The Transport Layer

- Having examined the TCP/IP Protocol Reference the focus now shifts to the TCP layer.
  - This layer is the heart of the whole protocol hierarchy.
- It sits below the Application Layer providing a ‘Transport’ service to the applications that wish to communicate across a Network.

The Transport Layer

- TCP 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.

Services

- Each layer of the reference model provides a set of functionality to the layer immediately above:
  - This set of functionality is known as “The Services”.
- The layer below is known as the “Service Provider” and the layer above is known as the “Service User”
- The Services are accessed through the interface between the layers.

Protocols

- Protocols on the other hand are how the Services are implemented.
- Typically a Protocol specifies a framing structure which will include a number of fields containing Control data:
  - Generically this framing structure is known as a Protocol Data Unit (PDU)
  - Examples include Data Link Frame, IP Datagram etc.

Protocols

- The Protocol will also typically specify a procedure for interpreting and responding to the Control data within the PDU.
- For instance, if a service provides for reliable transfer of data then there must be some means of tracking and recovering from data loss.

Protocols

- In this instance part of the PDU Control field will include numbering:
  - e.g. byte numbers, frame numbers etc.
- The Protocol will also specify how data in the Control Fields are to be interpreted and responded to if necessary.
  - e.g. for missing frame return a REJ message.

The TCP Transport Service Offering

- The TCP Transport Service has the following characteristics:
  - ***Connection Orientation***: Before two applications entities can communicate they must first establish a connection.
  - ***Point-To-Point Communication***: Each TCP connection has exactly <u>two</u> 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 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 TCP transport service is offered to a *user process* that exists within the application layer:
  - This *user process* is considered a “Transport Service User”,
  - The service is typically offered through a set of **<u>primitives</u>** across the interface between the layers,
  - Calls to these primitives cause the transport service provider to perform some action.

The Transport Entity

- To aid future discussions the TCP software within the transport layer that implements the service will be referred to as the *Transport Entity*:
  - This is the “Transport Service Provider”.

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.

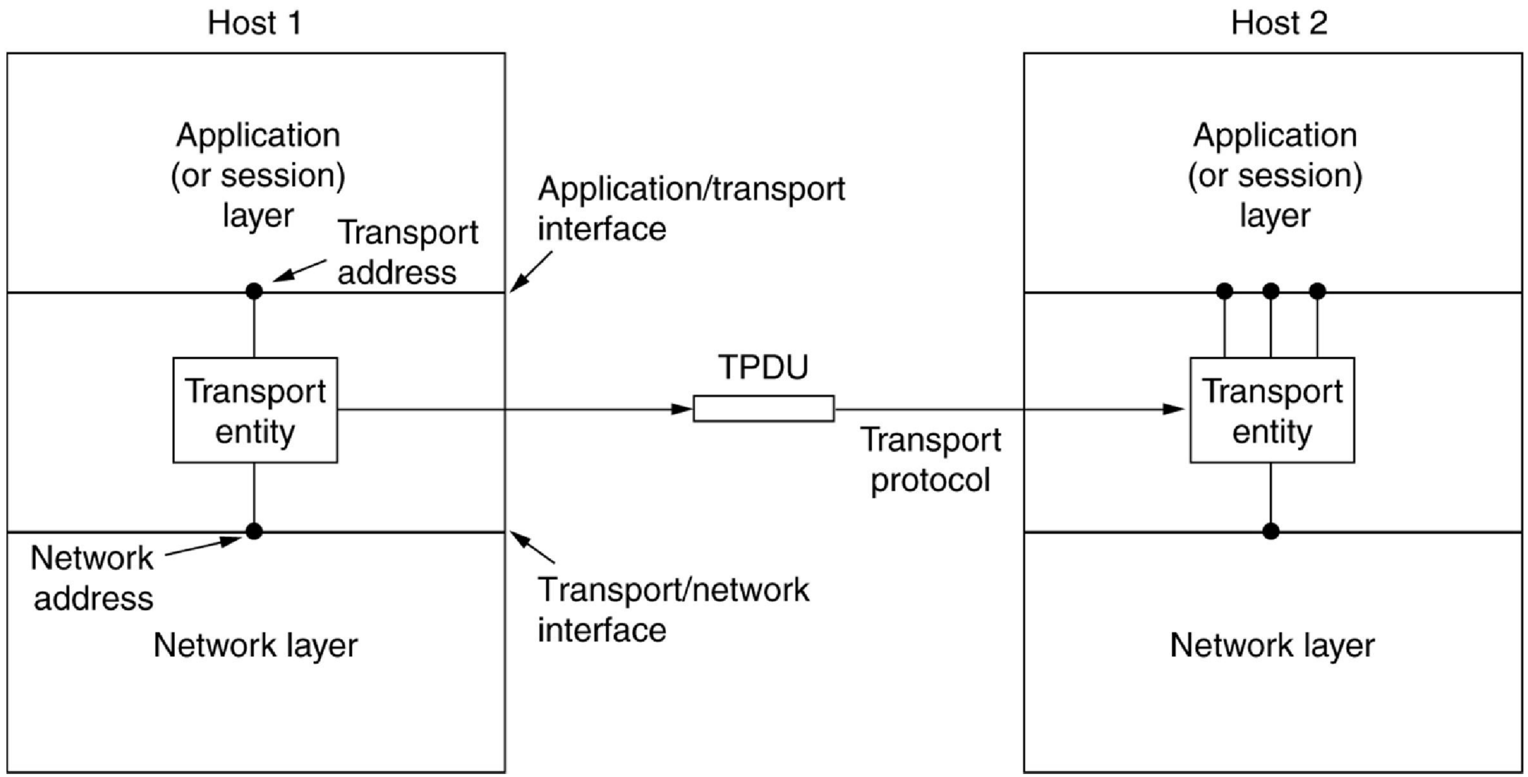
The Transport Entity

- 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 and receive the necessary PDUs.
- The next diagram shows the Transport Entity in context.
- As can be seen it shows the Transport Layer sitting between the Application and Network layers.

The Transport Entity

- It has a connection to each of the layers above and below:
  – It is the "Service Provider" to the Application Layer and,
  – A "Service User" of the Network Layer.
- The TPDU represents the framing structure that is exchanged between *Peer Entities*:
  – The PDU does NOT move horizontally between the Transport layers,
  – Instead it moves up-and-down through the Protocol Stack.

The Network, Transport and Application Layers

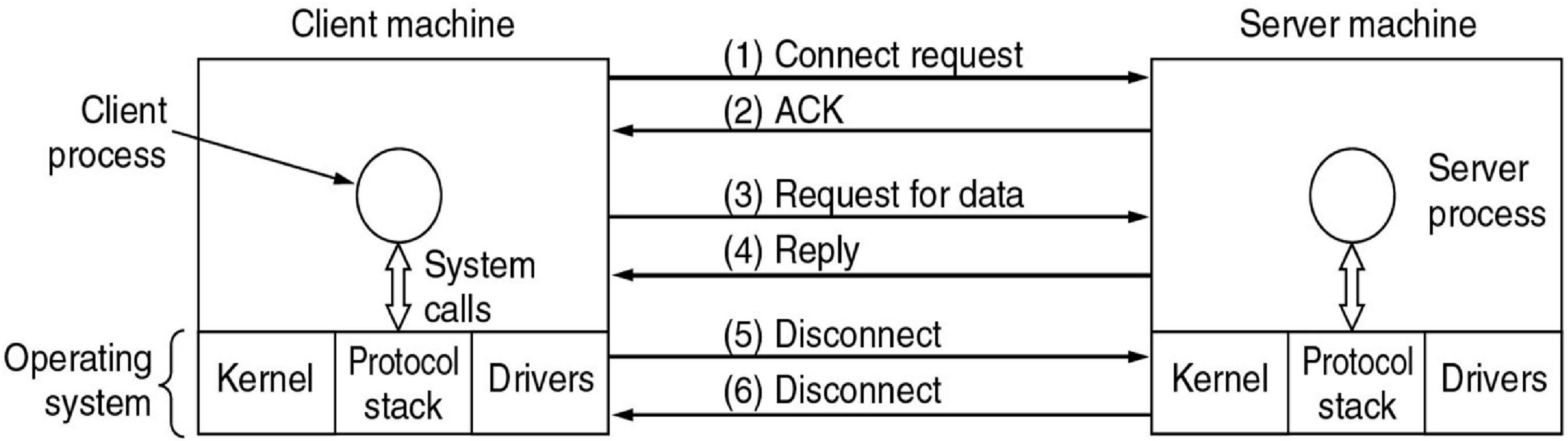


Transport Service Primitives

- What do these primitives look like?
- Consider a ***generic*** transport interface as shown in the next diagram:
  - Here can be seen a list of Primitives.
  - These primitives allow application programs to establish, use and release connections.
- Typically there is a very precise sequence of calls to these primitives:
  - The exact sequence differs for *clients* and *servers*.

A set of basic Transport Service primitives

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



Managing connections

- Before explaining the sequence of primitive calls it is important to understand the *Phases of Communication*.
- Recall from previous discussions on HDLC that there are generally three phases of communication associated with a connection-oriented services:
  - Connection Establishment,
  - Data Transfer, and,
  - Connection Release.
- During each phase a variety of PDUs are exchanged between the *client* and *server*:
  - Each PDU contains a message for the “other side”.

Connection interactions per phase

- *Connection Establishment Phase*:
  - The *server* firstly executes a LISTEN primitive:
    - The server's *transport entity* responds to this primitive call by blocking the server until a client request arrives,
  - The client's then executes a CONNECT call:
    - This causes a CONNECTION REQUEST TPDU to be sent to the Server,
  - The server transport entity unblocks the server and returns a CONNECTION ACCEPTED TPDU to the Client,
  - The client transport entity then unblocks the client and the connection is deemed **established**.

Connection interactions per phase

- *Data Transfer Phase:*
  - With an <u>active</u> connection data can now be exchanged between the client and server using the SEND and RECEIVE primitives,
  - Each side must take turns using (blocking) RECEIVE and SEND.
- *Release Phase:*
  - Either the client or the server can call a DISCONNECT primitive:
    - This sends a DISCONNECT TPDU to the remote transport entity
    - Upon arrival, the connection is deemed <u>**released.**</u>

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

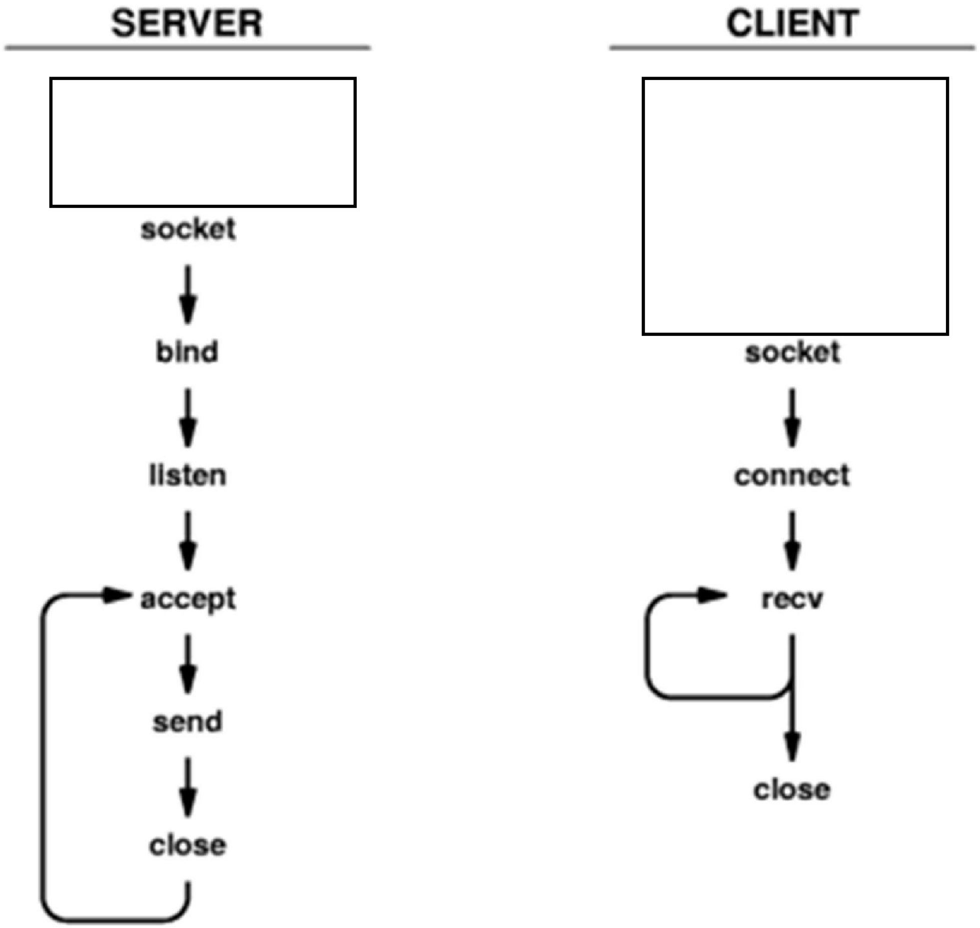
Socket Communication And UNIX

- To use the TCP/IP protocols to communicate an application must request the OS to create a ***socket***:
  - This is an abstract concept which will be explained in detail later.
- The OS returns a descriptor that *identifies* the socket:
  - For now just consider it as a *file handle*.
- As with File I/O this descriptor must be used in all future interactions with the socket
- The following slides lists the Socket API primitives and illustrates an example of how these primitives are used by a client and server application

The Socket Transport Primitives

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
WRITE/SEND	Send some data over the connection
READ/RECEIVE	Receive some data from the connection
CLOSE	Release the connection

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* within the Transport Entity:
    - Table space is allocated within the transport entity,
    - A file descriptor is returned 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 a queuing space within the transport entity for 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* <u>and</u> *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

Sockets and Socket Libraries

- 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:*
  - These 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 as opposed to the OS.
- Both implementations provide the same semantics from a programmer's perspective
- Applications using either implementation can be **ported** to other computer systems.

Example use of Sockets

- The next slide shows an example Client application using the Sockets API.
- It is a simple Daytime Client which connects to a Daytime Server for the current date and time on the Server Host.
- This application is written in ‘C’.
- Lines 24, 41, 44 and 57 show four of the socket primitives being called.
- This programme will be explained in class and you will have a chance to compile and run it against a pre-compiled server.

An Example Client Using *Sockets*

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "Practical.h"

int main(int argc, char *argv[]) {
    char recvbuffer[BUFSIZE]; // I/O buffer
    int numBytes = 0;

    if (argc < 3) // Test for correct number of arguments
    DieWithUserMessage("Parameter(s)",
        "<Server Address> <Server Port>");

    char *servIP = argv[1];     // First arg: server IP address (dotted quad)

    in_port_t servPort = atoi(argv[2]);

    // Create a reliable, stream socket using TCP
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (sock < 0)
        DieWithSystemMessage("socket() failed");

    // Construct the server address structure
    struct sockaddr_in servAddr;             // Server address
    memset(&servAddr, 0, sizeof(servAddr)); // Zero out structure
    servAddr.sin_family = AF_INET;           // IPv4 address family
    // Convert address
    int rtnVal = inet_pton(AF_INET, servIP, &servAddr.sin_addr.s_addr);
```

An Example Client Using *Sockets*

```
  if (rtnVal == 0)
      DieWithUserMessage("inet_pton() failed", "invalid address string");
  else if (rtnVal < 0)
      DieWithSystemMessage("inet_pton() failed");
  servAddr.sin_port = htons(servPort);    // Server port

// Establish the connection to the echo server
  if (connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0)
      DieWithSystemMessage("connect() failed");

  while ((numBytes = recv(sock, recvbuffer, BUFSIZE - 1, 0)) > 0) {
      recvbuffer[numBytes] = '\0';    // Terminate the string!
      fputs(recvbuffer, stdout);      // Print the echo buffer
          /* Receive up to the buffer size (minus 1 to leave space for
             a null terminator) bytes from the sender */
      }
  if (numBytes < 0)
    DieWithSystemMessage("recv() failed");
      //    else if (numBytes == 0)
      //    DieWithUserMessage("recv()", "connection closed prematurely");

  fputc('\n', stdout); // Print a final linefeed

  close(sock);
  exit(0);
}
```