Transport Layer

Introduction
Flow Control (Credit Allocation)
Connection Management
Examples: TCP, UDP

Introduction
Transport layer protocols are end-to-end protocols
Transport layer is only implemented at the hosts

Why the Transport Layer?
- The IP layer provides a mechanism for transmitting datagrams from point A to point B.
- What are the missing pieces?
  - ?
  - ?
  - ?
- What are the functions of the transport layer?

Functions of the Transport Layer
- Addressing
- Flow Control
- Error Control
- Connection Management

Note: The mechanisms needed to implement a transport service are largely dependent on the existing network layer service

Addressing
- An address at the transport layer is typically a tuple (Station, Port) where
  - Station is the network address of the host, and
  - Port identifies the application
- The <IP address, port number> tuples used in Unix sockets are in fact transport layer addresses
- Problem with Addressing: How to find the address of a service?

Homework 6.5
- What is the well known port number for the following services:
  - Web servers
  - POP Mail Servers v3 (POP=Post Office Protocol)
  - Quake
  - Napster
  - ICQ
  - Mobile IP (Home Agent)
  - FTP
  - Telnet
  - DNS (Domain Name Service)
  - SSH (Secure Shell – SecureCRT)
Flow Control

Why do we need flow control at the transport layer?
- User of receiving transport entity cannot keep up with the data flow.
- Receiving transport entity itself cannot keep up with flow of incoming packet.

Result: Buffer overflows in the receiving transport entity.

Need for Flow Control

Flow Control at the Transport Layer

Flow Control at the transport layer is more complex than flow control at the data link layer:
- Delays are variable and longer
- Flow control involves the transport users, the transport entities, and the network service
- What’s the problem with fixed-size sliding window flow control?

Credit Allocation Flow Control

Credit Allocation Flow Control is an extension of the sliding window flow control.

Main Idea:
- Enhance the sliding window protocol by a mechanism that decouples acknowledgments from flow control.

Then:
- Packets can be acknowledged without granting permission for new transmissions
- Used in many existing transport protocols, including TCP

Credit Allocation Flow Control

Initialization during connection setup:
- Set initial window size of receiver
- Receiver both acknowledges TPDU's and grants credit by sending a message:
  (ACK N, CREDIT M)
  - ACK N:Acknowledges all sequence numbers through N-1
  - CREDIT M:Sets the number of credits to M

Credit is the maximum window size (buffer space at the receiver)

Example

Initial Setting: Credit = 7
### Error Control at the Transport Layer

- **Basic techniques** for error recovery are the same as at the Data Link Layer:
  - Lost or damaged TPDUs are recovered with one of the ARQ retransmission schemes
  - Mostly: Go-back-N, Selective Repeat.

- **Problem:** The end-to-end delay of TPDUs is variable. This makes it difficult to set the timeout values.
  - Small timeout value: unnecessary retransmissions.
  - Large timeout value: low throughput.
- **Most transport protocols** have adaptive timers.

### Connection Management

- A connection is an abstraction of a communication channel between a sender and a receiver.
- A reliable connection guarantees that:
  - All transmitted packets are received correctly
  - All transmitted packets are received in the order they were sent
  - Order is enforced via sequence numbers
  - A connection may be bi-directional (e.g., sockets)
- **Issues in connection management**
  - Connection establishment
  - Connection termination

### Connection Establishment

- Connection establishment is asymmetric:
  - One side puts itself in a LISTEN state (server)
  - One side issues a request for connection or RFC (client)

### Simple Solution (which has problems)

- **CR (SeqNo = x)**: Connection Request, A wants to start with SeqNo = x
- **ACK (SeqNo = y)**: Acknowledge request, B will want to start with SeqNo = y
- **DATA (SeqNo = x)**: Data transmission with SeqNo x

### Problems with Two-Way Handshake

- B responds to CR(SeqNo = y), an old duplicate connection requests from A
- In the shown scenario, A believes that the ACK is for the connection request CR(SeqNo = y)
- Result: A starts to send data with Sequence x. B will throw the data away since it expects SeqNo = z
Three-Way Handshake

- Note: A and B acknowledge the sequence number from the other side
- This solution provides protection from old duplicate connection requests

Scenario 1

- Duplicate connection request (CR) appears
- Host A rejects the invalid connection request in the REJ(ACK=y) packet
- Note: The connection request CR(SeqNo=x) is completed successfully

Scenario 2

- A duplicate acknowledgement (ACK) appears
- Host A rejects the invalid ACK by sending REJ(ACK=y)
- Note: The connection request CR(SeqNo=x) is completed successfully

Connection Termination

- A connection release should involve both sides of the connection (otherwise data is lost)
- Here: B should wait after Disconnection Request (DR) is sent until all data has arrived

Connection Termination in 4 steps

- An elegant way to terminate connections is to have each end shut down independently (“half-close”)
- If one end wants to shut down, it sends a DR message
- Four steps involved:
  1. A sends a DR to B (active close)
  2. B ACKs the DR, (at this time: B can still send data to A)
  3. and B sends a DR to A (passive close)
  4. A ACKs the DR

Connection Termination in 4 steps

- To account for packet losses, a timer is needed to limit the waiting time of a side
Transport Protocols in the Internet

The Internet uses two transport protocols:

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

UDP - User Datagram Protocol

- datagram oriented
- unreliable, connectionless
- simple
- unicast and multicast
- useful only for few applications, e.g., multimedia applications
- used a lot for services:
  - network management (SNMP), routing (RIP), naming (DNS), etc.

TCP - Transmission Control Protocol

- stream oriented
- reliable, connection-oriented
- complex
- only unicast
- used for most Internet applications:
  - web (http), email (smtp), file transfer (ftp), terminal (telnet), etc.

UDP Format

- Port numbers identify sending and receiving applications (processes).
  - Maximum port number is $2^{16} - 1 = 65,535$
- Message Length is at least 8 bytes (i.e., Data field can be empty) and at most 65,535
- Checksum is for header (of UDP and some of the IP header fields)

Port Numbers

- UDP (and TCP) use port numbers to identify applications
- A globally unique address at the transport layer (for both UDP and TCP) is a tuple <IP address, port number>
- There are 65,535 UDP ports per host
TCP

TCP = Transmission Control Protocol
- Connection-oriented protocol
- Provides a reliable unicast end-to-end byte stream over an unreliable internetwork.

TCP is Connection-Oriented
- Before any data transfer, TCP establishes a connection:
  - One TCP entity is waiting for a connection ("server")
  - The other TCP entity ("client") contacts the server
- The actual procedure for setting up connections is the three-way handshake
- Each connection is full duplex

Reliable Data Transfer
- Byte stream is broken up into chunks which are called segments
  - Receiver sends acknowledgements (ACKs) for segments
  - TCP maintains a timer. If an ACK is not received in time, the segment is retransmitted
- Detecting errors:
  - TCP has checksums for header and data. Segments with invalid checksums are discarded
  - Each byte that is transmitted has a sequence number

TCP gives a Byte Stream Service
- To the lower layers, TCP handles data in blocks, the segments.
- To the higher layers TCP handles data as a sequence of bytes and does not identify boundaries between bytes
- So: Higher layers do not know about the beginning and end of segments!

TCP Segment Format
- TCP segments have a 20 byte header with \( \geq 0 \) bytes of data.

TCP header fields
- Port Number:
  - A port number identifies the endpoint of a connection.
  - A pair \(<\text{IP address}, \text{port number}>\) identifies one endpoint of a connection.
  - Two pairs \(<\text{client IP address}, \text{server port number}>\) and \(<\text{server IP address}, \text{server port number}>\) identify a TCP connection.
**TCP header fields**

- **Sequence Number (SeqNo):**
  - Sequence number is 32 bits long.
  - So the range of SeqNo is $0 \leq \text{SeqNo} < 2^{32} - 1 \approx 4.3 \text{ Gbyte}$
  - Each sequence number identifies a byte in the byte stream
  - Initial Sequence Number (ISN) of a connection is set during connection establishment

- **Acknowledgement Number (AckNo):**
  - Acknowledgements are piggybacked
  - A host uses the AckNo field to send acknowledgements. (If a host sends an AckNo in a segment it sets the “ACK flag”)
  - The AckNo contains the next SeqNo that a host wants to receive
  - Example: The acknowledgement for a segment with sequence numbers 0-1500 is AckNo=1501

- **Flag bits:**
  - **URG:** Urgent pointer is valid
    - If the bit is set, the following bytes contain an urgent message in the sequence number range “SeqNo <= urgent message <= SeqNo+urgent pointer”
  - **ACK:** Segment carries a valid acknowledgement
  - **PSH:** PUSH Flag
    - Notification from sender to the receiver that the receiver should pass all data that it has to the application.
    - Normally set by sender when the sender’s buffer is empty
  - **RST:** Reset the connection
    - The flag causes the receiver to reset the connection
    - Receiver of a RST terminates the connection and indicates higher layer application about the reset
  - **SYN:** Synchronize sequence numbers
    - Sent in the first packet when initiating a connection
  - **FIN:** Sender is finished with sending
    - Used for closing a connection
    - Both sides of a connection must send a FIN

**TCP Connection Establishment**

- TCP uses a three-way handshake
  1. **ACTIVE OPEN:** Client sends a segment with
    - SYN bit set
    - port number of client
    - initial sequence number (ISN) of client
  2. **PASSIVE OPEN:** Server responds with a segment
    - SYN bit set
    - initial sequence number of server
    - ACK for ISN of client
  3. **Client acknowledges** by sending a segment with
    - ACK ISN of server

**Three-Way Handshake**
TCP Connection Termination

- Each end of the data flow must be shut down independently ("half-close")
- If one end is done it sends a FIN segment. This means that no more data will be sent

- Four steps involved in closing a connection
  (1) X sends a FIN to Y (active close)
  (2) Y ACKs the FIN,  
    (at this time: Y can still send data to X)
  (3) and Y sends a FIN to X (passive close)
  (4) X ACKs the FIN.

TCP States in Connection Lifetime

- SYN (SeqNo = x)
- SYN (SeqNo = y, AckNo = x + 1)
- SYN_SENT (active open)
- SYN_RCVD (passive open)
- ESTABLISHED (normal data transfer state)
- FIN_WAIT_1 (active close)
- FIN_WAIT_2 (passive close)
- TIME_WAIT (passive close)
- CLOSE_WAIT (active close)
- LAST_ACK (active close)
- CLOSED (closed state)

TCP States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>No connection is active or pending</td>
</tr>
<tr>
<td>LISTEN</td>
<td>The server is waiting for an incoming call</td>
</tr>
<tr>
<td>SYN_RCVD</td>
<td>A connection request has arrived; wait for Ack</td>
</tr>
<tr>
<td>SYN_SENT</td>
<td>The client has started to open a connection</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>Normal data transfer state</td>
</tr>
<tr>
<td>FIN_WAIT_1</td>
<td>Client has said it is finished</td>
</tr>
<tr>
<td>FIN_WAIT_2</td>
<td>Server has agreed to release</td>
</tr>
<tr>
<td>TIME_WAIT</td>
<td>Wait for pending packets (2MSL wait state)</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Both Sides have tried to close simultaneously</td>
</tr>
<tr>
<td>LAST_ACK</td>
<td>Wait for pending packets</td>
</tr>
</tbody>
</table>

TCP State Transition Diagrams

Opening A Connection

- CLOSED
- LISTEN
- SYN_RCVD
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_1
- FIN_WAIT_2
- TIME_WAIT
- CLOSING
- LAST_ACK
- CLOSED

Closing A Connection

- CLOSED
- LISTEN
- SYN_RCVD
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_1
- FIN_WAIT_2
- TIME_WAIT
- CLOSING
- LAST_ACK
- CLOSED

Flow Control in TCP

- **Flow Control**: How to prevent that the sender overruns the receiver with information?

  - **Flow Control in TCP**
    - TCP implements sliding window flow control
    - Sending acknowledgements is separated from setting the window size at sender. Acknowledgements do not automatically increase the window size (credit allocation flow control)
    - Acknowledgements are cumulative
    - Window changes depending on application behavior
Flow Control

- Sender does not have to send all data immediately
- Receiver does not have to deliver immediately
- What if
  - Sender application generates one character at a time?
  - Receiver application consumes one character at a time?

Nagle’s Algorithm

- Deals with slow senders
  - Send first byte
  - Hold back until ACK is received, then send what’s in the buffer
  - Send if accumulated MSS worth of data or if accumulated half the window size worth of data

The Silly Window Syndrome

- If receiver dequeues its buffer one character at a time it gives “silly” window information to the sender
- Solution: Clark’s Algorithm
  - Do not send window updates until you can handle one MSS or half your buffer is empty.

Slow Start/Congestion Avoidance

- TCP has a mechanism for congestion control. The mechanism is implemented at the sender
- The sender has two additional parameters:
  - Congestion Window (cwnd: Initial value is 1 MSS counted as bytes)
  - Threshold Value (ssthresh: Initial value is 65536 bytes)
- The window size at the sender is set as follows:
  - Allowed Window = MIN (flow control window, congestion window)

Slow Start

- Whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced:
  - Set cwnd = MSS bytes (=1 segment)
  - Each time an ACK is received, the congestion window is increased by 1 segment (= MSS bytes).
  - If an ACK acknowledges two segments, cwnd is still increased by only 1 segment.
  - Even if ACK acknowledges a segment that is smaller than MSS bytes long, cwnd is increased by MSS bytes.

- Does Slow Start increment slowly? Not really.
  - In fact, the increase of cwnd can be exponential

Slow Start Example

- The congestion window size grows very rapidly
  - For every ACK, we increase cwnd by 1 irrespective of the number of segments ACKed
  - TCP slows down the increase of cwnd when cwnd > ssthresh
Slow Start

“Slow Start” slows down if the congestion window is larger than a threshold value.

If \( \text{cwnd} > \text{ssthresh} \) then each time an ACK is received, increment \( \text{cwnd} \) as follows:

\[ \text{cwnd} = \text{cwnd} + \text{MSS} \times \frac{\text{MSS}}{\text{cwnd}} \]

So \( \text{cwnd} \) is increased by one only if all segments have been acknowledged.

\( \text{ssthresh} \) is modified if there is congestion in the network.

Slow Start Example

Assume that \( \text{ssthresh} = 8 \)

### Slow Start / Congestion Avoidance

- Implemented with two variables:
  - Congestion window (\( \text{cwnd} \))
  - Slow start threshold (\( \text{ssthresh} \))

- Initialization:
  \[ \text{cwnd} = \text{MSS} \text{ bytes} \]
  \[ \text{ssthresh} = 64535 \text{ bytes} \]

- The window size at the sender is set as follows:
  Allowed Window = MIN (advertised window, \( \text{cwnd} \))

### Slow Start / Congestion Avoidance

Here we give a more accurate version than in our earlier discussion of Slow Start:

\[
\begin{align*}
\text{If } \text{cwnd} & \leq \text{ssthresh} \text{ then } \\
& \text{Each time an Ack is received:} \\
& \text{cwnd} = \text{cwnd} + \text{MSS} \\
\text{else } & \text{ /* cwnd > ssthresh */ } \\
& \text{Each time an Ack is received:} \\
& \text{cwnd} = \text{cwnd} + \text{MSS} \times \frac{\text{MSS}}{\text{cwnd}} + \text{segsize} / 8
\end{align*}
\]

### Slow Start / Congestion Avoidance

Each time when congestion occurs (timeout or receipt of duplicate ACK),

- \( \text{cwnd} \) is reset to one:
  \[ \text{cwnd} = 1 \]

- \( \text{ssthresh} \) is set to half the current size of the congestion window:
  \[ \text{ssthresh} = \text{cwnd} / 2 \]
**Slow Start / Congestion Avoidance**

- A typical plot of cwnd for a TCP connection (MSS = 1500 bytes):

**Error Control in TCP**

- TCP implements a variation of the Go-back-N retransmission scheme.
- TCP couples error control and congestion control (i.e., it assumes that errors are caused by congestion).

**Round-Trip Time Measurements**

- The retransmission mechanism of TCP is adaptive.
- The retransmission timers are set based on round-trip time (RTT) measurements that TCP performs.

**Round-Trip Time Measurements**

- Retransmission timer is set to a Retransmission Timeout (RTO) value.
- RTO is calculated based on the RTT measurements.
- The RTT measurements are smoothed by the following estimators, srtt and rttvar:

\[
\begin{align*}
srtt_{n+1} &= \alpha \cdot RTT + (1 - \alpha) \cdot srtt_n \\
rttvar_{n+1} &= \beta \cdot |RTT - srtt_{n+1}| + (1 - \beta) \cdot rttvar_n \\
RTO_{n+1} &= srtt_{n+1} + 4 \cdot rttvar_{n+1}
\end{align*}
\]

The gains are set to \( \alpha = 1/4 \) and \( \beta = 1/8 \).
- \( srtt_0 = 0 \ sec, \ rttvar_0 = 3 \ sec \).
- Also: \( RTO_0 = srtt_0 + 2 \cdot rttvar_0 \)

**Karn’s Algorithm**

- If an ACK for a retransmitted segment is received, the sender cannot tell if the ACK belongs to the original or the retransmission.

**RTO Calculation: Example**

- At \( t_1 \): \( RTO = srtt + 2 \cdot rttvar = 6 \ sec \)
- At \( t_2 \): \( RTO = 2 \cdot (srtt + 4 \cdot rttvar) = 24 \ sec \) (exponential backoff)
- At \( t_3 \): \( RTO \) is not updated (Due to Karn’s algorithm)