The Transport Layer

◆ This layer is the heart of the whole protocol hierarchy
  – It provides a reliable, cost-effective end-to-end data transport service *independently* of the physical network(s)

◆ It is important to realise that a *service* is very different to a *protocol*, although they are frequently confused
The complete TCP Transport Service offering

- The TCP Transport Service has the following characteristics:
  - *Connection Orientation*: Already discussed above
  - *Point-To-Point Communication*: Each TCP connection has exactly two endpoints.
  - *Complete Reliability*: TCP guarantees that the data will be delivered exactly as sent i.e. no data missing or out of sequence
  - *Full Duplex Communication*: A TCP connection allows data to flow in either direction
    - TCP buffers outgoing and incoming data
    - This allows applications to continue executing other code whilst the data is being transferred
The complete TCP Transport Service offering

- **Stream Interface**: The source application sends a continuous sequence of octets across a connection
  - The data is passed *en bloc* to TCP for delivery
  - TCP does not guarantee to deliver the data in the same size pieces that it was transferred by the source application.

- **Reliable Connection Startup**: TCP both applications to agree to any new connection

- **Graceful Connection Shutdown**: Either can request a connection to be shut down
  - TCP guarantees to deliver all the data reliably before closing the connection
The Transport Service

◆ The transport service is offered to a user process that exists within the application layer (the transport user)
  – The service is offered through a set of primitives
  – Requests made through these primitives cause the service provider to perform some action

◆ The hardware and/or software within the transport layer that does the work is called the transport entity
The Transport Entity

◆ The *transport entity* can be located in any number of places including:
  – Within the operating system (OS) kernel
  – As a separate user process
  – Within a library package bound to the network application
  – On the network interface card

◆ If the *protocol stack* is located within the OS the primitives are implemented as *system calls*:
  – These calls turn control of the machine over to the OS to send the necessary packets
The Network, Transport and Application Layers

[Diagram showing layers of network with Host 1 and Host 2]
Consider a basic transport interface

This interface allows application programs to establish, use and release connections

The sequence of operations is as follows:

- The server executes a LISTEN primitive
  - This is typically implemented as a library procedure. The transport entity carries out this primitive by blocking the server until a client request arrives
  - A client wishing to talk to the server executes a CONNECT primitive. The remote transport entity blocks the caller and sends a packet to the server
  - Within this packet is a transport entity message to the server's transport entity.
  - These messages are called Transport Protocol Data Units (TPDU)
A set of basic Transport Service primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Packet sent</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTEN</td>
<td>(none)</td>
<td>Block until some process tries to connect</td>
</tr>
<tr>
<td>CONNECT</td>
<td>CONNECTION REQ.</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>DATA</td>
<td>Send information</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>(none)</td>
<td>Block until a DATA packet arrives</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>DISCONNECTION REQ.</td>
<td>This side wants to release the connection</td>
</tr>
</tbody>
</table>
A set of basic Transport Service primitives

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

Diagram:
- Client machine
  - Client process
    - Operating system
    - Kernel
    - Protocol stack
    - Drivers
  - System calls
- Server machine
  - Server process
  - Kernel
  - Protocol stack
  - Drivers

Communication steps:
1. Connect request
2. ACK
3. Request for data
4. Reply
5. Disconnect
6. Disconnect
The interaction between the client and server is as follows:

- **Establishment**
  - The client's CONNECT call causes a CONNECTION REQUEST TPDU to be sent to the server
  - The server transport entity unblocks the server and returns a CONNECTION ACCEPTED TPDU
  - Upon arrival the client is unblocked and the connection is established

- **Data Transfer**
  - Data can now be exchanged using the SEND and RECEIVE primitives
  - Each side must take turns using (blocking) RECEIVE and SEND
Transport Entity interaction

- **Release**
  - Either *transport user* can call a DISCONNECT primitive:
    - This sends a DISCONNECT TPDU to the remote transport entity
    - Upon arrival, the connection is released
Berkeley Sockets

- The basic transport primitives discussed above are not standardised.
- Instead, most OS designers have adopted the socket primitives.
  - These originated from the Berkeley University of California’s UNIX OS which contained the TCP/IP suite of internetworking protocols.
- Consequently, the socket API has become the de facto standard for interfacing to TCP/IP.
Socket Communication and UNIX

- Coming from a UNIX background, sockets use many concepts found in UNIX.
- An application communicates through a socket in the same way that it transfers data to or from a file.
- For this UNIX, uses an open-read-write-close paradigm:
  - An application calls:
    - `open` to prepare a file for reading/writing
    - `read` or `write` to retrieve or send data to/from the file
    - `close` to finish using the file
- When `open` is first called, a descriptor is returned:
  - All future interaction with the file requires this descriptor.
  - Socket communication also uses this descriptor approach.
Socket Communication And UNIX

- To use the TCP/IP protocols to communicate an application must request the OS to create a socket
- The OS returns a descriptor that identifies the socket
- As before this descriptor must be used in all future interactions with the socket
- The following slides lists the socket transport primitives and illustrates an example of how these primitives are used by a client and server application
The Socket Transport Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>BIND</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Announce willingness to accept connections; give queue size</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block the caller until a connection attempt arrives</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>WRITE/SEND</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>READ/RECEIVE</td>
<td>Receive some data from the connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
An Example Client-Server Interaction Using Sockets
The Socket Primitives - explained

- The following primitives are executed by servers:
  - SOCKET: This primitive creates a new end point. It allocates table space for it within the transport entity.
    - This call returns an ordinary file descriptor which is used in all future calls.
  - BIND: This primitive binds a socket to a network address.
    - This allows remote clients to connect to it.
The Socket Primitives - explained

- LISTEN: This primitive allocates space to queue incoming call requests
- ACCEPT: This primitive blocks the server waiting for an incoming connection:
  - Upon receipt of a connection request the transport entity creates a new socket identical to the original one and returns a file descriptor to the server
  - The server forks off a new process or service thread to handle the connection on the new socket
  - The server also continues to wait for more connections on the original socket
The Socket Primitives - explained

- The following primitives are executed by clients:
  - SOCKET: As before
  - CONNECT: This primitive blocks the client and actively starts the connection process

- The following primitives are executed by clients and servers:
  - SEND and RECV: These primitives are used to transmit and receive data over the full-duplex connection
  - CLOSE: This primitive releases the transport connection
In most systems the socket functions are part of the OS

Some systems, however, require a socket library to provide the interface to the transport entity

Both implementations provide the same semantics from a programmer's perspective

However, socket libraries operate differently to a native socket API
  - The code for the library socket procedures are linked into the application program and resides in its address space
  - Calls to a socket library pass control to the library routine

Applications using either implementation can be ported to other computer systems
An Example Client Using Sockets

```c
#include "unp.h"

int main(int argc, char **argv)
{
    int sockfd, n, counter = 0;
    char recvline[MAXLINE + 1];
    struct sockaddr_in servaddr;

    if (argc != 2)
        err_quit("usage: a.out <IPaddress>");

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(13); /* daytime server */
    if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
        err_quit("inet_pton error for \"", argv[1]);

    if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
        err_sys("connect error");

    while ((n = read(sockfd, recvline, MAXLINE)) > 0) {
        counter++;
        recvline[n] = 0; /* null terminate */
        if (fputs(recvline, stdout) == EOF)
            err_sys("fputs error");
    }

    if (n < 0)
        err_sys("read error");

    printf("counter = %d\n", counter);
    exit(0);
}
```