Transport Layer

- The transport layer protocols provide for logical communication between application processes (not hosts).
  - Provides reliable connection-oriented services.
  - Provides unreliable connectionless services.
  - Provides parameters for specifying quality of services.
- Depending on the underlying network layer, the added functionality can be big or small.

Transport Service Interface

Example: Consider the Berkeley socket interface, which has been adopted by all UNIX systems, as well as Windows:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication endpoint</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 N connections</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block until someone remote wants to establish a connection</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send data over a connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Receive data over a connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>

- The client and server each bind a transport-level 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 transport-level address by which it can locate the server.
- Now the client and server communicate through send/receive operations on their respective sockets.
  - Also standard read/write operations can be used.

Notes
Connection-Oriented Socket Communication

**Question**

- What about connectionless communication? There is no connection ⇒ no need for listen, accept, and connect.

**Notes**

The ServerSocket constructor also implicitly binds the socket to the port.

- Sockets are read / write using the standard operations to access files.
- To handle multiple clients at the same time, a new thread must be created per each new socket.

**Sockets**

**Server**

```java
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(2008);
        System.out.println("Listening on port 2008...\n");
        while (true) {
            Socket socket = serverSocket.accept();
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);
            System.out.println(in.readLine());
            out.println(System.console().readLine());
            socket.close();
        }
    }
}
```

**Client**

```java
public class Client {
    public static void main(String[] args) throws UnknownHostException, IOException {
        Socket socket = new Socket("127.0.0.1", 2008);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);
        out.println(System.console().readLine());
        System.out.println(in.readLine());
    }
}
```

**Notes**

- The Socket constructor also implicitly connects the socket to the IP address and port specified.
- Sockets are read / write using the standard operations to access files.

**Observations**

- Messages sent by clients are encapsulated as transport protocol data units (TPDUs) to the network layer.
- 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.
  - 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.
Transport Protocol

- Transport protocols strongly resemble those in the data link layer
  - e.g., lots of error and flow control

- Big differences when it comes to solutions!
  - explicit addressing
  - establishing, maintaining, and releasing connections
  - the many connections require different solutions
    - e.g., packets can follow different paths and arrive with different delays
  - handle effects of subnet storage capabilities

Addressing

- Each layer has its own way of dealing with addresses.
- In Internet, a transport service access point is an IP address with a port number.

Service Locations

Fixed Addresses

- Solution: have a single process, located at a well-known address, handle a large number of services
  - `inetd` in the UNIX world

Unknown Addresses

- Problem: Sometimes you just can't have a process handle all services, e.g., because the service requires special hardware (file server)
  - ⇒ find address of the server
- Solution: you'll have to use a name server.

Question

- Great, so how do we find the name server? Fixed address :)

Notes

- Notes
- Notes
- Notes
- Notes
- Notes
- Notes
- Notes
- Notes
- Notes
Connection Establishment

- To establish a connection, you send off a connection request to the other end.
- The other end then accepts the connection, and returns an acknowledgment.
- **Problem**: Suppose you don’t get an answer, so you do another request.
  - (a) Your first request didn’t make it → no harm done
  - (b) The ack didn’t make it back → you’re establishing a second connection, but this can probably be detected.
  - (c) Your first request didn’t make it yet → now you’re really making a second connection and no one knows you didn’t do this on purpose.

**Question**

- Why can (c) happen?
  - The network has storage capabilities, and unpredictable delays. This means that things can pop up out of the blue.

Attacking Duplicates

- **Solution**: Restrict the lifetime of TPDUs
  - if the maximum lifetime is known in advance, we can be sure that a previous packet is discarded and that it won’t interfere with successive ones.
- Three possible solutions:
  - (a) Restricted subnet design
  - (b) Putting a hop counter in each packet
  - (c) Timestamping each packet (requires global clock synchronization)
- **Basic idea**: With packet lifetimes bounded, assign sequence numbers to TPDUs, and let the sequence number space be so large that no two outstanding TPDUs can have the same number.
- **Problem #2**: When a host crashes, it has to start numbering TPDUs again. So, where does it start?
  - You can’t just wait the maximum packet lifetime $T$ and start counting from the start again: in wide-area systems, $T$ may be too large to do that.
  - The point is that you must avoid that an initial sequence number corresponds to a TPDU still floating around.
  - So, just find the right initial number.

Forbidden Region

- **Solution**: Assign sequence numbers in accordance to clock ticks, and assume that the clock continues ticking during a crash
  - when a connection is set up, the low-order $k$ bits of the clock are used as the initial sequence number (also $k$ bits)
  - the clocks at different hosts need not be synchronized
  - This leads to a forbidden region:
    - every time you want to assign a next sequence number, check whether that number is in the forbidden region.
- **Attention**:
  - when sequence numbers are assigned at a lower pace than the clock ticks, we may enter the region “from the top.”
  - likewise, assigning them too fast makes you enter the region “from the bottom.”

Error-Free Connection Establishment

- **Problem**: Great, we have a way of avoiding duplicates, but how do we get a connection in the first place?
  - One way or the other we have to get the sender and receiver to agree on initial sequence numbers.
  - We need to avoid that an old (unnumbered) connection request pops up.
Error-free Connection Establishment

Three-way handshake

Solution: Three-way handshake.
1. Host 1 chooses a sequence number, x, and sends a CONNECTION REQUEST TPDU containing it to host 2.
2. Host 2 replies with an ACK TPDU acknowledging x and announcing its own initial sequence number, y.
3. Finally, host 1 acknowledges host 2's choice of an initial sequence number in the first data TPDU that it sends.

(a) Normal Operation
(b) Old duplicate CONNECTION REQUEST appearing out of nowhere
(c) Duplicate CONNECTION REQUEST and duplicate ACK

Error-free Connection Release

There are two styles of terminating a connection: asymmetric release and symmetric release.
- asymmetric release is the way the telephone system works: when one party hangs up, the connection is broken.
- symmetric release treats the connection as two separate unidirectional connections and requires each one to be released separately.

asymmetric release may result in loss of data:

Symmetric Connection Release

Big problem: Can we devise a solution to release a connection such that the two parties will always agree?
⇒ the answer is simple: NO.

There is a famous problem that illustrates this issue. It is called the two-army problem:
- if either blue army attacks by itself, it will be defeated, but if the two blue armies attack simultaneously, they will be victorious.
- their only communication medium is to send messengers into the valley, where they might be captured and the message lost.
- It can be proven that no protocol to synchronize the attack exists.
- neither of the two commanders can be certain that the other one has received the message.

Symmetric Connection Release

To see the relevance of the two-army problem to releasing connections, just substitute "disconnect" for "attack."
- if neither side is prepared to disconnect until it is convinced that the other side is prepared to disconnect too, the disconnection will never happen.

Examples:
- Normal case: Host 1 sends disconnect request (DR). Host 2 responds with a DR. Host 1 acknowledges, and ACK arrives at host 2.
- ACK is lost: What should host 2 do? It doesn’t know for sure that its DR came through.
- Host 2’s DR is lost: What should host 1 do? Of course, send another DR, but this brings us back to the normal case. This still means that the ACK sent by host 1 may still get lost.

Pragmatic solution: Use timeout mechanisms
- this will catch most cases, but it is never a fool-proof solution
- the initial DR and all retransmissions may still be lost, resulting in a half-open connection.
Flow Control and Buffering

**Problem:** Hosts may have so many connections that it becomes infeasible to allocate a fixed number of buffers per connection to implement a proper sliding window protocol.

⇒ we need a dynamic buffer allocation scheme.

- With an unreliable network, i.e. an unreliable datagram service provided by the network layer, the sender will have to buffer TPDS until they are acknowledged.
  - the receiver may decide to drop incoming TPDS if it has no buffer space available.

- With a reliable network, the sender will still have to buffer a TPDU until it is acknowledged.
  - the network only acknowledges successful receipt, not delivery.

**Solution:** the sender and receiver need to negotiate the number of TPDS that can be transmitted in sequence.

because buffer space no longer comes for free.

Buffer Reservation

- The sender requests a number of buffers at the receiver's side when opening a connection.
- The receiver responds with a credit grant.
- After that, the receiver grants more credit when buffer space becomes available.

**Question**

- What can we do about the potential deadlock? Just use yet another timer mechanism.

Flow Control

- As memory prices continue to fall dramatically, it may become feasible to equip hosts with so much memory that lack of buffers is rare, if ever, a problem.
- **Problem:** another bottleneck will appear: the carrying capacity of the subnet.
  - it may not be enough for what the sender and receiver want to do.
  - if the network can handle $c$ TPDS per second, and takes a total of $r$ seconds to transmit, propagate, queue and process the TPDU, and to send an ACK, the sender need only maintain $c \cdot r$ buffers.
  - more buffers is overkill for the network.

- **Solution:** Let the sender estimate $c$ and $r$ and adjust its buffers.
  - the capacity can be determined by simply counting the number of TPDS acknowledged during some time period and then dividing by the time period.
  - the time required for a transmitted TPDU to be acknowledged can be measured exactly and a running mean maintained.
Multiplexing

(a) Upward Multiplexing
- Assume that the network offers only a limited number of virtual circuits, or that a user doesn’t want to pay so much
  ⇒ use a single circuit for several connections
(b) Downward Multiplexing
- If a user requires a lot of bandwidth that cannot be supported by a single network virtual circuit
  ⇒ use several circuits for a single connection

Crash Recovery

Problem
- Problem: A host responds to the receipt of a TPDU by performing an operation and returning an acknowledgment
  how should the sending host respond when the receiving host crashes before, during, or after its response?
- Situation:
  - assume the sender is informed that the receiver has just recovered from a crash.
  - should the sender retransmit the TPDU it just sent, or not?
  - Distinguish between:
    (a) S0: sender had no outstanding (unacknowledged) TPDUs
    (b) S1: sender had one outstanding TPDO
  - The receiving host may:
    (a) first send ACK and then deliver the data to the application
    (b) first deliver the data to the application and then send ACK

Crash Recovery

Solution (?)
- The sending host can employ four different strategies
  (a) Always retransmit
  (b) Never retransmit
  (c) Retransmit only in S0
  (d) Retransmit only in S1
- Unfortunately none of this strategies works in all cases!
- Consider the following events
  - A: send ACK
  - C: host crashes
  - W: data written to the application

Example Protocol

Service Primitives
- LISTEN: Block connection request comes in
- CONNECT: Attempt to establish a connection
- SEND: Send data over a connection
- RECEIVE: Receive data over a connection
- DISCONNECT: Release the connection

Network Layer Primitives
- CALL REQUEST: Sent to establish a connection
- CALL ACCEPTED: Response to CALL REQUEST
- CLEAR REQUEST: Sent to release a connection
- CLEAR CONFIRM: Response to CLEAR REQUEST
- DATA: Used to transport data
- CREDIT: For managing the window

States of a connection
- IDLE: Not yet established
- WAITING: CONNECT called; CALL_REQ. sent
- QUEUED: CALL REQ. arrived; LISTEN not called
- ESTABLISHED: Connection established
- SENDING: Waiting for permission to send TPDU
- RECEIVING: RECEIVED has just been called
- DISCONNECTING: DISCONNECT has just been called
It is useful to represent the state of a protocol as a finite state machine.

Alternative, a graph-based representation is used.

The User Datagram Protocol (UDP) is essentially just a transport-level version of IP.

UDP is simple:
- no flow control
- no error control
- no retransmissions
- UDP packets cannot be larger than 65 K (that is the maximum IP packet size)

So why not use IP instead? Because we still need the port fields to deliver the packet to the correct application.

Note: Connection-less protocol
- no need for listen / accept
- each datagram packet must carry the full address / port of the recipient.
**User Datagram Protocol (UDP)**

When shall we use it?

- A typical question is why people should ever use UDP (instead of TCP) since it does not provide any reliability nor flow control.
- Here are some reasons:
  1. Finer application-level control over what data is sent and when (e.g., real-time, skype).
  2. No connection establishment.
  3. No connection state.
  4. Small packet header overhead.

Besides real-time apps, one area where UDP is really useful is in client-server situations:

- Often, the client sends a short request to the server and expects a short reply back.
- If either the request or reply is lost, the client can just time out and try again.

  ⇒ Simpler code and fewer messages required.
  
  - E.g., DNS query: symbolic address (e.g., www.cs.vu.nl) → IP address translation (e.g., 130.37.20.20).

**Remote Procedure Call (RPC)**

- UDP is also used by the Remote Procedure Call (RPC):
  - A client-server communication in which a procedure is made available to remote clients.
  - The call (including its parameters) is shipped to the server.

1. Client calls the procedure at a local stub.
2. Client stub marshals request: It puts everything into a (UDP) message.
3. The message is transferred over the network.
4. The server stub unmarshals the message...
5. ... and calls the local implementation of the procedure.

- RMI (Remote Method Invocation) is the Java version of RPC.

**Real Time Protocol (RTP)**

- Problem: Can we support multimedia streaming over the Internet?
- Solution: The Real-Time Transport Protocol provides some best-effort support.

- RTP essentially just multiplexes a number of multimedia streams into a single UDP stream.
- The receiver is responsible for compensating missing packets:
  - Highly application dependent (e.g., through interpolation).
- RTP packets can be timestamped:
  - Packets belonging to the same substream can receive a timestamp indicating how far off they are with respect to their predecessor.
  - This approach allows the system to reduce jitter.
  - In addition, timestamps can be used to synchronize multiple substreams (e.g., video with two audio channels).
Transmission Control Protocol (TCP)

- Connection-oriented service that supports byte streams (not message streams)
  - A sender may send eight 512-byte packets that are received as two chunks of 1024 bytes, and one of 2048 bytes.
- Transport address consists of a 16-bit port number, which augments the underlying IP address.
- TCP ensures reliable, point-to-point connections
  - No support for multicasting or broadcasting.
- A TCP TPDU is called a segment, consisting of (minimal) 20-byte header, and maximum total length of 65,535 bytes.
  - A segment is fragmented by the network layer when it is larger than the network's maximum transfer unit (MTU).

TCP Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port</td>
<td></td>
</tr>
<tr>
<td>destination port</td>
<td></td>
</tr>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgment number</td>
<td></td>
</tr>
<tr>
<td>header length</td>
<td></td>
</tr>
<tr>
<td>unused</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>window size</td>
<td></td>
</tr>
<tr>
<td>checksum</td>
<td></td>
</tr>
<tr>
<td>urgent data pointer</td>
<td></td>
</tr>
<tr>
<td>options field</td>
<td></td>
</tr>
</tbody>
</table>

- Sequence and acknowledgment numbers refer to bytes not to messages.
- URG indicates immediate processing and transmission: the receiver is signalled.
- Acknowledgments are piggybacked when ACK = 1.
- acks specify the next byte expected, not the last one received.
- PSH used to solicit the receiver to pass the data to the application immediately.
- RST is used to reset a connection (also by ISPs to stop P2P traffic).
- SYN is for connection setup (ACK = 0: request; ACK = 1: accepted).
- FIN is for connection release. Data sent before the release is not lost.
- Window size can be at most 64 Kbytes (16 bits).
  - a Window Scale option can be specified to have a window size of 2^30.

Sequence Numbers

- Sequence numbers are associated with bytes in the data stream.
  - not with segments, as we have used them before.
- The sequence number in a TCP segment indicates the sequence number of the first byte carried by that segment.

```
application data stream

MSS=1024b:
1... 1024 1025... 2048 2049... 3072 3073... 4096

sequence number a TCP segment
```

Acknowledgment Numbers

- An acknowledgment number represents the first sequence number not yet seen by the receiver.
- TCP acknowledgments are "cumulative."
TCP Connection Management

- Connection establishment:
  - three-way handshake protocol
  - initial sequence numbers are chosen through a clock based scheme (clock tick is 4 µs)

- Connection release:
  - to be thought of as independent releases of two simplex connections
  - to release a connection, either party can send a TCP segment with the FIN bit set
  - when the FIN is acknowledged, that direction is shut down
  - when both directions are shut down, the connection is released
  - to avoid the two-army problem, timers are used

TCP Connection Management Modeling

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>No connection active or pending</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Server waiting for conn. request</td>
</tr>
<tr>
<td>SYN RCVD</td>
<td>Conn. request has arrived; wait for ACK</td>
</tr>
<tr>
<td>SYN SENT</td>
<td>Conn. request just sent; wait for SYN+ACK</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>Data can be sent and received</td>
</tr>
<tr>
<td>FIN WAIT 1</td>
<td>Client just sent conn. release</td>
</tr>
<tr>
<td>FIN WAIT 2</td>
<td>Server just agreed to release connection</td>
</tr>
<tr>
<td>TIMED WAIT</td>
<td>Wait for all packets to die</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Client &amp; server both tried to close</td>
</tr>
<tr>
<td>CLOSE WAIT</td>
<td>Other side initiated release</td>
</tr>
<tr>
<td>LAST ACK</td>
<td>Wait for all packets to die</td>
</tr>
</tbody>
</table>

TCP Window Management

- The receiver sends an acknowledgment for the next byte that can be sent in the current stream, and the maximum number of bytes that may be sent.

- a window size of 0 is legal
  - it means that no more messages can be sent...
  - ...apart from a 1-byte segment to ping the receivers (no deadlocks)
TCP Window Management

- **Optimizations**
  - The TCP entity is not obliged to immediately transmit data that the application hands over.
  - It can do as much buffering as it likes.
  - Same goes for acknowledgments: wait in the hope to have some data and piggyback them.

- **Nagle’s algorithm**
  - **Problem**: consider interactive, character-oriented applications: sending 1 character per message would cost 40 byte of overhead (TCP + IP header) per byte of data.
  - **Solution**: Send the first character immediately and then buffer as much characters as possible until the previous batch is acknowledged.
  - *Not* suitable for remote desktop (mouse movements need be smooth).

- **Silly window syndrome**
  - **Problem**: the receiver reads one byte at a time and acknowledges one at a time.
  - **Solution**: The receiver should wait until it has freed enough buffer and can receive a reasonable amount of bytes in a row.

TCP Congestion Control

- **Problem**: As before, the transport layer has to take into account that the underlying network can be the bottleneck.
- **How do we detect and react to congestion?**
  - **Solution**: Use a congestion window next to the window granted by the receiver.
  - The actual window size is the minimum of the two.

- **Jacobson's Slow start**
  1. Initialize congestion window to the maximum segment size.
  2. Send it off.
  3. If it gets acknowledged, double the size (slow start).
  4. Repeat until failure.
  5. In addition, use a threshold.
     - On a timeout, lower the threshold to 50% of the congestion window.
     - Do a slow start (exponential) until new threshold.
     - Add maximum segment size to congestion window size after that (linear growth).

TCP Timer Management (1/2)

- **Problem**: How do we determine the best retransmission timeout?
  - **Situation** is really different from what happens at data link layer (a).
  - At transport layer (b) we observe a large standard deviation.

- **Solution**: Use a highly dynamic algorithm that constantly adjusts the timeout interval, based on continuous measurements (Jacobson again...)
  - For each connection, TCP maintains a variable, $RTT$, that is the best current estimate of the round-trip time.
  - If the acknowledgment gets back before the timer expires, TCP measures how long the acknowledgment took, say, $M$.
  - It then updates $RTT$ according to the formula:
    $$RTT = \alpha RTT + (1 - \alpha)M$$
    Typically $\alpha = 7/8$. 

Notes

- Notes
  - Notes
    - Notes
      - Notes
TCP Timer Management (2/2)

- Even given a good value of RTT, choosing a suitable retransmission timeout, i.e., \( \beta \cdot RTT \), is nontrivial.
- Jacobson proposed making \( \beta \) roughly proportional to the standard deviation of the acknowledgment arrival time probability density.
- He suggested using the mean deviation \( D \) as a cheap estimator of the standard deviation.
- Whenever an ack comes in, a new value of \( D \) is created:
  \[
  D = \alpha D + (1 - \alpha) |RTT - M|
  \]
- If a packet got wrong (ack didn't arrive), don't update TTL (Karn's algorithm).
- Most TCP implementations now use this algorithm and set the timeout interval to:
  \[
  \text{Timeout} = RTT + 4 \cdot D
  \]
- Other timers are maintained by TCP:
  - Persistence timer: prevents deadlock when window size is 0.
  - Keepalive timer: check the connection if it has been idle for long.
  - Connection release timer: used when closing a connection.

Bandwidth vs. Latency

- Both bandwidth and latency influence the maximum throughput.
- Consider, for example, sending a 64-KB burst of data from San Diego to Boston in order to fill the window size.
  - Only 500 µs after the last TPDUs are out on the fiber.
  - Suppose that the link is 1 Gbps and the one-way speed-of-light-in-fiber delay is 20 ms.
  - After 20 ms, the lead TPDUs hits Boston and is acknowledged.
  - Finally, 40 ms after starting, the first acknowledgment gets back to the sender and the second burst can be transmitted.
- The transmission line was used for 0.5 ms out of 40 → the efficiency is about 1.25 percent.
- The product bandwidth \( \times \) latency represents the capacity of the network.
  - In the example, 1 Gbps \( \times \) 40 ms \( \rightarrow \) 40 millions of bits.
  - We just sent 524,288 bits (i.e., 64 Kbytes)!
- Most often, the latency is the real bottleneck!

\[
\text{Throughput} \leq \frac{0.75 \text{W}}{\text{RTT}} \quad \text{(if no losses),} \quad \text{throughput} \leq \frac{1.22 \text{MSS}}{\text{RTT} \sqrt{P_{\text{loss}}}} \quad \text{(with prob. } P_{\text{loss}}\text{)}
\]

Wireless TCP

- Problem: TCP assumes that IP is running across wires.
- When packets are lost, TCP assumes this is caused by congestion and slows down.
- In wireless environments, packets get lost due to reliability issues.
- In those cases, TCP should do the opposite: try harder!!
- Solution #1: Split TCP connections to distinguished wired/wireless IP.
- Solution #2: Let the base station do at least some retransmissions, but without informing the source.
- Effectively, the base station makes an attempt to improve the reliability of IP as perceived by TCP.