The Transport Layer: End-to-end Functions

Lecture 24
6.02 Fall 2007
December 7, 2007

Hari Balakrishnan

Layering in the Internet

HTTP, FTP, SMTP, ...

Application

TCP, UDP

Today

Transport

End-to-End Layer
Everything else!
Reliability, integrity,
ordering, jitter ctrl,
congestion response, ...

Forwarding & routing
(and addressing)
Framing, coding, [limited]
rxtms, channel access
Modulation/demodulation

Ethernet, WiFi, ...

Network

Data Link

Physical

End-to-End Layer
Everything else!
Reliability, integrity,
ordering, jitter ctrl,
congestion response, ...

Forwarding & routing
(and addressing)
Framing, coding, [limited]
rxtms, channel access
Modulation/demodulation
Packets in a Best-Effort Network Lead a Rough Life

- Can be lost for any number of reasons
  - No current route
  - Current route not working properly
  - Queue overflow at switch (due to congestion)
  - Packet corruption (noise/interference), collisions
  - Uncorrectable errors detected by CRC
  - Queue overflow at receiving node
- Can arrive out-of-order
- And experience variable delays

Transport Layer

- Transport layer “cleans up” best-effort behavior to present clean abstraction
- Common functions (not all needed for apps, so multiple transport layer choices)
  - Reliability:
    - At least once: each packet received at least once
    - At most once
    - Exactly once
  - Ordering
  - Data integrity (detect corruption)
  - Timeliness (“remove” variable latency)
  - Flow and congestion control: for performance
Fundamental Mechanisms

- All loss recovery mechanisms employ *redundancy*
- **Retransmissions**
  - Re-send after timeout
- **Forward Error Correction (FEC)**
  - Code data to overcome some errors or loss

“At Least Once” Transport Protocol: Take 1
Stop-and-Wait

- Each packet has a *sequence number* set by sender
  - A “nonce”: always fresh, never reused
  - Simple approx.: incrementing 32 or 64-bit number
- Receiver: upon receipts of packet $k$, send acknowledgment (ack) for $k$ (“I got $k$”)
- Sender: Upon ack $k$, send $k+1$. If no ack within *timeout*, then retransmit $k$ (persistently until ack)
- Why is this protocol *at least* once?

RTT = round trip time

\[ \text{Host A} \quad \text{Host B} \]

\[ \text{Data 1} \quad \text{ACK} \quad \text{Data 2} \]

\[ \text{Host A} \quad \text{Host B} \]

\[ \text{Data 1} \quad \text{X} \quad \text{Data 1} \]

\[ \text{Timeout and retransmit} \]
Setting a Timeout

- What should timeout duration depend on?
  - RTT: Round-Trip Time
- Fixed timeouts don’t work well
  - Too big ==> delay too long
  - Too small ==> unnecessary retransmission, extra load (problem when losses caused by congestion!)
- RTT depends on traffic load and congestion

- Solution: Sender estimates RTT over duration of connection
  - Set retransmit timeout (RTO) adapting to RTT variations

RTT Measurements
(collected by Caida)

lancelet.caida.org to anala

Graph showing RTT measurements with two lines for RTT and median-filtered RTT, and a red area for packet loss.
Calculating RTT and RTO: Take 1

- Use an Exponentially Weighted Moving Average (EWMA) to estimate RTT
  - Simple and powerful estimator

  Procedure CALC_RTT(sample)
  \[ srtt \leftarrow a \times \text{sample} + (1-a) \times srtt; \]
  \[ (a = 1/8 \text{ in Internet's TCP}) \]

- Procedure CALC_RTO(srtt)
  \[ RTO \leftarrow b \times srtt \text{ (e.g., b=2)} \]
  \[ (\text{Set timeout to multiple of srtt estimate}) \]

- Not quite good enough...

Round-Trip Time (RTT) Could Be Highly Variable

Example from a TCP connection over a wide-area wireless link
Mean RTT = 2.4 seconds; Std deviation = 1.5 seconds!
Calculating RTT and RTO: Take 2
(How it’s done in TCP)

Use EWMA to maintain running estimates of both the mean (srtt) and the (linear) deviation (rttdev)
Set timeout to function of srtt and rttdev
Err on the side of conservatism (avoid spurious retransmissions as much as possible)

Procedure CALC_RTT(sample)
\[
srtt \leftarrow a \times \text{sample} + (1-a) \times \text{srtt}; \quad /* a = 1/8 */
\]
\[
dev \leftarrow \text{sample} - \text{srtt};
\]
\[
\text{if (devsample < 0): devsample} \leftarrow -\text{devsample};
\]
\[
rttdev \leftarrow b \times \text{dev} + (1-b) \times \text{rttdev}; \quad /* b = 1/4 */
\]

Procedure CALC_RTO(srtt, rttdev)
\[
\text{RTO} \leftarrow \text{srtt} + 4 \times \text{rttdev}
\]

Improving Performance

- Stop-and-wait protocol too slow: send, wait for ack, send, wait for ack...
- Throughput is just one packet per RTT
- 1500 byte pkt, 100 ms RTT pegs throughput at 15 KB/s
- Solution: Use a window
  - Keep multiple packets outstanding in the network at once; overlap xmits with acks
  - Familiar idea (pipelining)
At Least Once Protocol: Take 2
Fixed Window

- Receiver tells the sender a window size
- Sender sends in window, retransmits as needed
- Receiver acks each packet as before
- Window advances when all previous packets in window are acked
  - E.g., 5-7 sent after 2-4 ack’d
  - (Several variants possible)
- Problem w/ fixed window protocol?

At Least Once Protocol: Take 3
Fixed-Size Sliding Window

- Senders advances the window by 1 for each in-sequence ack it receives
  - I.e., window slides
  - So, idle period reduces
- But what is the correct value for the window?
  - We’ll come back to this, but first, an example…
Sliding Window in Action

window = 2-6

1 2 3 4 5 6

Sndr

Rcvr

p1 p3

Sliding Window in Action

window = 2-6

1 2 3 4 5 6

Sndr

Rcvr

p1 p3

p4 p5 p6

p2

Timeout
This protocol is similar to TCP, but TCP uses *cumulative acks*, where an ACK is the highest in-sequence byte (packet) received so far.

### Setting the Window Size

- If we can get “Idle” to 0, will achieve goal.
- Suppose window size is $W$ and round-trip is RTT.
- What is the throughput?
  - $W/RTT$
  - This number can at most be $C$, the max rate between sender and receiver.
- So, if $W = C \times RTT$, path will be fully utilized.
  - The “bandwidth-delay product”
  - This calculation is a key result for transport protocols (note well!)
But How Do We Know $C$?

- In general, we don’t: it isn’t fixed, it depends on
  the path, and it changes with time

- It depends both on the receiver host (and
  application) and on network conditions (other
  traffic, congestion, …)

- The transport layer incorporates \textit{flow control} and
  \textit{congestion control} to determine a suitable $C$ (which
  will vary with time)
  - Flow control: size of window at receiver to avoid
    swamping its buffer
  - Congestion control: how to adjust window (or data rate) to
    prevent persistent congestion
  - Many approaches, sophisticated methods and analysis

At Most Once Protocol

- Main goal is to \textit{suppress duplicates}
  - Receiver must keep track of what has been
    received and sent to app so far

- To simplify state management, pick
  incrementing packet sequence numbers

- Receiver: keeps track of which sequences
  have been delivered to app

- With this protocol, achieving in-order
  delivery is easy: receiver’s transport layer
  holds on to packets that arrive out of order

- Note: “exactly once” = “at least once” +
  “at most once”
Summary

- End-to-end transport layer hide ugly best-effort properties from applications
  - Or, applications can implement their own versions of reliability, ordering, etc.

- Reliable data transport
  - End-to-end retransmissions using seq # and acks
  - Adaptive timers and timeout estimation
  - Windows to improve performance
  - Sliding windows to improve further: bandwidth-delay product achieves high utilization

- Next lecture: network scalability + wrap-up