The Transport Layer

CS168, Fall 2014
Scott Shenker (understudy to Sylvia Ratnasamy)
http://inst.eecs.berkeley.edu/~ee122/

Material thanks to Ion Stoica, Jennifer Rexford, Nick McKeown, and many other colleagues
Preliminaries

- Sylvia will be back next week
  - You are stuck with me this week

- Please ask questions....

- I will ask a few questions during this lecture
  - Someone should answer....
  - But for the rest of you, I ask questions to give you a chance to think, not because I want an answer...
The Transport Layer
(brief review from last lecture)
From Lecture#3: Transport Layer

- Layer at end-hosts, between the application and network layer
Why a transport layer?

- Transport layer and application both on host
- Why not just combine the two?
- And what should that code do anyway?
Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
  - No guidance on how much traffic to send and when
  - Dealing with this is tedious for application developers
Role of the Transport Layer

- Communication between application processes
  - Mux and demux from/to application processes
  - Implemented using *ports*
Role of the Transport Layer

- Communication between application processes
- Provide common end-to-end services for app layer
  - Reliable, in-order data delivery
  - Well-paced data delivery
    - too fast may overwhelm the network
    - too slow is not efficient
Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
  - also SCTP, MTCP, SST, RDP, DCCP, ...
Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
  - only provides mux/demux capabilities
Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
- TCP is the whole-hog protocol
  - offers apps a reliable, in-order, bytestream abstraction
  - with congestion control
  - but no performance guarantees (delay, bw, etc.)
Transport Design Issues
Socket: software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system

- socketID = socket(…, socket.TYPE)
- socketID.sendto(message, …)
- socketID.recvfrom(…)
- will cover in detail after midterm

Two important types of sockets
- UDP socket: TYPE is SOCK_DGRAM
- TCP socket: TYPE is SOCK_STREAM
Ports

- Problem: deciding which app (socket) gets which packets
- Solution: *port* as a transport layer identifier (16 bits)
  - packet carries source/destination port numbers in transport header
- OS stores mapping between sockets and ports
  - Port: in packets
  - Socket: in OS
- For UDP ports (SOCK_DGRAM)
  - OS stores (local port, local IP address) $\leftrightarrow$ socket
- For TCP ports (SOCK_STREAM)
  - OS stores (local port, local IP, remote port, remote IP) $\leftrightarrow$ socket
Two Questions

- Why the difference?
  - For UDP ports (SOCK_DGRAM)
    - OS stores (local port, local IP address) $\leftrightarrow$ socket
  - For TCP ports (SOCK_STREAM)
    - OS stores (local port, local IP, remote port, remote IP) $\leftrightarrow$ socket

- Why do you need to include local IP?
<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13-bit Fragment Offset</td>
<td>16-bit Header Checksum</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
<td>32-bit Source IP Address</td>
<td>32-bit Destination IP Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Payload</td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>6 = TCP, 17 = UDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit Type of Service (TOS)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-bit Flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Field</td>
<td>Bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit Type of Service (TOS)</td>
<td>4 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3-bit Flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 = TCP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 = UDP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Source Port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit Destination Port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>More transport header fields ...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recap: Multiplexing and Demultiplexing

- Host receives IP packets
  - Each IP header has source and destination IP address
  - Each Transport Layer header has source and destination port number

- Host uses IP addresses and port numbers to direct the message to appropriate socket
  - UDP maps local destination port and address to socket
  - TCP maps address pair and port pair to socket
Rest of Lecture

- More on ports
- UDP
- Reliable Transport
- Next lecture: Details of TCP
More on Ports

- Separate 16-bit port address space for UDP and TCP

- "Well known" ports (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - helps client know server’s port
  - Services can listen on well-known port

- Ephemeral ports (most 1024-65535): given to clients
UDP: User Datagram Protocol

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery

- UDP described in RFC 768 – (1980!)
  - Destination IP address and port to support demultiplexing
  - Optional error checking on the packet contents
    - (checksum field = 0 means “don’t verify checksum”)

```
<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
<tr>
<td>DATA</td>
<td></td>
</tr>
</tbody>
</table>
```
Question

- Why do UDP packets carry the sender’s port?
Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
  - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
  - Packets can be corrupted, delayed, dropped, reordered, duplicated
Reliable Transport

- In a perfect world, reliable transport is easy

@Sender
  - send packets

@Receiver
  - wait for packets
Reliable Transport

- In a perfect world, reliable transport is easy
- All the bad things best-effort can do
  - a packet is corrupted (bit errors)
  - a packet is lost (*why?*
  - a packet is delayed (*why?*
  - packets are reordered (*why?*
  - a packet is duplicated (*why?*)
Reliable Transport

- Mechanisms for coping with bad events
  - Checksums: to detect corruption
  - ACKs: receiver tells sender that it received packet
  - NACK: receiver tells sender it did not receive packet
  - Sequence numbers: a way to identify packets
  - Retransmissions: sender resends packets
  - Timeouts: a way of deciding when to resend a packet
  - Forward error correction: a way to mask errors without retransmission
  - Network encoding: an efficient way to repair errors
  - ....
Dealing with Packet Corruption

1

Sender

Time

Receiver

2

ack

nack

2

1
Dealing with Packet Corruption

Data and ACK packets carry sequence numbers
Dealing with Packet Loss

Timer-driven loss detection
Set timer when packet is sent; retransmit on timeout
Dealing with Packet Loss (of ack)

Sender

Time

Receiver

1

1

P(1)

P(1)

P(2)

timeout
duplicate!
Dealing with Packet Loss

Timer-driven retransmission can lead to duplicates.
Components of a solution (so far)

- checksums (to detect bit errors)
- timers (to detect loss)
- acknowledgements (positive or negative)
- sequence numbers (to deal with duplicates)

But we haven’t put them together into a coherent design...
Designing Reliable Transport
A Solution: “Stop and Wait”

We have a correct reliable transport protocol!

Probably the world’s most inefficient one (*why?*)
Stop & Wait is Inefficient

If $\text{TRANS} \ll \text{RTT}$ then

$$\text{Throughput} \sim \frac{\text{DATA}}{\text{RTT}}$$
Orders of Magnitude

- Transmission time for 10Gbps link:
  - ~ microsecond for 1500 byte packet

- RTT:
  - 1,000 kilometers ~ O(10) milliseconds
Three Design Decisions

- Which packets can sender send?
  - Sliding window

- How does receiver ack packets?
  - Cumulative
  - Selective

- Which packets does sender resend?
  - GBN
  - Selective repeat
Sliding Window

- **window** = set of adjacent sequence numbers
  - The size of the set is the **window size**; assume window size is $n$

- General idea: send up to $n$ packets at a time
  - Sender can send packets in its window
  - Receiver can accept packets in its window
  - Window of acceptable packets “slides” on successful reception/acknowledgement
  - Window contains all packets that might still be in transit

- Sliding window often called “packets in flight”
Sliding Window

- Let $A$ be the last ack’d packet of sender without gap; then window of sender $= \{A+1, A+2, \ldots, A+n\}$

\[
\begin{array}{c}
A \\
\downarrow \\
\text{sequence number} \rightarrow
\end{array}
\]

- Let $B$ be the last received packet without gap by receiver, then window of receiver $= \{B+1, \ldots, B+n\}$

\[
\begin{array}{c}
B \\
\downarrow \\
\text{sequence number} \rightarrow
\end{array}
\]
Throughput of Sliding Window

- If window size is $n$, then throughput is roughly $\min\left[ \frac{n \text{DATA}}{\text{RTT}}, \text{Link Bandwidth} \right]$

- Compare to Stop and Wait: $\text{Data}/\text{RTT}$

- Two questions:
  - What happens when $n$ gets too large?
  - How do we choose $n$?
Acknowledgements w/ Sliding Window

- Two common options
  - cumulative ACKs: ACK carries next in-order sequence number that the receiver expects
Cumulative Acknowledgements (1)

- At receiver

- After receiving B+1, B+2

- Receiver sends ACK(B+3) = ACK(B_{new} + 1)
Cumulative Acknowledgements (2)

- At receiver

- After receiving B+4, B+5

- Receiver sends ACK(B+1)
Acknowledgements w/ Sliding Window

- Two common options
  - cumulative ACKs: ACK carries next in-order sequence number the receiver expects
  - selective ACKs: ACK individually acknowledges correctly received packets

- Selective ACKs offer more precise information but require more complicated book-keeping
Sliding Window Protocols

- Resending packets: two canonical approaches
  - Go-Back-N
  - Selective Repeat

- Many variants that differ in implementation details
Go-Back-N (GBN)

- Sender transmits up to $n$ unacknowledged packets
- Receiver only accepts packets in order
  - discards out-of-order packets (i.e., packets other than $B+1$)
- Receiver uses cumulative acknowledgements
  - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1$^{st}$ outstanding ack ($A+1$)
- If timeout, retransmit $A+1$, … , $A+n$
Sliding Window with GBN

- Let A be the last ack’d packet of sender without gap; then window of sender = \{A+1, A+2, \ldots, A+n\}

- Let B be the last received packet without gap by receiver, then window of receiver = \{B+1, \ldots, B+n\}

- Already ACK’d
- Sent but not ACK’d
- Cannot be sent
- Received and ACK’d
- Acceptable but not yet received
- Cannot be received
GBN Example w/o Errors

Window size = 3 packets

Sender Window:

- {1} 1
- {1, 2} 2
- {1, 2, 3} 3
- {2, 3, 4} 4
- {3, 4, 5} 5
- {4, 5, 6} 6
-...

Receiver Window:

-...

Sender

Time

Receiver
GBN Example with Errors

Window size = 3 packets

Sender

Timeout
Packet 4

Receiver
Selective Repeat (SR)

- Sender: transmit up to \( n \) unacknowledged packets
- Assume packet \( k \) is lost, \( k+1 \) is not
- Receiver: indicates packet \( k+1 \) correctly received
- Sender: retransmit only packet \( k \) on timeout
- Efficient in retransmissions but complex book-keeping
  - need a timer per packet
SR Example with Errors

Window size = 3 packets

{1} 1
{1, 2} 2
{1, 2, 3} 3
{2, 3, 4} 4
Timeout Packet 4
{4, 5, 6} 4
{4, 5, 6} 4
{7, 8, 9} 7

ACK=5
ACK=6
ACK=4

Sender
Receiver

Time
GBN vs Selective Repeat

- When would GBN be better?
- When would SR be better?
Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough.
- Sender has to buffer all unacknowledged packets, because they may require retransmission.
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits.
- Implementation complexity depends on protocol details (GBN vs. SR).
Recap: components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)

Reliability protocols use the above to decide when and what to retransmit or acknowledge
What does TCP do?

Most of our previous tricks + a few differences

- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgements (like GBN)
- Sender maintains a single retransmit timer
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmit (next time)
- Introduces timeout estimation algorithms (next time)
Next Time

- TCP
  - Reliability
  - Congestion control