UNIT 2 TCP/UDP

Structure **Page Nos.**

2.0 INTRODUCTION

In the previous unit, we discussed the fundamentals of the transport layer which covered topics related to the quality of services, addressing, multiplexing, flow control and buffering. The main protocols which are commonly used such as TCP (Transmission Control Protocol and UDP (User Diagram Protocol) were also discussed. TCP is the more sophisticated protocol of the two and is used for applications that need connection establishment before actual data transmission. Applications such as electronic mail, remote terminal access, web surfing and file transfer are based on TCP.

UDP is a much simpler protocol than TCP because, it does not establish any connection between the two nodes. Unlike TCP, UDP does not guarantee the delivery of data to the destination. It is the responsibility of application layer protocols to make UDP as reliable as possible.

In this unit we will go into details on these two protocols.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- list and explain the various services provided by the Transport Layer;
- establish and release TCP connections;
- describe TCP and UDP header formats;
- describe TCP Flow Control mechanism and how it is different from data link layer, and
- discuss the Congestion mechanism in TCP.

2.2 SERVICES PROVIDED BY INTERNET TRANSPORT PROTOCOLS

The Internet provides two service models: TCP and UDP. The selection of a particular service model is left to the application developers. TCP is a connection oriented and reliable data transfer service whereas UDP is connectionless and provides unreliable data transfer service.

2.2.1 TCP Services

When an application invokes TCP for its transport protocol, the application receives the following services from it:

- Connection oriented service
- Reliable transport service
- Congestion-control mechanism.

Now let us describe these services in brief:

Connection-oriented service: As you are aware from your understanding of the previous unit that the connection oriented service is comprised of the handshake procedure which is a full duplex connection in that, two processes can send messages to each other over the connection at the same time. When the application has finished sending the message, it must remove the connection. The service is referred to as a "connection oriented" service. We are also aware from the discussion on the network layer that this service is implemented through the virtual circuit mechanism.

Reliable transport service: Communicating processes can rely on TCP to deliver all the data that is sent, without error and in the proper order. When one side of the application passes a stream of bytes into a socket, it can count on TCP to deliver the same stream of data to the receiving socket, with no missing or duplicate bytes. Reliability in the Internet is achieved with the use of acknowledgement and retransmissions.

Congestion-control mechanism: TCP also includes a congestion-control mechanism for the smooth functioning of Internet processes. When the packet load offered to the network exceeds its handling capacity congestion builds up.

One point to be noted is that, although, the Internet connection oriented service is bundled with suitable data transfer, flow control and congestion control mechanisms, they are not the essential components of the connection oriented service [Ref. 2].

A connection oriented service can be provided with bundling these services through a different type of a network.

Now we will look at the services TCP does not provide? Some of these are:

i) It does not guarantee a minimum transmission rate,

ii) It does not provide any delay guarantee. But it guarantees delivery of all data. However, it provides no guarantee to the rate of data delivery.

2.2.2 UDP Services

Some of the important features of UDP are as follows:

i) UDP is connectionless, so there is no hand shaking before the two processes start communications.

ii) It provides unreliable data transfer service therefore, there is no guarantee that the message will reach. Due to this, the message may arrive as the receiving process at a random time.

iii) UDP does not provide a congestion-control service. Therefore, the sending process can pump data into a UDP socket at any rate it pleases. Then why do we require such a protocol at the transport layer? There are certain types of applications such as real time. Real-time applications are applications that can tolerate some loss but require a minimum rate. Developers of real-time applications often choose to run their applications over the UDP because their applications cannot wait for acknowledgements for data input. Now coming back to TCP, since it guarantees that all the packets will be delivered to the host, many application protocols, layer protocols are used for these services. For example, SMTP (E-mail), Telnet, HTTP, FTP exclusively uses TCP whereas NFS and Streaming Multimedia may use TCP or UDP. But Internet telephony uses UDP exclusively.

2.3 INTRODUCTION TO UDP

So you might have guessed from the previous section, that UDP is and unreliable transport protocol. Apart from multiplexing /demultiplexing and some error correction UDP adds little to the IP protocol. In this section, we take a look at UDP, at how it works and at what it does.

In case, we select UDP instead of TCP for application development, then the application is almost directly talking to the IP layer. UDP takes messages from the application process, attaches the source and destination port number fields for the multiplexing/demultiplexing service, adds two other small fields, and passes the resulting segment to the network layer. Without establishing handshaking between sending and receiving transport-layer entities, the network layer encapsulates the segment into an IP datagram and then makes a **best-effort** attempt to deliver the segment to the receiving host. If the segment arrives at the receiving host, UDP uses the destination port number to deliver the segment's data to the desired application process.

So you might ask a question then why is UDP required? TCP should be the choice for all types of application layer/protocol. DNS stands for domain name system. It provides a directory service for the internet. It is commonly used by other application protocols (HTTP, FTP etc.) to translate user given host names to IP address. But before we answer your question let-us look at another application called the DNS, which runs exclusively in UDP only but unlike other protocols DNS is not an application with which users interact directly. Instead DNS is a core Internet function that translates the host name to IP addresses. Also unlike other protocols, DNS typically uses UDP. When the DNS application in a host wants to make a query, it constructs a DNS query message and passes the message to the UDP. Without performing any handshaking with the UDP entity running on the destination end system, UDP adds header fields to the message and passes the resulting segment to the network layer. The network layer encapsulates the UDP segment into a datagram and sends the datagram to a name server. The DNS application at the querying host then waits for a reply to its query. If it doesn't receive a reply (possibly because the underlying network lost the query or the reply), either it tries sending the query to another name server, or it informs the invoking application that it can't get a reply.

Like DNS, there are many applications, which are better suited for UDP for the following reasons [Ref 2]:

- *No connection establishment:* Since UDP does not cause any delay in establishing a connection, this is a probably the principal reason why DNS runs

over UDP rather than TCP-DNS which would be much slower if it runs over TCP. HTTP uses TCP rather than UDP, since reliability is critical for Web pages with text.

- *More Client support:* TCP maintains connection state in the end systems. This connection state includes receiving and sending buffers, congestion-control parameters, and sequence and acknowledgement number parameters. We will see in Section 3.5 that this state information is needed to implement TCP's reliable data transfer service and to provide congestion control. UDP, on the other hand, does not maintain connection state and does not track any of these parameters. A server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP. Because UDP, (unlike TCP) does not provide reliable data service and congestion control mechanism, therefore, it does not need to maintain and track the receiving and sending of buffers (connection states), congestion control parameters, and sequence and acknowledgement number parameters.

- *Small packet header overhead.* The TCP segment has 20 bytes of header over-head in every segment, whereas UDP has only eight bytes of overhead.

Now let us examine the application services that are currently using the UDP protocol. For example, remote file server, streaming media, internet telephony, network management, routing protocol such as RIP and, of course, DNS use UDP. Other applications like e-mail, remote terminal access web surfing and file transfer use TCP. Please see reference [2] for further details.

Before discussing the UDP segments structure, now, let us try to answer another question. Is it possible to develop a reliable application on UDP? Yes, it may be possible to do it by adding acknowledgement and retransmission mechanisms, at the application level. Many of today's proprietary streaming applications do just this – they run over UDP, but they have built acknowledgements and retransmissions into the application in order to reduce packet loss. But you understand that it will lead to complete application software design.

UDP Segment Structure

UDP is an end to end transport level protocol that adds only port addresses, checksum error control and length information to the data from the upper layer.

The application data occupies the data field of the UDP segment. For example, for DNS, the data field contains either a query message or a response message. For a streaming audio application, audio samples fill the data field. The packet produced by UDP is called a user datagram.

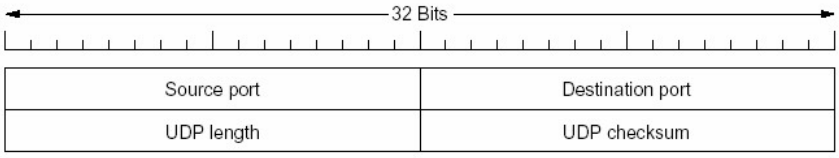


Figure 1: UDP segment structure

The UDP header has only four fields, each consisting of two bytes [*Figure 1*]. Let us discuss each field separately:

- **Source port address**: It is the address of the application program that has created the message.

- **Destination port address:** It is the address of the application program that will receive the message.

- **Total length:** It specifies the length of UDP segment including the header in bytes.

- **Checksum:** The checksum is used by the receiving host to check whether errors have been introduced into the segment. In truth, the checksum is also calculated over a few of the fields in the IP header in addition to the UDP segment.

☞ **Check Your Progress 1**

1) Is it true that if IP is the network layer protocol, so TCP must be the transport layer 1?

……

……

……

2) There are two protocols at the transport layer, which of the two is better?

……

……

……..

2.4 INTRODUCTION TO TCP

TCP is designed to provide reliable communication between pairs of processes (TCP users) across a variety of reliable and unreliable networks and Internets. TCP is stream oriented (Connection Oriented). Stream means that every connection is treated as stream of bytes. The user application does not need to package data in individual datagram as it is done in UDP. In TCP a connection must be established between the sender and the receiver. By creating this connection, TCP requires a VC at IP layers that will be active for the entire duration of a transmission. The data is placed in allocated buffers and transmitted by TCP in segments. In addition, TCP provides two useful facilities for labelling data, push and urgent:

[When the PSH (data push) bit is set, this is an indication that the receiver should pass the data to the upper layer immediately. The URG (Urgent) bit is used to indicate that there is data in this segment that the sending side upper layer entity has marked as urgent. The location of the last byte of this urgent data is indicated by the 16 bit urgent data pointer field 7).

Both IP and UDP treat multiple datagrams belongings to a single transmission as entirely separate units, un-related to each other. TCP on the other hand is responsible for the reliable delivery of entire segments. Every segment must be received and acknowledged before the VC is terminated.

2.5 TCP SEGMENT HEADER

TCP uses only a single type of protocol data unit, called a TCP segment. The header is shown in *Figure 2*. Because one header must perform all protocol mechanisms, it is rather large, with a minimum length of 20 octets. A segment beginning with 9 fixed format 20 byte header may be followed by header option [Ref. 1]. After the options, if any, up to 65,535 – 20 (IP header) – (TCP header) = 65,445 data bytes may follow. A Segment with no data is used, for controlling messages and acknowledgements.

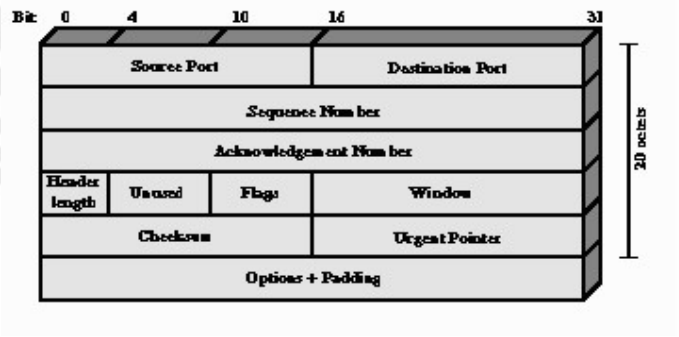


Figure 2: TCP header format

The fields are:

- **Source port (16 bits):** Source service access points and identify local end points of connection.

- **Sequence number (32 bits):** Sequence number of the first data octet in this segment except when SYN is present. If SYN is present, it is the initial sequence number (ISN), and the first data octet is ISN +1.

- **Acknowledgement number (32 bits):** A piggybacked acknowledgement contains the sequence number of the next byte that the TCP entity expects to receive and not for last byte correctly received. Separate number field and acknowledgement number fields are used by the TCP sender and the receiver has to implement a reliable data service.

- **Data offset (4 bits):** Number of 32-bit words in the header.

- **Reserved (6 bits):** Reserved for future use.

- **Flags (6 bits):**

  URG: Used to indicate that there is data in this segment which sends the at the upper layer has marked urgent. The location of the last byte of this urgent data is indicated by the 16 bit urgent data pointer field.

  ACK: Acknowledgement field indicates that the value carried in the ACK field is valid.

  PSH: Push function. The receiver is represented to deliver the data to the application upon arrival, and not buffer it until a full buffer has been received.

  RST: Reset the connection due to host crash or some other reason.

  SYN: Synchronise the sequence numbers.

  FIN: No more data from sender.

- **Receive Window (16 bits):** Used for credit based flow control scheme, in bytes. Contains the number of data bytes beginning with the one indicated in the acknowledgement field that the receiver is willing to accept.

- **Checksum (16 bits):** It provides extra reliability. It Checksums the header, the data and conceptual pseudo header. The Checksum algorithm simply adds up all the 16 bit words in one's complement and then takes one's complement of the sum. As a consequence, when the receiver performs calculations on the entire segment including the checksum field, the result should be 0 [Ref. 1].

- **Urgent data Pointer (16 bits):** Points to the octet following the urgent data; this allows the receiver to know how much urgent data is coming.

- **Options (Variable):** At present, only one option is defined, which specifies the maximum segment size that will be accepted.

Several of the fields in the TCP header warrant further elaboration. The *source port* and *destination port* specify the sending and receiving users of TCP. As with IP, there are a number of common users of TCP that have been assigned numbers; these numbers should be reserved for that purpose in any implementation. Other port numbers must be arranged by agreement between communicating parties.

2.6 TCP CONNECTION ESTABLISHMENT

Before data transmission a connection must be established. Connection establishment must take into account the unreliability of a network service, which leads to a loss of ACK, data as well as SYN. Therefore, the retransmission of these packets have to be done. Recall that a connection establishment calls for the exchange of SYNs.

Figure 3 illustrates typical three-way [Ref. 3] handshake operations.

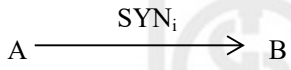
1) Transport entity A (at the sender side) initiates the connection to transport to entity B by setting SYN bit.

2) Transport entity B (at the receiver side) acknowledges the request and also initiates a connection request.

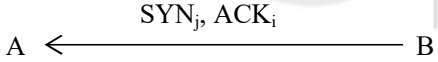
3) A acknowledges to connection request of B and B sends a retransmission and B retransmits.

Now, let us test how this mechanism handles delayed SYN and ACK packets. It is shown that old SYN *X* arrives at B after the close of the relevant connection as shown in *Figure 3* (b). B assumes that this is a fresh request and responds with SYN *j,* ACK *i.* When A receives this message, it realises that it has not requested a connection and therefore, sends an RST, ACK *j*. Note that the ACK *j* portion of the RST message is essential so that an old duplicate RST does not abort a legitimate connection establishment. The final example *Figure 3 (c)* shows a case in which an old SYN, ACK arrives in the middle of a new connection establishment. Because of the use of sequence numbers in the acknowledgements, this event causes no harm.

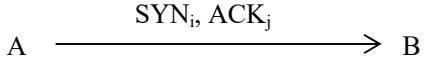
Sender A indicates a connection



Receiver B accepts and acknowledges

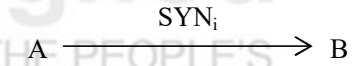


The sender also acknowledges the connection request from the receiver and begins transmission

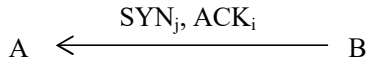


(a) Normal Operation

Obsolete SYN arrives from the previous connection at B



B accepts and acknowledges



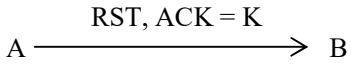
Sender A rejects Receiver B's connection

(b) Delayed arrived of SYN packet

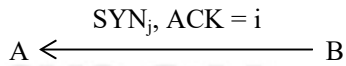
A initiates a connection

old SYN arrives at A

A rejects it



B accepts and acknowledge it



A acknowledges and begins transmission

(c) Delayed SYN and ACK

Figure 3: Three-way handshake mechanism

The three-way handshake procedure ensures that both transport entities agree on their initial sequence number, which must be different. Why do we need different sequence numbers? To answer this, let us assume a case, in which, the transport entities have

picked up the same initial sequence number. After a connection is established, a delayed segment from the previous connection arrives at B, which will be accepted because the initial sequence number turns out to be legal. If a segment from the current connection arrives after some time, it will be rejected by host B, thinking that it is a duplicate. Thus host B cannot distinguish a delayed segment from the new segment.

2.7 TCP CONNECTION TERMINATION

TCP adopts a similar approach to that used for connection establishment. TCP provides for a graceful close that involves the independent termination of each direction of the connection. A termination is initiated when an application tells TCP that it has no more data to send. Each side must explicitly acknowledge the FIN parcel of the other, to be acknowledged. The following steps are required for a graceful close.

- The sender must send FIN *j* and receive an ACK *j*
- It must receive a FIN *j* and send an ACK *j*
- It must wait for an interval of time, to twice the maximum expected segment lifetime.

Acknowledgement Policy

When a data segment arrives that is in sequence, the receiving TCP entity has two options concerning the timing of acknowledgment:

- **Immediate:** When data are accepted, immediately transmit an empty (no data) segment containing the appropriate acknowledgment number.

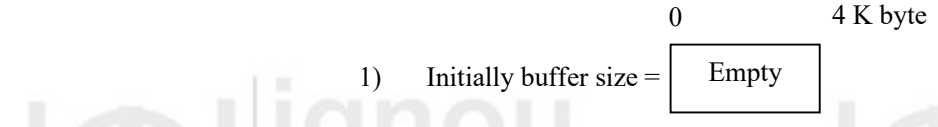
- **Cumulative:** When data are accepted, record the need for acknowledgment, but wait for an outbound segment with data on which to piggyback the acknowledgement. To avoid a long delay, set a window timer. If the timer expires before an acknowledgement is sent, transmit an empty segment containing the appropriate acknowledgement number.

2.8 TCP FLOW CONTROL

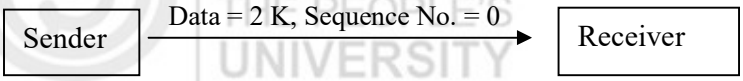
Flow Control is the process of regulating the traffic between two end points and is used to prevent the sender from flooding the receiver with too much data. TCP provides a flow control service to its application to eliminate the possibility of the sender overflowing. At the receivers buffer TCP uses sliding window **with credit scheme** to handle flow control. The scheme provides the receiver with a greater degree of control over data flow. In a **credit scheme** a segment may be acknowledged without the guarantee of a new credit and vice-versa. Whereas in a fixed sliding window control (used at the data link layer), the two are interlinked (tied).

Now, let us understand this scheme with the help of a diagram.

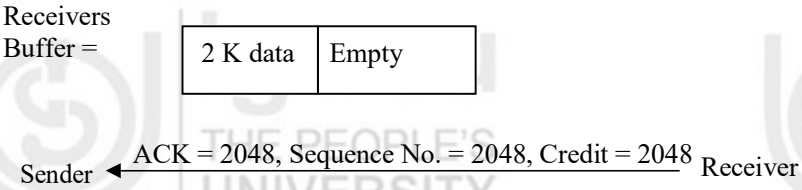
Assume that the sender wants to send application data to the receiver. The receiver has 4 K byte buffer which is empty as shown below:



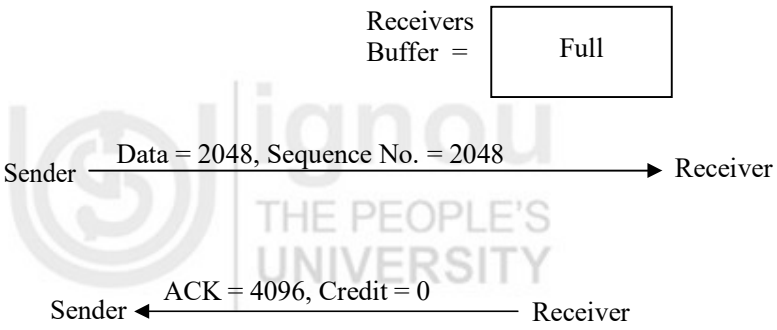
2) Sender transmits 2 K byte segment (data) with sequence number 0 as shown below :



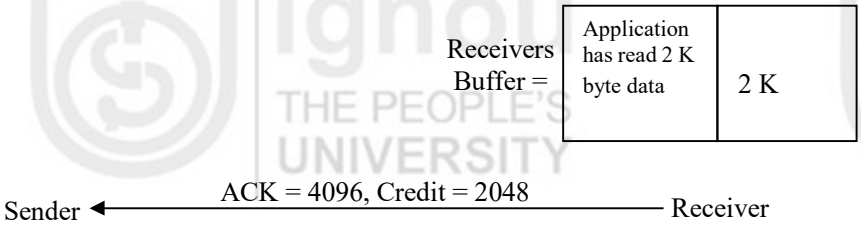
3) The packet is examined at the receiver. After that it will be acknowledged by it. It will also specify the credit value (window size) of the receiver. Until the application process running at the receiver side removes some data from buffer, its buffer size remains fixed at 2048. Therefore, credit value (window size) is 2048 byte.



4) Now the sender transmits another 2048 bytes, which is acknowledged, but the advertised window (credit) is 0.



5) The sender must stop sending data until the application process on the receiving host has removed some data from the buffer, at which time TCP can advertise, a large window (credit value).



When the credit size is zero, normally there is no transmission from the sender side except in two situations:

(i) Urgent data requires to be sent

(ii) Sender wants to know the credit size and the next byte expected by the receiver.

Both senders and receivers can delay the transmission from their side to optimise resources. If a sender knows that the buffer capacity of a receiver window is 8 K and currently it has received just 2 K, then it may buffer it at the sender side till it gets more data from the application process. Similarly, the receiver has to send some data to the sender it can delay the acknowledgement till its data is ready for that the acknowledgement can be **piggybacked.** Therefore, the basic reason of delaying the acknowledgement is to reduce the bandwidth.

Let us consider an example of a log-in session, for example, **Telnet** Session in which the user types one character at a time and the server (log-in server) reacts to every keystroke. You will notice that one character requires four exchanges of IP packets between the log-in session client and a Telnet server which is illustrated below:

Assume that TCP and IP header are 20 bytes each).

1st exchange (at the sender side)

- Whenever application data comes to TCP sender, it creates 20 bytes of a segment, which it gives to IP to create a IP datagram.

Total no. of bytes sent = 20 bytes (TCP) + 20 bytes (IP) + 1 byte for a character (a key stroke) = 41 byte.

2nd exchange (at the receiver side)

- The receiver (Telnet a log-in server) immediately sends an acknowledgement to the sender.

Total no. of bytes sent by the server = 20 bytes (TCP) +20 byte (IP) = 40 bytes. No extra byte is required for an acknowledgement.

3rd exchange (From the log-in server)

- Window update-related message which moves to window one byte right.

Total no. of byte sent – 20bytes (TCP) + 20byte (IP) = 40 bytes.

4th exchange (From the log-in server)

- Echos Character back to the client

Total no. bytes – 20 bytes (TCP) + 20 byte (IP) + 1 byte (0 char) = 41 bytes.

Therefore, the total number of bytes exchanged = 41 + 40 + 40 + 41 = 162 bytes.

So what is the solution to reduce the wastage of bandwidth? The solution has been proposed by **Nagle** and is known as **Nagle's algorithm.**

The algorithm works as follows: When data comes the sender one byte at a time, just send the first byte and buffer all the rest until the outstanding byte is acknowledged. In the meantime, if the application generates some more characters before the acknowledgement arrives, TCP will not transmit the character but buffer them instead. After the acknowledgement arrives TCP transmits all the characters that have been waiting in the buffer in a single segment.

Nagle's algorithm is widely used for the TCP implementation. But in certain cases, the algorithm might not be applicable. For example, in highly interactive applications where every keystroke or a curser movement is required to be sent immediately.

Another problem that wastes network bandwidth is when the sender has a large volume of data to transmit and the receiver can only process its receiver buffer a few bytes at a time. Sooner or later the receiver buffer becomes full. When the receiving application reads a few bytes from the receive buffer, the receiving TCP sends a small advertisement window to the sender, which quickly transmits a small segment and fills the receiver buffer again. This process goes on and on with many small segments being transmitted by the sender for a single application message. This problem is called the **silly window syndrome. It can be avoided if the receiver does not advertise the window until the window size is at least as large as half of the receiver buffer size, or the maximum segment size.** The sender side can cooperate by refraining from transmitting small segments.

To summarise, the silly window syndrome can be explained stepwise as:

Step I: Initially the receiver buffer is full

Step II: Application process reads one byte at the receiver side.

Step III: The TCP running at the receiver sends a window update to the sender.

Step IV: The TCP sender sends another byte.

Step V: The receiver buffer is filled up again.

Steps III, IV and V continue forever. **Clark's** solution is to prevent the receiver from sending a window update for 1 byte. In the solution the receiver window is forced to wait until it has a sufficient amount of space available and then only advertise. Specifically, the receiver should not send a window update until it can handle the maximum segment size it advertised when the connection was established, or its buffer half empty, whichever is smaller.

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

2.9 TCP CONGESTION CONTROL

As discussed in the previous section, TCP provides many services to the application process. Congestion Control is one such service. TCP uses end-to-end Congestion Control rather than the network supported Congestion Control, since the IP Protocol does not provide Congestion released support to the end system. The basic idea of TCP congestion control is to have each sender transmit just the right amount of data to keep the network resources utilised but not overloaded. You are also aware that TCP flow control mechanism uses a sliding-window protocol for end-to-end flow control. This protocol is implemented by making the receiver advertise in its acknowledgement the amount of bytes it is willing to receive in the future, called the *advertised window* to avoid the receiver's buffer from overflow. By looking at the advertised window, the sender will resist transmitting data that exceeds the amount that is specified in the advertised window. However, the advertised window does not prevent the buffers in the intermediate routers from overflowing due to which routers

get overloaded. Because IP does not provide any mechanism to control congestion, it is up to the higher layer to detect congestion and take proper action. It turns out that TCP window mechanism can also be used to control congestion in the network.

The protocols designers have to see that the network should be utilised very efficiently (i.e., no congestion and no underutilisation). If the senders are too aggressive and send too many packets, the network will experience congestion. On the other hand, if TCP senders are too conservative, the network will be underutilised. The maximum amount of bytes that a TCP sender can transmit without congesting the network is specified by another window called the *congestion window*. To avoid network congestion and receiver buffer overflow, the maximum amount of data that the TCP sender can transmit at any time is the minimum of the advertised window (receiver window) and the congestion window. Thus, the effective window is the minimum of what the sender thinks is OK and what the receiver thinks is OK. If the receiver advertises for 8 K window but the sender sends 4 K size of data if it thinks that 8 K will congest the network, then the effective windows size is 4K.

The approach used in TCP is to have each sender limit the rate of data traffic into the network as a function of **perceived network congestion**. If a TCP sender perceives that there is small congestion along the path, then it increases its send rate otherwise it reduce it.

The TCP congestion control algorithm dynamically adjusts the congestion window according to the network state. The algorithm is called slow start. The operation of the TCP congestion control algorithm may be divided into three phases: **slow start, congestion avoidance** and **congestion occurrence**. The first phase is run when the algorithm starts or restarts, assuming that the network is empty. The technique, **slow start,** is accomplished by first setting the congestion window to one maximum-size segment. Each time the sender receives an acknowledgement from the receiver, the sender increases the congestion window by one segment. After sending the first segment, if the sender receives an acknowledgement before a time-out, the sender increases the congestion window to two segments. If these two segments are acknowledged, the congestion window increases to four segments, and so on. As shown in the *Figure 4,* the congestion window size grows exponentially during this phase. The reason for the exponential increase is that slow start needs to fill an empty pipe as quickly as possible. The name "slow start" is perhaps a misnomer, since the algorithm ramps up very quickly.

Slow start does not increase the congestion window exponentially forever, since the network will be filled up eventually. Specifically, slow start stops when the congestion window reaches a value specified as the **congestion threshold**, which is initially set to 65,535 bytes. At this point a **congestion avoidance** (2^{nd} phase) phase takes over. This phase assumes that the network is running close to full utilisation. It is wise for the algorithm to reduce the rate of increase so that it will not overshoot excessively. Specifically, the algorithm increases the congestion window **linearly** rather than exponentially when it tries to avoid congestion. This is realised by increasing the congestion window by one segment for each round-trip time.

Obviously, the congestion window cannot be increased indefinitely. The congestion window stops increasing when TCP detects that the network is congested. The algorithm now enters the third phase.

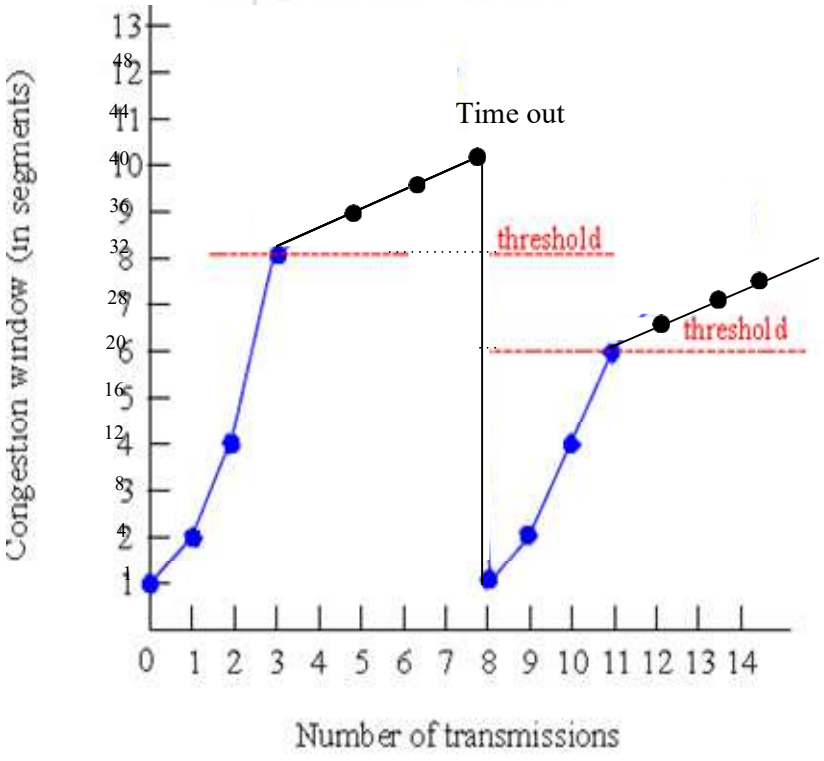


Figure 4: Dynamics of TCP congestion window

At this point the congestion threshold is first set to one-half of the current window size (the minimum of the congestion window and the advertised window, but at least two segments). Next the congestion window is set to one maximum-sized segment. Then the algorithm restarts, using the slow start technique.

How does TCP detect network congestion? There are two approaches:

i) Acknowledgement does not arrive before the timeout because of a segment loss.

ii) Duplicate acknowledgement is required.

The basic assumption the algorithm is making is that, a segment loss is due to congestion rather than errors. This assumption is quite valid in a wired network where the percentage of segment losses due to transmission errors is generally low (less than 1 per cent). Duplicate ACK can be due to either segment reordering or segment loss. In this case the TCP decreases the congestion threshold to one-half of the current window size, as before. However, the congestion window is not reset to one. If the congestion window is less than the new congestion threshold, then the congestion window is increased as in slow start. Otherwise, the congestion window is increased as in congestion avoidance. However, the assumption may not be valid in a wireless network where transmission errors can be relatively high.

The *Figure 4* illustrates the dynamics of the congestion window as time progresses. Thus, assumes that maximum segment size is 1024, threshold to 32K and congestion window to 1 K for the first transmission. The congestion window then grows exponentially (slow start phase) until it hits the threshold (32 K). After that it grows linearly (congestion avoidance phase I). Now, when congestion occurs, the threshold is set to half the current window (20 K) and slow start initiated all over again. In case there is no congestion, the congestion window will continue to grow.

In short, TCP's congestion control algorithm operates as follows:

1) When the Congestion Window is below the threshold, the sender uses for slow start phase and congestion window grows exponentially.

2) When the congestion window is above the threshold the sender is in the congestion avoidance phase and congestion window grows linearly.

You may wish to refer the references [1], [2] and [4] for further clarification on the subject.

2.10 REMOTE PROCEDURE CALL

In this section we will look at two other files access mechanisms such as Network File System and Remote Procedure Call used for accessing files from a server at the client machine. These two mechanism allow programs to call procedures located on remote machines.

Network File System (NFS)

The network file system (NFS) is a file access protocol. FTP and TFTP transfer entire files from a server to the client host. A file access service, on the other hand, makes file systems on a remote machine visible, though they were on your own machine but without actually transferring the files. NFS provides a number of features:

(1) NFS allows you to edit a file on another machine exactly as you would if it were on your own machine.

(2) It even allows you to transfer files from the server to a third host not directly connected to either of you.

Remote Procedure Call (RPC)

NFS works by invoking the services of a second protocol called remote procedure call (RPC). *Figure 5* shows a local procedure call (in this case, a C program calls the open function to access a file stored on a disk).

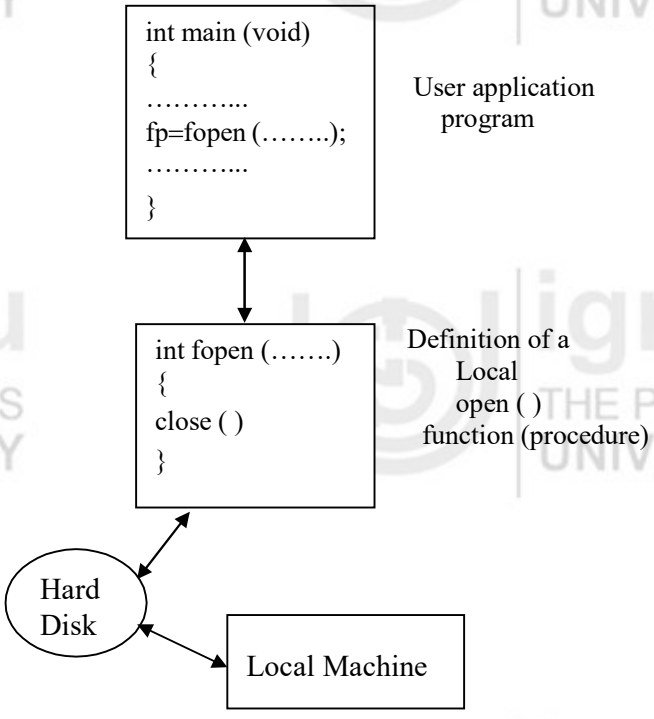


Figure 5: Concept of local procedure call

RPC transfers the procedure call to another machine. Using RPC, local procedure calls are mapped onto appropriate RPC function calls. *Figure 6* illustrates the process: a program issues a call to the NFS client process. The NFS client formats the call for the RPC client and passes it along. The RPC client transforms the data into a different format and provides the interface with the actual TCP/IP transport mechanisms. At the remote host, the RPC server retrieves the call, reformats, and passes it to the NFS server. The NFS server relays the call to the remote disk, which responds as if to a local call and opens the file to the NFS server. The same process is followed in reverse order to make the calling application believe that the file is open on its own host.

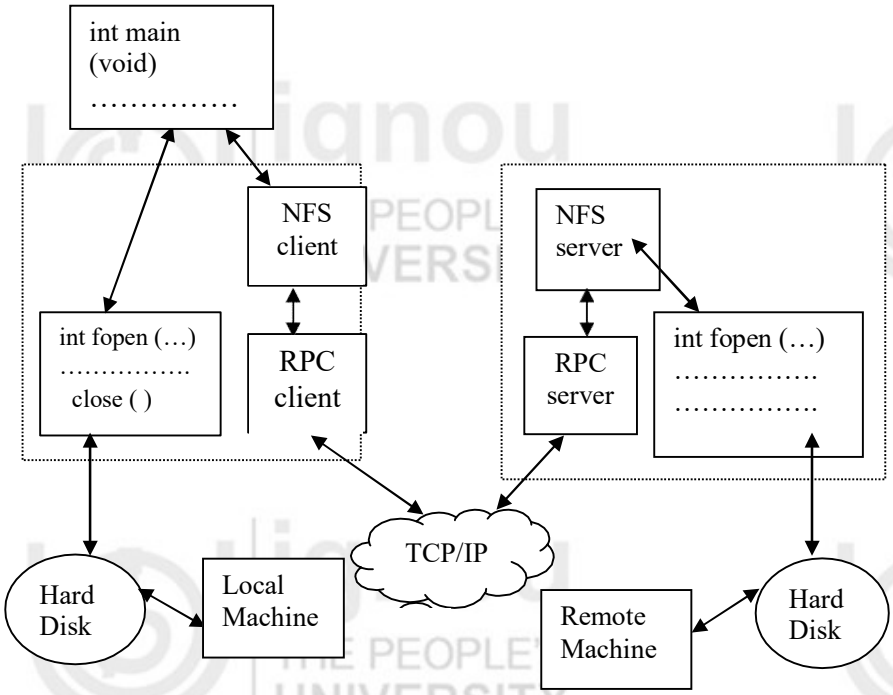


Figure 6: Concept of remote procedure call

There are three independent components in NFS protocol:

(i) NFS itself

(ii) General purpose RPC

(iii) Data representing format (DRF).

In order to make the NFS computer independent what can be done is to use RPC and DRF in other software, (including application), programme by dividing a program into a client side and a server side that use RPC as the chief transport mechanism. On the client side, the transfer procedures may be designated as remote, forcing the compiler to incorporate RPC code into those procedures. On the server side, the desired procedures are implemented by using other RPC facilities to declare them as a part of a server. When the executing client program calls one of the remote procedures, RPC automatically collects values for arguments, from a message, sends the message to the remote server, awaits a response, and stores returned values in the designated arguments. In essence, communication with the remote server occurs automatically as a side-effect of a remote call. The RPC mechanism hides all the details of protocols, making it possible for programmers who know little about the underlying communication protocols but who can write distributed application programs.

☞ **Check Your Progress 2**

1) What is the difference between FTP and NFS?

……

……

……

2) What is a flow control problem? What is the basic mechanism at the transport layer to handle flow control problem?

……

……

……

2.11 SUMMARY

In this unit, we discussed the two transport layer protocols (TCP & UDP) in detail. The transport port can be light weight (very simple) which provide very little facility. In such cases, the application directly talks to the IP. UDP is an example of a transport protocol, which provides minimum facility, with no control, no connection establishment, no acknowledgement and no congestion handling feature. Therefore, if the application is to be designed around the UDP, then such features have to be supported in the application itself. On the other hand, there is another protocol-TCP which provides several feature such as reliability, flow control, connection establishment to the various application services. Nevertheless, the services that the transport layer can provide often constrain the network layer protocol.

We had a close look at the TCP connection establishment, TCP flow control, TCP congestion handling mechanism and finally UDP and TCP header formats. With respect to congestion control, we learned that congestion control is essential for the well-being of the network. Without congestion control the network can be blocked.

2.12 SOLUTIONS/ANSWERS

Check Your Progress 1

1) No, TCP is only part of the TCP/IP transport layer. The other part is UDP (user datagram protocol).

2) It depends on the type of an application. TCP provide a connection-oriented, reliable service (extra overheads). Stream means that the connection is treated as stream of bytes. The user application does not need to package data in individual datagrams.

   Reliable means that every transmission of data is acknowledged by the receiver. UDP offers minimum datagram delivery service.

Check Your Progress 2

1) FTP transfers entire files from a server to the client host. NFS, on the other hand makes file systems on a remote machine visible as they are on your own machine but without actually transferring the files.

2) Flow control is the process of regulating the traffic between points and is used to present the sender from flooding the receiver with too much data.

   The basic mechanism to handle flow control at the transport layer is the sliding window with credit scheme. The credit scheme provides the receiver with a greater degree of control over data flow. In decouples acknowledgement from flow control. In fixed sliding window control such as x.25 and HDLC, the two are synonymous. In a credit scheme, a segment may be acknowledged without granting new credit and vice-versa.

2.13 FURTHER READINGS

1) *Computer Networks,* A.S Tanebaum, 4th Edition, PHI, New Delhi, 2003.

2) *Computer Networking, Computer Networking, A top down approach featuring the Internet,* J.F. Kurose & K.W. Ross, Pearson Edition, New Delhi, 2003.

3) *Data and Computer Communication,* William Stallings, 6th Edition, Pearson Education, New Delhi. 2002.

4) *Communications Networks,* Leon Garcia, and Widjaja, Tata McGraw Hill, 2000.