how real socket calls are made, consider the client and server code of Fig. 4. Here we have a very primitive Internet file server along with an example client that uses it. The code has many limitations (discussed below), but in principle the server code can be compiled and run on any UNIX system connected to the Internet. The client code can be compiled and run on any other UNIX machine on the Internet, anywhere in the world. The client code can be executed with appropriate parameters to fetch any file to which the server has access on its machine. The file is written to standard output, which, of course, can be redirected to a file or pipe. Let us look at the server code first. It starts out by including some standard headers, the last three of which contain the main Internet-related definitions and data structures. Next comes a definition of *SERVER PORT* as 12345. This number was chosen arbitrarily. Any number between 1024 and 65535 will work just as well, if it is not in use by some other process; ports below 1023 are reserved for privileged users.

The next two lines in the server define two constants needed. The first one determines the chunk size in bytes used for the file transfer. The second one determines show many pending connections can be held before additional ones are discarded upon arrival. After the declarations of local variables, the server code begins. It starts outby initializing a data structure that will hold the server's IP address. This data structure will soon be bound to the server's socket. The call to *memset* sets the data structure to all 0s. The three assignments following it fill in three of its fields. The last of these contains the server's port. The functions *htonl* and *htons* have to do with converting values to a standard format so the code runs correctly on both little-endian machines (e.g., Intel x86) and big-endian machines (e.g., the SPARC). Their exact semantics are not relevant here. Next, the server creates a socket and checks for errors (indicated by $s < 0$). In a production version of the code, the error message could be a trifle more explanatory. The call to *setsockopt* is needed to allow the port to be reused so the server can run indefinitely, fielding request after request. Now the IP address is bound to the socket and a check is made to see if the call to *bind* succeeded. The final step in the initialization is the call to *listen* to announce the server's willingness to accept incoming calls and tell the system to hold up to *QUEUE SIZE* of them in case new requests arrive while the server is still processing the current one. If the queue is full and additional requests arrive, they are quietly discarded.

At this point, the server enters its main loop, which it never leaves. The only way to stop it is to kill it from outside. The call to *accept* blocks the server until some client tries to establish a connection with it. If the *accept* call succeeds, it returns a socket descriptor that can be used for reading and writing, analogous to how file descriptors can be used to read from and write to pipes. However, unlike pipes, which are unidirectional, sockets are bidirectional, so *sa* (the accepted socket) can be used for reading from the connection and also for writing to it. A pipe file descriptor is for reading or writing but not both. After the connection is established, the server reads the file name from it. If the name is not yet available, the server blocks waiting for it. After getting the file name, the server opens the file and enters a loop that alternately reads blocks from the file and writes them to the socket until the entire file has b

Then the server closes the file and the connection and waits for the next connection to show up. It repeats this loop forever. Now let us look at the client code. To understand how it works, it is necessary to understand how it is invoked. Assuming it is called *client*, a typical call is

client flits.cs.vu.nl /usr/tom/filename >f

This call only works if the server is already running on *flits.cs.vu.nl* and the file */usr/tom/filename* exists and the server has read access to it. If the call is successful, the file is transferred over the Internet and written to *f*, after which the client program exits. Since the server continues after a transfer, the client can be started again and again to get other files.

The client code starts with some includes and declarations. Execution begins by checking to see if it has been called with the right number of arguments (*argc*= 3 means the program name plus two arguments). Note that *argv* [1] contains the name of the server (e.g., *flits.cs.vu.nl*) and is converted to an IP address by *gethostbyname*. This function uses DNS to look up the name.

Next, a socket is created and initialized. After that, the client attempts to establish a TCP connection to the server, using *connect*. If the server is up and running on the named machine and attached to *SERVER PORT* and is either idle or has room in its *listen* queue, the connection will (eventually) be established. Using the connection, the client sends the name of the file by writing on the socket. The number of bytes sent is one larger than the name proper, since the 0 byte terminating the name must also be sent to tell the server where the name ends. Now the client enters a loop, reading the file block by block from the socket and copying it to standard output. When it is done, it just exits. The procedure *fatal* prints an error message and exits. The server needs the same procedure, but it was omitted due to lack of space on the page. Since the client and server are compiled separately and normally run on different computers, they cannot share the code of *fatal*.

http://www.pearsonhighered.com/tanenbaum

Just for the record, this server is not the last word in serverdom. Its error checking is meager and its error reporting is mediocre. Since it handles all requests strictly sequentially (because it has only a single thread), its performance is poor. It has clearly never heard about security, and using bare UNIX system calls is not the way to gain platform independence. It also makes some assumptions that are technically illegal, such as assuming that the file name fits in the buffer and is transmitted atomically. These shortcomings notwithstanding, it is a working Internet file server. In the exercises, the reader is invited to improve it. For more information about programming with sockets, see Donahoo and Calvert

```
// This page contains a client program that can request a file from the server program
#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);
}
```

Figure .4.Client code using sockets.

```
// The server responds by sending the whole file.
#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; /* holds 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 */
}
```

}

ELEMENTS OF TRANSPORT PROTOCOLS

Transport protocol similar to data link protocols . Both do error control and flow control .

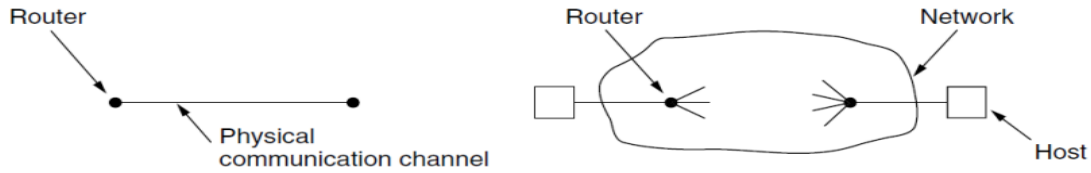


Fig.4. (a) Environment of the data link layer. (b) Environment of the transport layer.

However, significant differences between the two are due to major dissimilarities between the environments in which the two protocols operate, as shown in Fig. 4. At the data link layer, two routers communicate directly via a physical channel, whether wired or wireless, whereas at the transport layer, this physical channel is replaced by the entire network. This difference has many important implications for the protocols.

For one thing, over point-to-point links such as wires or optical fiber, it is usually not necessary for a router to specify which router it wants to talk to—each outgoing line leads directly to a particular router. In the transport layer, explicit addressing of destinations is required. For another thing, the process of establishing a connection over the wire of Fig. 4(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.

Differences between Data link Layer and Transport Layer

	Data link layer	Transport layer
Communication	directly via physical channel	over the entire network
Addressing	no need to specify address. Just select outgoing line	explicit addressing of destination is required
Connection establishment	over a wire is simple	more complicated
Delay	frame either arrives or lost, not stored and delayed	packets might be stored for seconds and delivered later
Buffering and flow control	simpler	more complicated; large and dynamic number of simultaneous connections

The elements of transport protocols are:

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

1. Addressing

When an application (e.g., a user) process wishes to set up a connection to a remote application process, it must specify which one to connect to. (Connectionless transport has the same problem: to whom should each message be sent?) The method normally used is to define transport addresses to which processes can listen for connection requests. In the Internet, these endpoints are called **ports**. We will use the generic term **TSAP** (**Transport Service Access Point**) to mean a specific endpoint in the transport layer. The analogous endpoints in the network layer (i.e., network layer addresses) are not-surprisingly called **NSAPs** (**Network Service Access Points**). IP addresses are examples of transport connection. Application processes, both clients and servers, can attach themselves to a local TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown. The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport endpoints that share that NSAP.

Fig.5 illustrates the relationship between the NSAPs, the TSAPs, and a transport connection. Application processes, both clients and servers, can attach themselves to a local TSAP to establish a connection to a remote TSAP. These connections run through NSAPs on each host, as shown. The purpose of having TSAPs is that in some networks, each computer has a single NSAP, so some way is needed to distinguish multiple transport endpoints that share that NSAP.

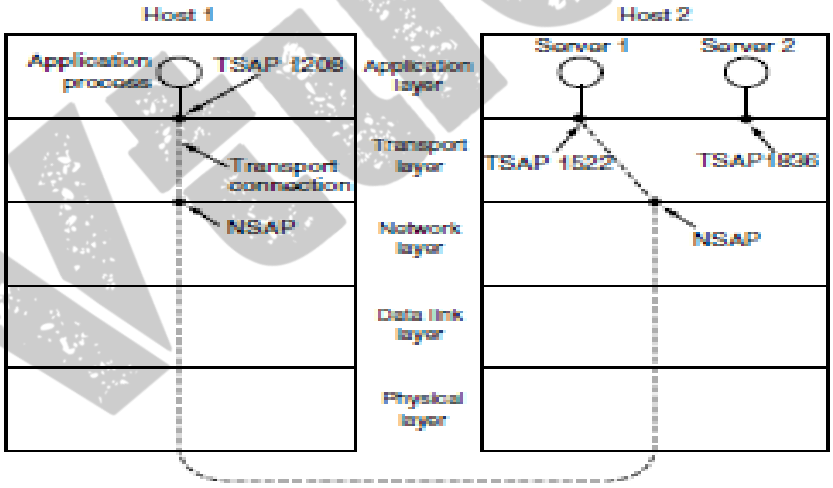


Fig.5. TSAPs, NSAPs, and transport connections

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. How a process attaches itself to a TSAP is outside the networking model and depends entirely on the local operating system. 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

3. connection being established between the application process and the server.
4. The application process sends over the mail message.
5. The mail server responds to say that it will deliver the message.
6. The transport connection is released.

Many of the server processes that can exist on a machine will be used only rarely. It is wasteful to have each of them active and listening to a stable TSAP address all day long. An alternative scheme is shown in Fig. 6 in a simplified form. It is known as the **initial connection protocol**. Instead of every conceivable server listening at a well-known TSAP, each machine that wishes to offer services to remote users has a special **process server** that acts as a proxy for less heavily used servers. This server is called *inetd* on UNIX systems. It listens to a set of ports at the same time, waiting for a connection request. Potential users of a service begin by doing a CONNECT request, specifying the TSAP address of the service they want. If no server is waiting for them, they get a connection to the process server, as shown in Fig. 6(a).

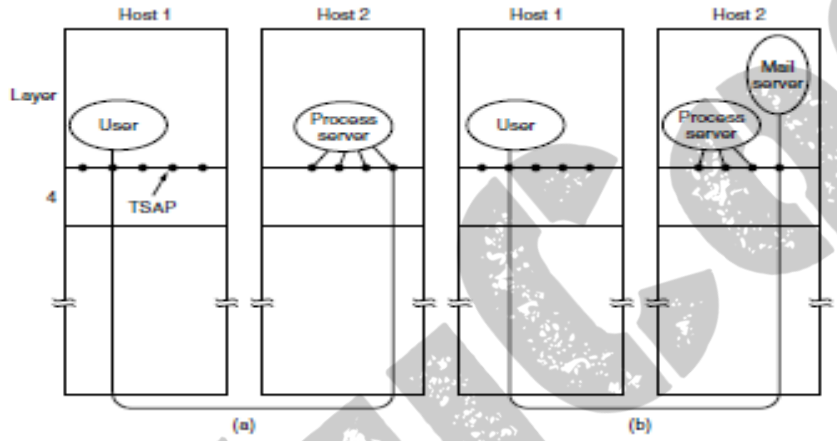


Fig.6 How a user process in host 1 establishes a connection with a mail server in host 2 via a process server.

After it gets the incoming request, the process server spawns the requested server, allowing it to inherit the existing connection with the user. The new server does the requested work, while the process server goes back to listening for new requests, as shown in Fig. 6(b). This method is only applicable when servers can be created on demand.

Connection Establishment

With packet lifetimes bounded, it is possible to devise a fool proof way to establish connections safely. Packet lifetime can be bounded to a known maximum using one of the following techniques:

• Restricted subnet design

• Putting a hop counter in each packet

• Time stamping in each packet

Using a 3-way handshake, a connection can be established. This establishment protocol doesn't require both sides to begin sending with the same sequence number.

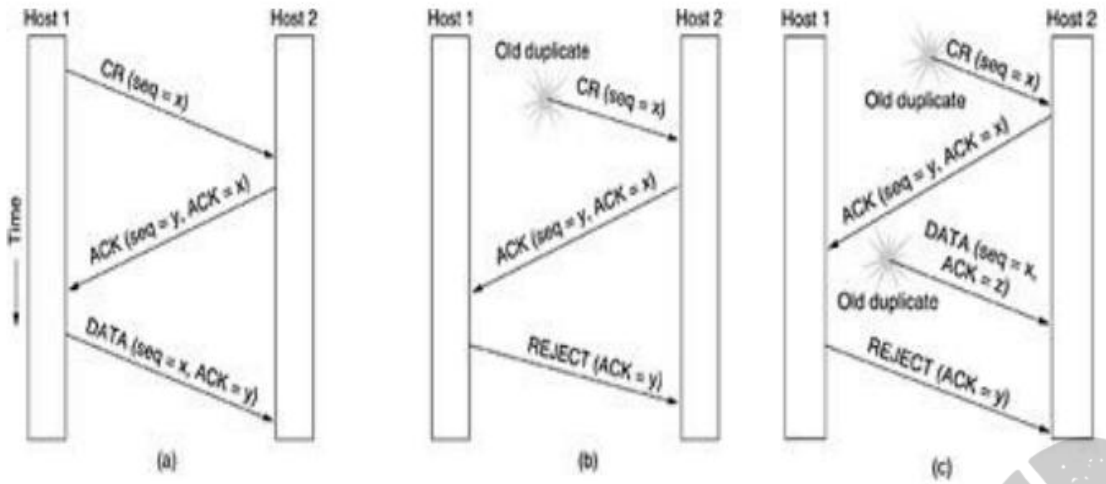


Fig 7: Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST (a) Normal operation. (b) Old duplicate CONNECTION REQUEST appearing out of nowhere. (c) Duplicate CONNECTION REQUEST and duplicate ACK .

- The **first technique** includes any method that prevents packets from looping, combined with some way of bounding delay including congestion over the longest possible path. It is difficult, given that internets may range from a single city to international in scope.
- The **second method** consists of having the hop count initialized to some appropriate value and decremented each time the packet is forwarded. The network protocol simply discards any packet whose hop counter becomes zero.
- The **third method** requires each packet to bear the time it was created, with the routers agreeing to discard any packet older than some agreed-upon time.

In fig (A) Tomlinson (1975) introduced the three-way handshake.

- This establishment protocol involves one peer checking with the other that the connection request is indeed current. Host 1 chooses a sequence number, x , and sends a CONNECTION REQUEST segment containing it to host 2. Host 2replies with an ACK segment acknowledging x and announcing its own initial sequence number, y.
- Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data segment that it sends

In fig (B) the first segment is a delayed duplicate CONNECTION REQUEST from an old connection.

- This segment arrives at host 2 without host 1's knowledge. Host 2 reacts to this segment by sending host1an ACK segment, in effect asking for verification that host 1 was indeed trying to set up a new connection.
- When host 1 rejects host 2's attempt to establish a connection, host 2 realizes that it was tricked by a delayed duplicate and abandons the connection. In this way, a delayed duplicate does no damage.
- The worst case is when both a delayed CONNECTION REQUEST and an ACK are floating around in the subnet.

In fig (C) previous example, host 2 gets a delayed CONNECTION REQUEST and replies to it.

- At this point, it is crucial to realize that host 2 has proposed using y as the initial sequence number for host 2 to host 1 traffic, knowing full well that no segments containing sequence number y or acknowledgements to y are still in existence.
- When the second delayed segment arrives at host 2, the fact that z has been acknowledged rather than y tells host 2 that this, too, is an old duplicate.
- The important thing to realize here is that there is no combination of old segments that can cause the protocol to fail and have a connection set up by accident when no one wants it.

3.CONNECTION RELEASE

A connection is released using either asymmetric or symmetric variant. But, the improved protocol for releasing a connection is a 3-way handshake protocol. There are two styles of terminating a connection:

1) Asymmetric release and

2) 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.

Fig-8(a)	Fig-8(b)	Fig-8(c)	Fig-8(d)
One of the user sends a DISCONNECTION REQUEST TPDU in order to initiate connection release. When it arrives, the recipient sends back a DR-TPDU, too, and starts a timer. When this DR arrives, the original sender sends back an ACKTPDU and releases the connection. Finally, when the ACK-TPDU arrives, the receiver also releases the connection.	Initial process is done in the same way as in fig-(a). If the final ACK-TPDU is lost, the situation is saved by the timer. When the timer is expired, the connection is released.	If the second DR is lost, the user initiating the disconnection will not receive the expected response, and will timeout and starts all over again.	Same as in fig-(c) except that all repeated attempts to retransmit the DR is assumed to be failed due to lost TPDUs. After 'N' entries, the sender just gives up and releases the connection.

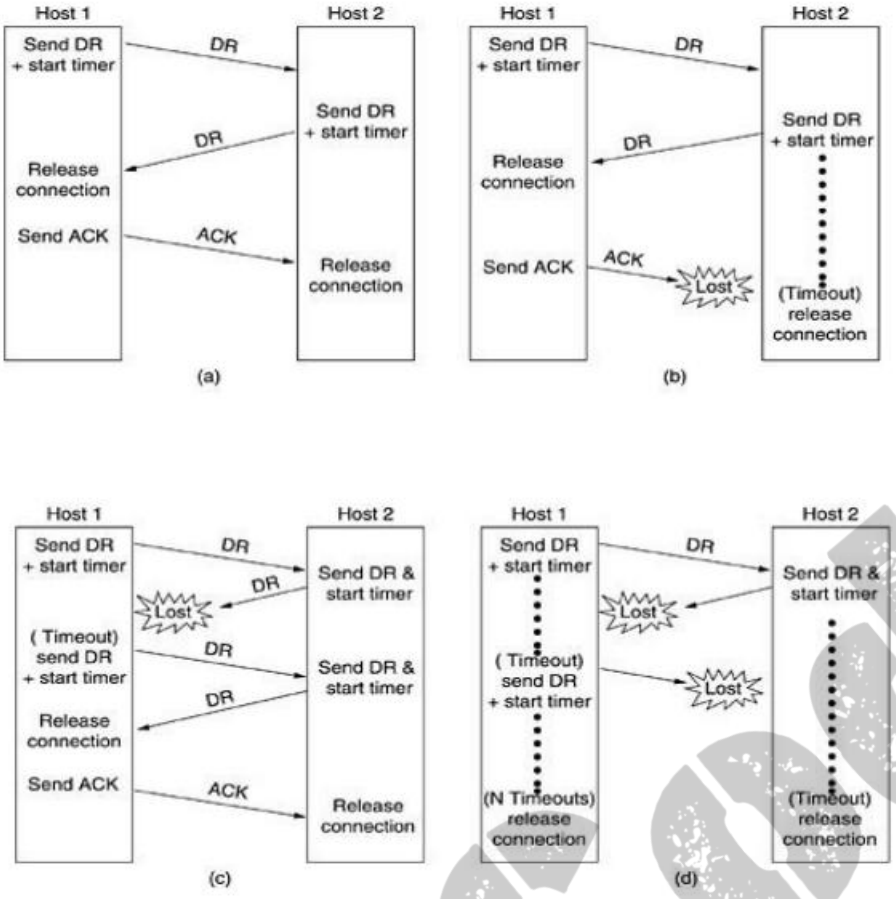


Fig.8: Four protocol scenarios for releasing a connection. (a) Normal case of three-way handshake. (b) Final ACK lost. (c) Response lost. (d) Response lost and subsequent DRs lost.

4.FLOW CONTROL AND BUFFERING:

Flow control is done by having a sliding window on each connection to keep a fast transmitter from over running a slow receiver. Buffering must be done by the sender, if the network service is unreliable. The sender buffers all the TPDUs sent to the receiver. The buffer size varies for different TPDUs.

They are:

a) Chained Fixed-size Buffers
b) Chained Variable-size Buffers
c) One large Circular Buffer per Connection

(a) Chained Fixed-size Buffers: If most TPDUs are nearly the same size, the buffers are organized as a pool of identical size buffers, with one TPDU per buffer.

(b) Chained Variable-size Buffers: This is an approach to the buffer-size problem. i.e., if there is wide variation in TPDU size, from a few characters typed at a terminal to thousands of characters from file transfers, some problems may occur:

- 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. To overcome these problems, we employ variable-size buffers.

(c) One large Circular Buffer per Connection: A single large circular buffer per connection is dedicated when all connections are heavily loaded.

- Source Buffering is used for low band width bursty traffic
- Destination Buffering is used for high band width smooth traffic.
- Dynamic Buffering is used if the traffic pattern changes randomly.

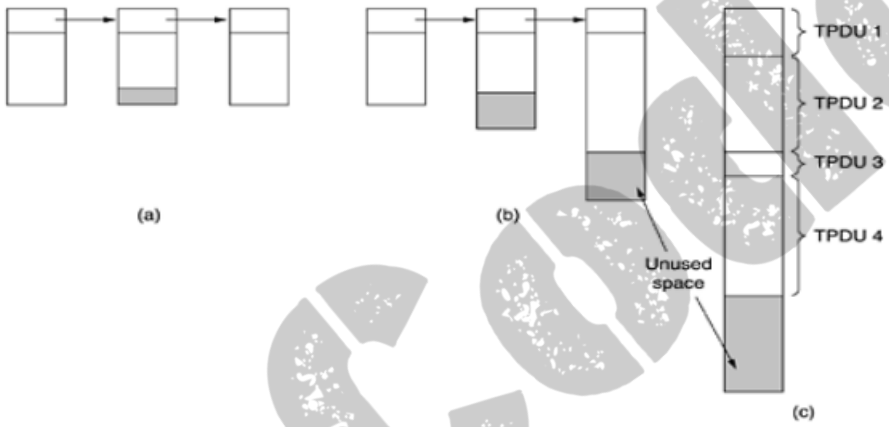


Figure. (a) Chained fixed-size buffers. (b) Chained variable-sized buffers. (c) One large circular buffer per connects

Figure 9 shows an example of how dynamic window management might work in a datagram network with 4-bit sequence numbers. In this example, data flows in segments from host *A* to host *B* and acknowledgements and buffer allocations flow in segments in the reverse direction. Initially, *A* wants eight buffers, but it is granted only four of these. It then sends three segments, of which the third is lost. Segment 6 acknowledges receipt of all segments up to and including sequence number 1, thus allowing *A* to release those buffers, and furthermore informs *A* that it has permission to send three more segments starting beyond 1 (i.e., segments 2, 3, and 4). *A* knows that it has already sent number 2, so it thinks that it may send segments 3 and 4, which it proceeds to do. At this point it is blocked and must wait for more buffer allocation. Timeout-induced retransmissions (line 9), however, may occur while blocked, since they use buffers that have already been allocated. In line 10, *B* acknowledges receipt of all segments up to and including 4 but refuses to let *A* continue. The next segment from *B* to *A* allocates another buffer and allows *A* to continue. This will happen when *B* has buffer space, likely because the transport user has accepted more segment data.

	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

Fig.9. Dynamic buffer allocation. The arrows show the direction of transmission. An ellipsis (...) indicates a lost segment.

Problems with buffer allocation schemes of this kind can arise in datagram networks if control segments can get lost—which they most certainly can. Look at line 16. *B* has now allocated more buffers to *A*, but the allocation segment was lost. Oops. Since control segments are not sequenced or timed out, *A* is now deadlocked. To prevent this situation, each host should periodically send control segments giving the acknowledgement and buffer status on each connection. That way, the deadlock will be broken, sooner or later. Until now we have tacitly assumed that the only limit imposed on the sender's data rate is the amount of buffer space available in the receiver. This is often not the case. Memory was once expensive, but prices have fallen dramatically. Hosts may be equipped with sufficient memory that the lack of buffers is rarely, if ever, a problem, even for wide area connections. Of course, this depends on the buffer size being set to be large enough, which has not always been the case for TCP (Zhang et al., 2002).

When buffer space no longer limits the maximum flow, another bottleneck will appear: the carrying capacity of the network. If adjacent routers can exchange at most *x* packets/sec and there are *k* disjoint paths between a pair of hosts, there is no way that those hosts can exchange more than *kx* segments/sec, no matter how much buffer space is available at each end. If the sender pushes too hard (i.e., sends more than *kx* segments/sec), the network will become congested because it will be unable to deliver segments as fast as they are coming in.

5.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 as well, a user who left a terminal logged into a remote machine, there is need for multiplexing. There are 2 kinds of multiplexing

a) Up-Ward Multiplexing
b) Down-Ward Multiplexing

a) Up-Ward Multiplexing.

In the below figure, all the 4 distinct transport connections use the same network connection to the remote host. When connect time forms the major component of the carrier's bill, it is up to the transport layer to group port connections according to their destination and map each group onto the minimum number of port connections.

b). Down-Ward Multiplexing.

- If too many transport connections are mapped onto the one network connection, the performance will be poor.
- If too few transport connections are mapped onto one network connection, the service will be expensive.
- The possible solution is to have the transport layer open multiple connections and distribute the traffic among them on round-robin basis, as indicated in the below figure: With 'k' network connections open, the effective band width is increased by a factor of 'k'

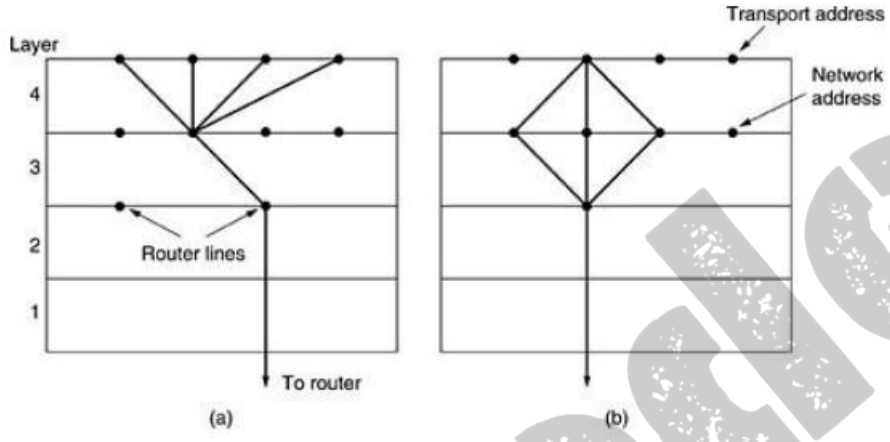


Figure .10 (a) Upward multiplexing. (b) Downward multiplexing

6. Crash Recovery

If hosts and routers are subject to crashes, recovery from these crashes becomes an issue. If the transport entity is entirely within the hosts, recovery from network and router crashes is straightforward.

Problem from host crashes?

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

How to recover from host crashes?

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

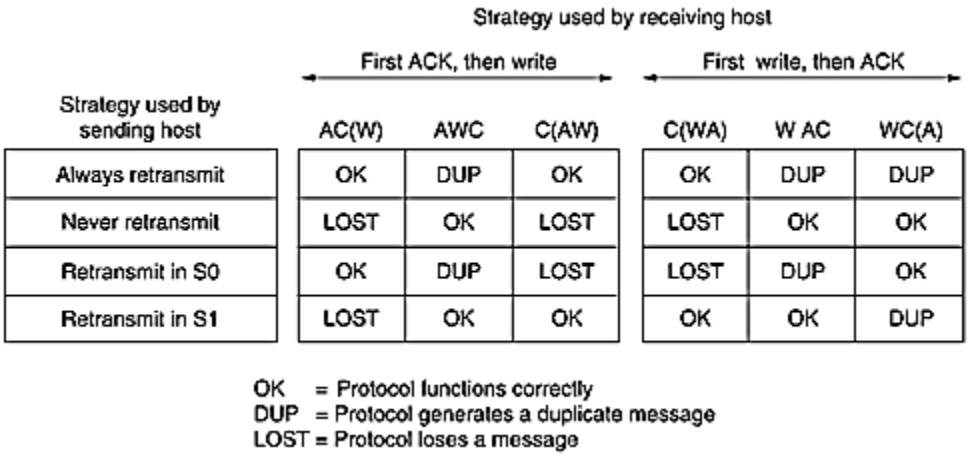
- At first glance it would seem obvious: the client should retransmit only if and only if it has an unacknowledged TPDU outstanding (i.e., is in state S1) when it learns of the crash.

Problem with this approach:

- Consider, for **example,** the situation in which the server's transport entity first sends an acknowledgement, and then, when the acknowledgement has been sent, writes to the application process.
- Writing a TPDU onto the output stream and sending an acknowledgement are two distinct events that cannot be done simultaneously.
- If a crash occurs after the acknowledgement has been sent but before the write has been done, the client will receive the acknowledgement and thus be in state S0 when the crash recovery announcement arrives.
- The client will therefore not retransmit, (incorrectly) thinking that the TPDU has arrived.
- This decision by the client leads to a missing TPDU.
- At this point you may be thinking: "That problem can be solved easily.
- All you have to do is reprogram the transport entity to first do the write and then send the acknowledgement." Try again.
- Imagine that the write has been done but the crash occurs before the acknowledgement can be sent.
- The client will be in state S1 and thus retransmit, leading to an undetected duplicate TPDU in the output stream to the server application process.

Different combinations of client and server strategy:

- The server can be programmed in one of two ways: acknowledge first or write first.
- The client can be programmed in one of four ways: always retransmit the last TPDU, never retransmit the last TPDU, retransmit only in state S0, or retransmit only in state S1.
- This gives eight combinations,
- Three events are possible at the server: **sending an acknowledgement (A), writing to the output process (W), and crashing (C).**
- The three events can occur in six different orderings: **AC(W), AWC, C(AW), C(WA), WAC, and WC(A**), where the parentheses are used to indicate that neither A nor W can follow C (i.e., once it has crashed, it has crashed).
- Figure 11 .shows all eight combinations of client and server strategy and the valid event sequences for each one.
- Notice that for each strategy there is some sequence of events that causes the protocol to fail. For example, if the client always retransmits, the AWC event will generate an undetected duplicate, even though the other two events work properly.



Strategy used by sending host	Strategy used by receiving host: First ACK, then write			Strategy used by receiving host: First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Figure 11. Different combinations of client and server strategies

Making the protocol more elaborate does not help. Even if the client and server exchange several segments before the server attempts to write, so that the client knows exactly what is about to happen, the client has no way of knowing whether a crash occurred just before or just after the write. The conclusion is inescapable: under our ground rules of no simultaneous events—that is, separate.

events happen one after another not at the same time—host crash and recovery cannot be made transparent to higher layers.

Put in more general terms, this result can be restated as ‘‘recovery from a layer *N* crash can only be done by layer *N* + 1,’’ and then only if the higher layer retains enough status information to reconstruct where it was before the problem occurred. This is consistent with the case mentioned above that the transport layer can recover from failures in the network layer, provided that each end of a connection keeps track of where it is.

This problem gets us into the issue of what a so-called end-to-end acknowledgement really means. In principle, the transport protocol is end-to-end and not chained like the lower layers. Now consider the case of a user entering requests for transactions against a remote database. Suppose that the remote transport entity is programmed to first pass segments to the next layer up and then acknowledge. Even in this case, the receipt of an acknowledgement back at the user’s machine does not necessarily mean that the remote host stayed up long enough to actually update the database. A truly end-to-end acknowledgement, whose receipt means that the work has actually been done and lack thereof means that it has not, is probably impossible to achieve. This point is discussed in more detail by **Saltzer et al. (1984).**

TRANSPORT PROTOCOLS – UDP

The Internet has two main protocols in the transport layer, a connectionless protocol and a connection oriented one. The protocols complement each other. The connectionless protocol is UDP. It does almost nothing beyond sending packets between applications, letting applications build their own protocols on top as needed.

The connection-oriented protocol is TCP. It does almost everything. It makes connections and adds reliability with retransmissions, along with flow control and congestion control, all on behalf of the applications that use it. Since UDP is a transport layer protocol that typically runs in the operating system and protocols that use UDP typically run in user s pace, these uses might be considered applications.

INTROUCTION TO UDP

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

➢ UDP transmits segments consisting of an 8-byte header followed by the pay-load. The two ports serve to identify the end-points 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. 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.

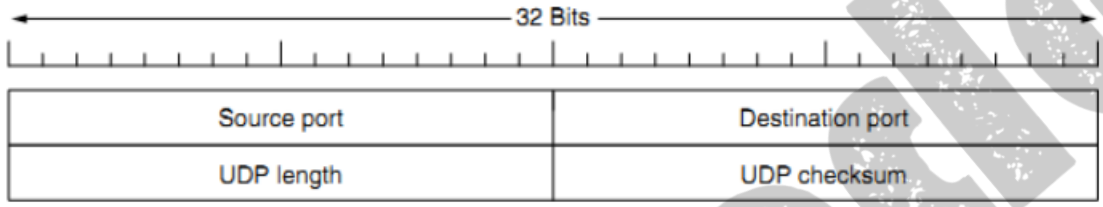


Fig .11: The UDP header

Source port, destination port: Identifies the end points within the source and destination machines. **UDP length:** Includes 8-byte header and the data.

UDP checksum: Includes the UDP header, the UDP data padded out to an even number of bytes if need be. It is an optional field.

REMOTE PROCEDURE CALL

➢ In a certain sense, sending a message to a remote host and getting a reply back is like making a function call in a programming language. This is to arrange request-reply interactions on networks to be cast in the form of procedure calls.

➢ For example, just imagine a procedure named get IP address (host name) that works by sending a UDP packet to a DNS server and waiting or the reply, timing out and trying again if one is not forthcoming quickly enough. In this way, all the details of networking can be hidden from the programmer.

➢ RPC is used to call remote programs using the procedural call. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended and execution of the called procedure takes place on 2.

➢ Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing is visible to the application programmer. This technique is known as RPC (Remote Procedure Call) and has become the basis for many networking applications. Traditionally, the calling procedure is known as the client and the called procedure is known as the server.

➢ 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.

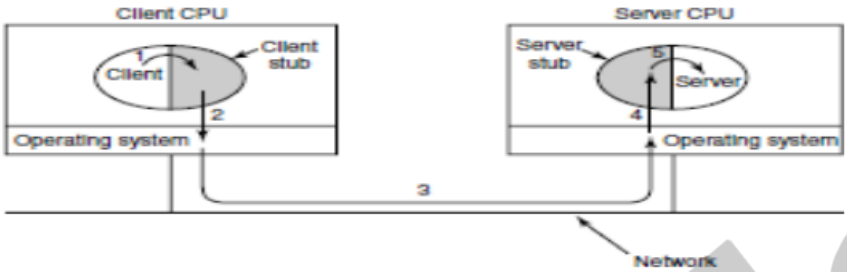


Fig.12 Steps in making a RPC

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

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

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

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

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. In this way, instead of I/O being done on sockets, network communication is done by faking a normal procedure call. With RPC, passing pointers is impossible because the client and server are in different address spaces.

THE INTERNET TRANSPORT PROTOCOLS: TCP

It was specifically designed to provide a reliable end-to end byte stream over an unreliable network. It was designed to adapt dynamically to properties of the inter network and to be robust in the face of many kinds of failures.

Each machine supporting TCP has a TCP transport entity, which accepts user data streams from local processes, breaks them up into pieces not exceeding 64kbytes and sends each piece as a separate IP

datagram. When these datagrams arrive at a machine, they are given to TCP entity, which reconstructs the original byte streams. It is up to TCP to time out and retransmits them as needed, also to reassemble datagrams into messages in proper sequence.

The different issues to be considered are:

1. The TCP Service Model

2. The TCP Protocol

3. The TCP Segment Header

4. The Connection Management

5. TCP Transmission Policy

6. TCP Congestion Control

7. TCP Timer Management.

1.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, called a "PORT" (= TSAP)

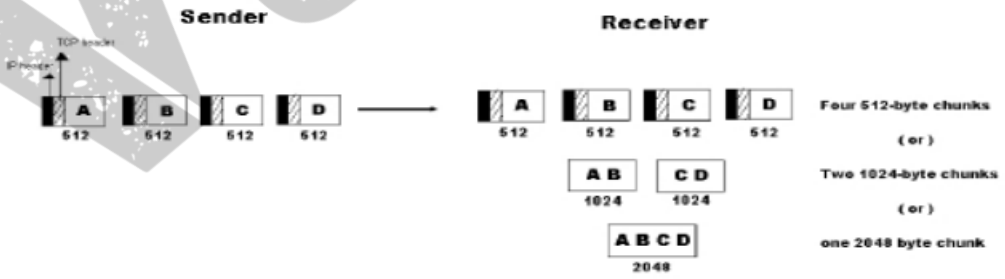
• To obtain TCP service a connection must be explicitly established between a socket on the sending machine and a socket on the receiving machine

• All TCP connections are full duplex and point to point i.e., multicasting or broadcasting is not supported.

• A TCP connection is a byte stream, not a message stream i.e., the dat

a is delivered as chunks.

E.g.: 4 * 512 bytes of data is to be transmitted.



Sockets:

A socket may be used for multiple connections at the same time. In other words, 2 or more connections may terminate at same socket. Connections are identified by socket identifiers at same socket. Connections are identified by socket identifiers at both ends. Some of the sockets are listed below:

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

Ports:

Port numbers below 256 are called Well- known ports and are reserved for standard services.

Eg:

PORT-21	To establish a connection to a host to transfer a file using FTP
PORT-23	To establish a remote login session using TELNET

2 The TCP Protocol

- A key feature of TCP, and one which 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 basic protocol used by TCP entities is the sliding window protocol.
- 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, otherwise without data) bearing an acknowledgement number equal to the next sequence number it expects to receive.
- If the sender's timer goes off before the acknowledgement is received, the sender transmits the segment again.

3.The TCP Segment Header

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.

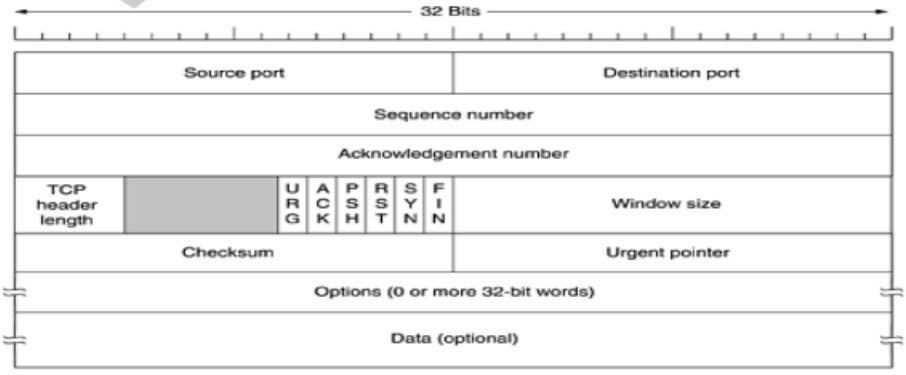


Fig.13.The TCP Header

Source Port, Destination Port : Identify local end points of the connections

Sequence number: Specifies the sequence number of the segment.

Acknowledgement Number: Specifies the next byte expected.

TCP header length: Tells how many 32-bit words are contained in TCP header

URG: It is set to 1 if URGENT pointer is in use, which indicates start of urgent data.

ACK: It is set to 1 to indicate that the acknowledgement number is valid.

PSH: Indicates pushed data.

RST: It is used to reset a connection that has become confused due to reject an invalid segment or refuse an attempt to open a connection.

FIN: Used to release a connection. SYN: Used to establish connections.

4.TCP Connection Establishment

To establish a connection, one side, say, the server, passively waits for an incoming connection by executing the LISTEN and ACCEPT primitives, 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.

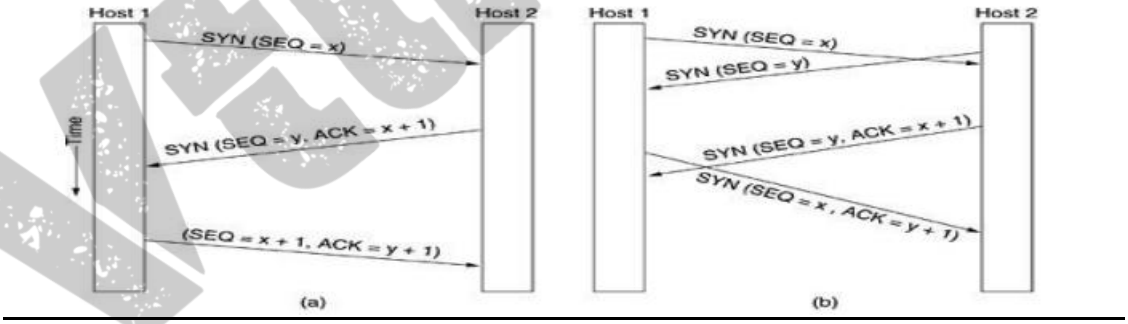


Fig.14.: a) TCP Connection establishment in the normal case b) Call Collision

5.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.
- 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.

6.TCP Connection Management Modelling

The steps required establishing and release connections can be represented in a finite state machine with the 11 states listed in Fig. 4.13. 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.

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

Fig.15. The states used in the TCP connection management finite state machine.

1. TCP Connection management from server's point of view:
2. The server does a LISTEN and settles down to see who turns up.
3. When a SYN comes in, the server acknowledges it and goes to the SYNRCVD state
4. When the servers SYN is itself acknowledged the 3-way handshake is complete and server goes to the ESTABLISHED state. Data transfer can now occur.
5. When the client has had enough, it does a close, which causes a FIN to arrive at the server [dashed box marked passive close].
6. The server is then signalled.
7. When it too, does a CLOSE, a FIN is sent to the client.
8. When the client's acknowledgement shows up, the server releases the connection and deletes the connection record

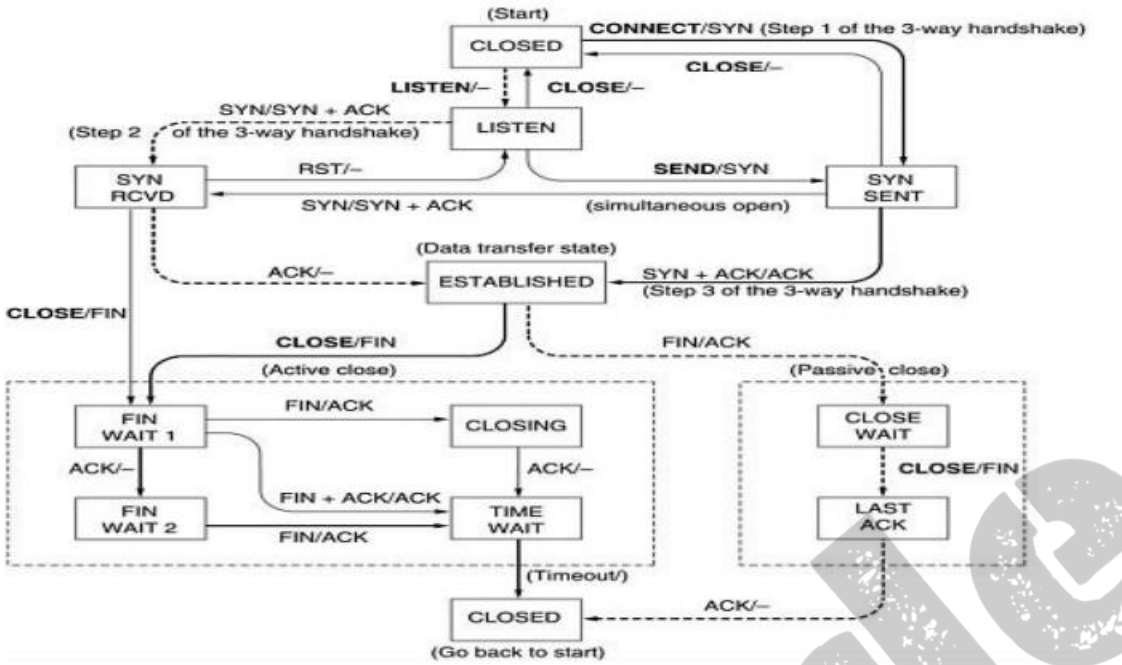


Figure 16. TCP connection management finite state machine. The heavy solid line is the normal path for a client. The heavy dashed line is the normal path for a server. The light lines are unusual events. Each transition is labelled with the event causing it and the action resulting from it, separated by a slash.

The finite state machine itself is shown in Fig. 16. 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 Each line in Fig. 16 is marked by an *event/action* pair. The event can either be a user-initiated system call (CONNECT, LISTEN, SEND, or CLOSE), a segment arrival

(*SYN*, *FIN*, *ACK*, or *RST*), or, in one case, a timeout of twice the maximum packet lifetime. The action is the sending of a control segment (*SYN*, *FIN*, or *RST*) or nothing, indicated by —. Comments are shown in parentheses.

One can best understand the diagram by first following the path of a client (the heavy solid line), then later following the path of a server (the heavy dashed line). When an application program on the client machine issues a CONNECT request, the local TCP entity creates a connection record, marks it as being in the *SYN SENT* state, and shoots off a *SYN* segment. Note that many connections may be open (or being opened) at the same time on behalf of multiple applications, so

the state is per connection and recorded in the connection record. When the *SYN+ACK* arrives, TCP sends the final *ACK* of the three-way handshake and switches into the *ESTABLISHED* state. Data can now be sent and received.

When an application is finished, it executes a CLOSE primitive, which causes the local TCP entity to send a *FIN* segment and wait for the corresponding *ACK* (dashed box marked ‘‘active close’’). When the *ACK* arrives, a transition is made to the state *FIN WAIT 2* and one direction of the connection is closed. When the other side closes, too, a *FIN* comes in, which is acknowledged. Now both sides are closed, but TCP waits a time equal to twice the maximum packet lifetime to guarantee that all packets from the connection have died off, just in case the acknowledgement was lost. When the timer goes off, TCP deletes the connection record.

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

7.TCP Sliding Window

What is a TCP Window?

A TCP window is the amount of unacknowledged data a sender can send on a particular connection before it gets an acknowledgment back from the receiver, that it has received some of the data.

TCP Sliding Window

The working of the TCP sliding window mechanism can be explained as below.

The sending device can send all packets within the TCP window size (as specified in the TCP header) without receiving an ACK, and should start a timeout timer for each of them.

The receiving device should acknowledge each packet it received, indicating the sequence number of the last well-received packet. After receiving the ACK from the receiving device, the sending device slides the window to right side.

In this case, the sending device can send up to 5 TCP Segments without receiving an acknowledgement from the receiving device. After receiving the acknowledgement for Segment 1 from the receiving device, the sending device can slide its window one TCP Segment to the right side and the sending device can transmit segment 6 also.

If any TCP Segment lost while its journey to the destination, the receiving device cannot acknowledge the sender. Consider while transmission, all other Segments reached the destination except Segment 3. The receiving device can acknowledge up to Segment 2. At the sending device, a timeout will occur and it will re-transmit the lost Segment 3. Now the receiving device has received all the Segments, since only Segment 3 was lost. Now the receiving device will send the ACK for Segment 5, because it has received all the Segments to Segment 5.

Acknowledgement (ACK) for Segment 5 ensures the sender the receiver has successfully received all the Segments up to 5.

TCP uses a byte level numbering system for communication. If the sequence number for a TCP segment at any instance was 5000 and the Segment carry 500 bytes, the sequence number for the next Segment will be 5000+500+1. That means TCP segment only carries the sequence number of the first byte in the segment.

The Window size is expressed in number of bytes and is determined by the receiving device when the connection is established and can vary later. You might have noticed when transferring big files from one Windows machine to another, initially the time remaining calculation will show a large value and will come down later.

We have four categories in above example.

1) Bytes already sent and acknowledged (up to Byte 20).

2) Bytes sent but not acknowledged (Bytes 21-24).

3) Bytes the receiver is ready to accept (Bytes 25-28).

4) Bytes the receiver is not ready to accept (Byte 29 onwards).

The Send Window is the sum of Bytes sent but not acknowledged and Bytes the receiver is ready to accept (Usable Window).

8. TCP CONGESTION CONTROL:

TCP does to try to prevent the congestion from occurring in the first place in the following way: When a connection is established, a suitable window size is chosen and the receiver specifies a window based on its buffer size. If the sender sticks to this window size, problems will not occur due to buffer overflow at the receiving end. But they may still occur due to internal congestion within the network. Let's see this problem occurs.

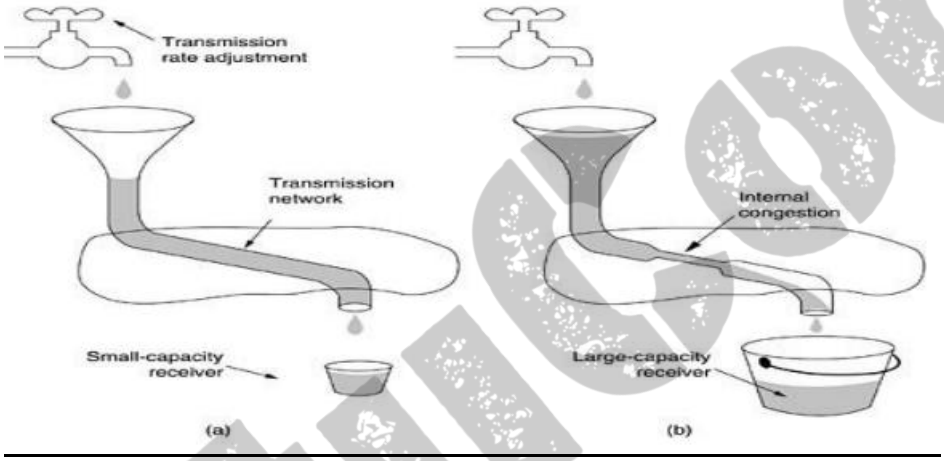


Fig.17.(a) A fast network feeding a low-capacity receiver. (b) A slow network feeding a high-capacity receiver.

In fig (a): We see a thick pipe leading to a small- capacity receiver. As long as the sender does not send more water than the bucket can contain, no water will be lost.

In fig (b): The limiting factor is not the bucket capacity, but the internal carrying capacity of the n/w. if too much water comes in too fast, it will backup and some will be lost.

- When a connection is established, the sender initializes the congestion window to the size of the max segment in use our connection.
- It then sends one max segment .if this max segment is acknowledged before the timer goes off, it adds one segment s worth of bytes to the congestion window to make it two maximum size segments and sends 2 segments.
- As each of these segments is acknowledged, the congestion window is increased by one max segment size.

- When the congestion window is 'n' segments, if all 'n' are acknowledged on time, the congestion window is increased by the byte count corresponding to 'n' segments.
- The congestion window keeps growing exponentially until either a time out occurs or the receiver's window is reached.
- The internet congestion control algorithm uses a third parameter, the "threshold" in addition to receiver and congestion windows.

Different congestion control algorithms used by TCP are:

- RTT variance Estimation.
- Exponential RTO back-off.
- Karn's Algorithm.
- Slow Start .
- Dynamic window sizing on congestion.
- Fast Retransmit .
- Fast Recovery.