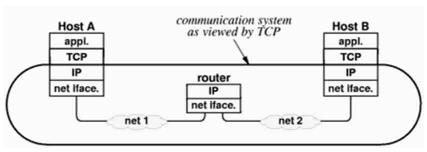
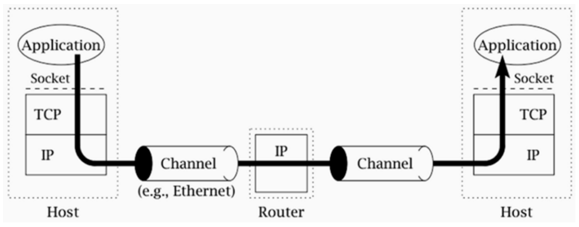
06. The Transport Layer

The Transport Layer

The TCP 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’s view of the Network

TCP provides a reliable, cost-effective and-to-end data transport service independently of the physical networks.

- IMPORTANT: a service is very different to a protocol

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 are how the services are implemented. A protocol specifies a framing structure which will include a number of filed containing control data.

This framing structure is known as **Protocol Data Unit** (**PDU**). For example: Data Link Frame, IP Datagram.

The **protocol** will also typically specify a procedure for interpreting and responding to the control data within the PDU.

If a service provides for reliable transfer of data then there must be some means specified within the protocol for tracking and recovering from data loss. In the case, part of the PDU control field will include numbering (byte numbers, frame numbers, ...).

The protocol will also specify how data in the control fields are to be interpreted and responded to if necessary, for example, 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 establish a connection.
- **Point-to-Point Communication**: each TCP connection has exactly two endpoints.
- **Complete Reliability**: TCP guarantees that the data will be delivered exactly as sent, 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.
- **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 **primitives** across the interface between the layers.
- Calls to these primitives cause the transport service provider to perform some action.

The Transport Entity

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.

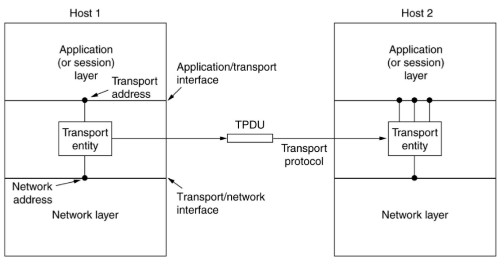
The transport entity can be located in any number of places including:

- 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 in 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.

It has a connection to each of the layers above and below:

- It is the service provider to the Application Layer.
- It is a service user of the Network Layer

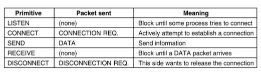


The Network, Transport and Application Layers

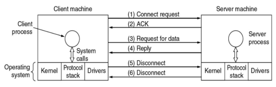
The TDPU represents the framing structure that is exchanged between peer entities:

- The PDU does not move horizontally between the Transport layers, it moves up-and-down through the Protocol Stack.

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



Generic Transport Interface

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.

Managing Connections

There are generally three phases of communication associated with a connection-oriented service:

- Phase 1: Connection Establishment
- Phase 2: Data Transfer
- Phase 3: Connection Release

During each phase, a variety of PUDs are exchanged between the client and server. Each PUD contains a message destined for the peer entity on the remote end of the channel.

Connection interactions per phase

- Connection Establishment Phase:
  1. The server application executes a LISTEN primitive (**call to Listen**).
  2. The server's transport entity responds to this primitive call by **blocking** the server until a client request arrives.
  3. The client application then executes a CONNECT primitive (**call to Connect**).
  4. The client's transport entity responds to this call by **blocking** the client and sending a CONNECTION REQUEST TPDU to the server.

5. Upon receipt of the CONNECTION REQUEST TPDU the server's transport entity unblock the server and returns a CONNECTION ACCEPTED TPDU to the client.
6. The client's transport entity the unblocks the client.
7. The connection is now deemed established.

- Data Transfer Phase:
    - With an active connection now established data can 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 application can call a DISCONNECT primitive. This causes a DISCONNECT TDPU to be sent to the remote transport entity.
    - Once the DISCONNECT TDPU has been received and acknowledged the connection is deemed **release**.

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 original TCP/IP suite of internetworking protocols.

The socket API has become the de facto standard for interfacing to TCP/IP.

Origins of the Socket concept

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 File I/O UNIX uses an open-read-write-close paradigm. An application makes the following calls in strict order:

- **open** to prepare a file for reading/writing.

- **read** or **write** to retrieve or send data to/from the file.
- **close** to release the file.

When **open** is first called, a descriptor is returned and all calls to the file use this descriptor.

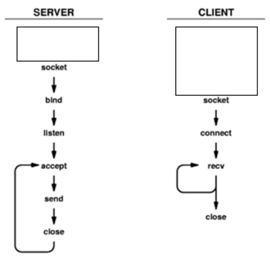
Socket communication also uses this **descriptor** approach.

Socket Communication in UNIX

Applications that need to use TCP/IP protocols to communicate must request the OS to create a **socket**. The OS returns a **descriptor** that uniquely identifies the socket and this descriptor must be used in all interactions with the socket.

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 attempts 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



Example Client-Server Interaction Using Sockets

The Socket Primitives

- **SOCKET**: this primitive creates a new end point within the Transport Entity. Table space is allocated within the transport entity and a file descriptor, which is used in all future calls, is returned.

- **BIND**: this primitive binds a socket to a network address. This allows remote clients to connect to it.

- **LISTEN**: this primitive allocates 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.

Primitives executed by clients

SOCKET and CONNECT (this primitive blocks the client and actively starts the connection).

Primitives executed by clients and servers

SEND and RECV: this 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 and applications using either implementation can be ported to other computer systems.