Transport Layer

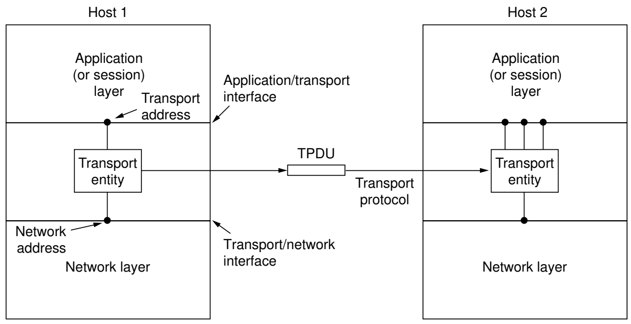
Essence: The transport layer is responsible for completing the services of the underlying network to the extent that application development can take place:

- provide reliable connection-oriented services
- provide unreliable connectionless services
- provide parameters for specifying quality of services

Important: we're talking about efficient and cost-effective services, in particular reliable connections.

Note: depending on the services offered by the network layer, the added functionality in the transport layer can vary considerable.

Transport Layer cnt'd



Note: The issue here is that the network layer is in the hands of carriers: organizations that offer a (generally wide-area) computer network to their clients. Clients have no say in what the carrier actually offers.

Consequence: If we want to develop applications that are independent of the particular services offered by a carrier, we'll have to devise a standard communication interface and implement that interface at the client's sites. The transport layer contains such implementations.

Transport QoS

- **Connect establishment delay:** how long does it take to establish a connection. The shorter, the better.
- **Connection establishment failure probability:** Chance of not being able to establish a connection within the specified maximum delay. Should be as high as possible.
- **Throughput:** how many bytes per second can a client pass through a connection.
- **Transit delay:** how long does it take for a message to make it from sender to receiver. The shorter, the better.
- **Residual error ratio:** fraction of lost or garbled messages. Should be as near to zero as possible.
- **Protection:** to what extent is a client's message protected against eavesdropping or tampering.
- **Priority:** is it possible for a client to indicate that one connection should be serviced at a higher priority than another one.
- **Resilience:** probability that the transport layer looses a connection (i.e. that the connection breaks on account of network failures).

Transport Layer Interface

Example: Consider the Berkeley **socket interface**, which has been adopted by most UNIX systems, as well as Windows 95/NT:

SOCKET	Create a new communication endpoint
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept N connections
ACCEPT	Block until someone remote wants to establish a connection
CONNECT	Attempt to establish a connection
SEND	Send data over a connection
RECEIVE	Receive data over a connection
CLOSE	Release the connection

Socket communication

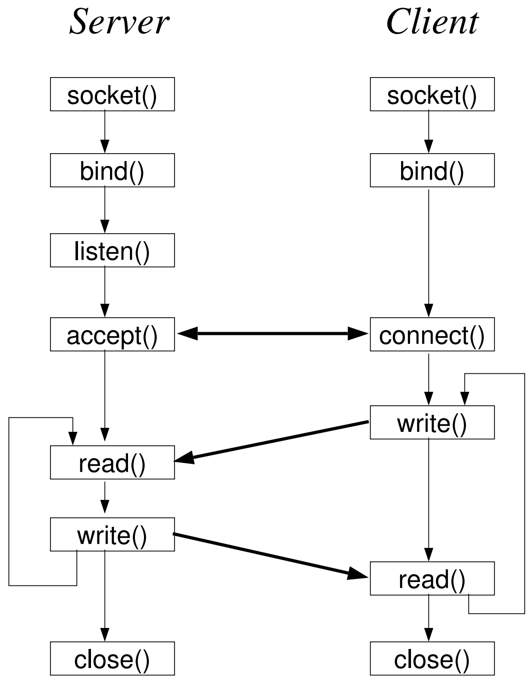
Basic principle:

- The client and server each **bind** a socket address and a name to the locally created socket.
- The server must **listen** to its socket, thereby telling the kernel that it will subsequently wait for connections from clients.
- After that, the server can **accept** or **select** connections from clients.
- The client **connects** to the socket. It needs to provide the socket address by which it can locate the server.

After a connection has been accepted (or selected), the client and server communicate through **read/write** operations on their respective sockets.

Communication ends when a connection is **closed**.

Connection-oriented socket communication



Question: what about connectionless communication?

Sockets – server side

```
serverAddress : SocketAddress –– Publicly known address
...
process Server is
   clientSocket : Socket; –– Private socket
   ...
begin
   serverSocket := new Socket;
   serverSocket.bind(serverAddress);
   serverSocket.listen(maxConnections);
   loop
      serverSocket.accept(clientSocket);
      clientSocket.recv(request);
      clientSocket.send(answer);
      clientSocket.close();
   end loop ;
end Server;
```

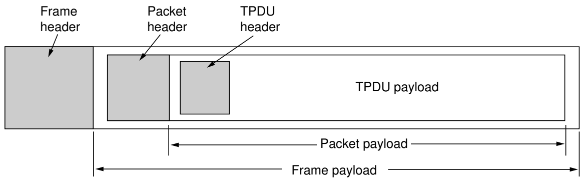

Sockets – client side

```
serverAddress : SocketAddress –– Publicly known address
...
process Client is
   clientAddress : SocketAddress; –– Private address
   clientSocket : Socket; –– Private socket
   ...
begin
   clientAddress := new SocketAddress;
   clientSocket := new Socket;

   clientSocket.bind(clientAddress);
   loop
      if clientSocket.connect(serverAddress)
         then exit ;
         else sleep(1);
      end if ;
   end loop ;

   clientSocket.recv(request);
   clientSocket.send(answer);
   clientSocket.close();
end Client;
```

Question: What am I doing in the loop?

Some Observations

Note 1: Messages sent by clients are encapsulated as **transport protocol data units** (TPDUs) to the network layer:



Note 2: A real hard part is establishing and releasing connections. The model can be either symmetric or asymmetric:

Symmetric: one side sends a disconnect request, and waits for the other to acknowledge that the connection is closed. Yes, there are some problems with this model. In fact, it turns out it is impossible to implement.

Asymmetric: one side just closes the connection, and that's it. Yes, it's simple, but you may lose some data this way. Not really acceptable.