End-to-End Transport Over Wireless II: Snoop and Explicit Loss Notification

COS 463: Wireless Networks
Lecture 3
Kyle Jamieson

[Various parts adapted from S. Das, B. Karp, N. Vaidya]
Today

1. Transmission Control Protocol (TCP)
   - Window-based flow control
   - Retransmissions and congestion control

2. TCP over Wireless
   - TCP Snoop
   - Explicit Loss Notification
Window-Based Flow Control: Motivation

- Suppose sender sends one packet, awaits ACK, repeats…
- **Result:** At most one packet sent, per RTT
- *e.g.,* 70 ms RTT, 1500-byte packets → **Max t’put: 171 Kbps**
Idea: Pipeline Transmissions (Fixed Window-Based Flow Control)

But: RTT idle time from grant of new window to data arrival at receiver

Even better approach, used by TCP: sliding window, extends as each ACK returns, so no idle time!
Choosing Window Size: The Bandwidth-Delay Product

• **Network bottleneck**: point of *slowest rate* along path between sender and receiver

• What size sender window keeps the pipe full?

• **Window too small**: can’t fill pipe
• **Window too large**: unnecessary network load/queuing/loss
Increasing utilization with pipelining

Data packet size $L$ bits, bottleneck rate $R$ bits/second

First bit sent, $t = 0$

Last bit sent, $t = L / R$

ACK arrives, send next packet, $t = RTT + L / R$

Last bit of 2nd packet, send ACK
The bandwidth-delay product

Data packet size $L$ bits, bottleneck rate $R$ bits/second

- **Keep sending** for time $RTT = (N-1)L / R$

\[
\underbrace{(N-1)L} \quad = \quad RTT \cdot R
\]

Number of bits “in flight”  
Delay $\times$ Bandwidth product

**Goal: window size = $RTT \times$ bottleneck rate**

- e.g., to achieve bottleneck rate of 1 Mbps, across a 70 ms RTT, need window size:

\[
W = (10^6 \text{ bps} \times .07 \text{ s}) = 70 \text{ Kbits} = 8.75 \text{ Kbytes}
\]
TCP Packet Header

- TCP header: 20 bytes long
- Checksum covers TCP packet + “pseudo header”
  - IP header source and destination addresses, protocol
  - Length of TCP segment (TCP header + data)
TCP Header Details

- Connections inherently bidirectional; all TCP headers carry both data & ACK sequence numbers

- 32-bit sequence numbers are in units of bytes

- Source and destination port numbers
  - Multiplexing of TCP by applications
  - UNIX: local ports below 1024 reserved (only root may use)

- Window field: advertisement of number of bytes advertiser willing to accept
TCP: Data Transmission

• Each byte numbered sequentially (modulo $2^{32}$)

• **Sender buffers data** in case retransmission required
• **Receiver buffers data** for in-order reassembly

• **Sequence number (seqno) field** in TCP header indicates first user payload byte in packet
TCP: Receiver functionality

• Receiver indicates *offered window size* $W$ explicitly to sender in *window* field in TCP header
  – Corresponds to *available buffer space* at receiver

• Receiver sends *cumulative ACKs*:
  – ACK number in TCP header names *highest contiguous byte number received* thus far, +1
  – one ACK per received packet, or:
    • *Delayed ACK*: receiver batches ACKs, sends one for every pair of data packets (200 ms max delay)
TCP: Sender’s Window

- **Usable window** at sender:
  - Left edge advances as packets sent
  - Right edge advances as receive window updates arrive

---

**Offered window (advertised by receiver)**

- **Sent and Acknowledged**
- **Sent and Not Acknowledged**
- **Being Sent (Usable Window)**
- **Cannot Send Until Window Moves**

---

Figure 15-9: The TCP sender-side sliding window structure keeps track of which sequence numbers have already been acknowledged, which are in flight, and which are yet to be sent. The size of the offered window is controlled by the **Window Size** field sent by the receiver in each ACK.
Today

1. Transmission Control Protocol (TCP)
   - Window-based flow control
   - Retransmissions and congestion control

2. TCP over Wireless
   - TCP Snoop
   - Explicit Loss Notification
**TCP: Retransmit Timeouts**

- **Recall:** Sender sets timer for each sent packet
  - Expected time for ACK to return: RTT
  - when ACK returns, timer canceled
  - if timer expires before ACK returns, packet resent

- TCP estimates RTT using measurements $m_i$ from timed packet/ACK pairs
  - $RTT_i = ((1 - \alpha) \times RTT_i - 1 + \alpha \times m_i)$

- **Original TCP retransmit timeout:** $RTO_i = \beta \times RTT_i$
  - original TCP: $\beta = 2$
Mean and Variance: Jacobson’s RTT Estimator

- Above link load of 30% at router, $\beta \times RTT_i$ will retransmit too early!
  - Response to increasing load: waste bandwidth on duplicate packets; result: congestion collapse!

- Idea [Jacobson 88]: Estimate mean deviation $v_i$, (EWMA of $|m_i - RTT_i|$), a stand-in for variance:
  \[ v_i = v_{i-1} \times (1-\gamma) + \gamma \times |m_i - RTT_i| \]
  - Then use retransmission timeout $RTO_i = RTT_i + 4v_i$

Mean and Variance RTT estimator used by all modern TCPs
Self-Clocking Transmission

• Self-clocking transmission: Conservation of Packets
  – each ACK returns, one data packet sent
  – spacing of returning ACKs: matches spacing of packets in time at **slowest link on path**
Goals in Congestion Control

1. Achieve **high utilization** on links; don’t waste capacity!

2. Divide bottleneck link capacity **fairly among users**

3. Be **stable**: converge to steady allocation among users

4. Avoid **congestion collapse**
Congestion Collapse

- Cliff behavior observed in [Jacobson 88]
Congestion Requires Slowing Senders

• Bigger buffers cannot prevent congestion: **senders must slow down**
• Absence of ACKs **implicitly indicates congestion**
• TCP sender’s window size determines sending rate

• **Recall:** Correct window size is *bottleneck link bandwidth-delay product*

• How can the sender learn this value?
  – **Search** for it, by adapting window size
  – **Feedback** from network: ACKs return (window OK) or do not return (window too big)
Reaching Equilibrium: Slow Start

• At connection start, sender sets congestion window size, $cwnd$, to $pktSize$ (one packet’s worth of bytes), not whole window

• Sender sends up to $\min(cwnd, W)$
  – Upon return of each ACK, increase $cwnd$ by $pktSize$ bytes until $W$ reached
  – “Slow” means exponential window increase!

• Takes $\log_2\left(\frac{W}{pktSize}\right)$ RTTs to reach receiver’s advertised window size $W$
Avoiding Congestion: Multiplicative Decrease

- Recall sender uses window of size \( \min(cwnd, W) \), where \( W \) is receiver’s advertised window.

- Upon \textit{timeout} for sent packet, sender \textit{presumes packet lost to congestion}, and:
  1. sets \( ssthresh = cwnd / 2 \)
  2. sets \( cwnd = \text{pktSize} \)
  3. uses slow start to grow \( cwnd \) up to \( ssthresh \)

- End result: \( cwnd = cwnd / 2 \), via slow start.
Taking Your Fair Share: Additive Increase

- **Drops indicate** sending more than fair share of bottleneck
- **No feedback** to indicate using less than fair share

- **Solution**: *Speculatively increase window size* as ACKs return
  - **Additive increase**: For each returning ACK,
    \[ \text{cwnd} = \text{cwnd} + \left( \text{pktSize} \times \text{pktSize} \right) / \text{cwnd} \]
    - Increases cwnd by \( \approx \text{pktSize} \) bytes per RTT

**Combined algorithm: Additive Increase, Multiplicativative Decrease (AIMD)**
Refinement: Fast Retransmit (I)

• Sender **must wait well over RTT** for timer to expire before loss detected

• TCP’s minimum retransmit timeout: 1 second

• **Another indicator of loss:**
  – Suppose sender sends: 1, 2, 3, 4, 5 (...but 2 is lost)

    – Receiver receives: 1, 3, 4, 5

    – Receiver sends cumulative ACKs: 2, 2, 2, 2
      • Loss causes **duplicate ACKs**
Fast Retransmit (II)

- Upon arrival of three duplicate ACKs, sender:
  1. **sets** $cwnd = cwnd / 2$
  2. retransmits “missing” packet
  3. no slow start

- Not only loss causes dup ACKs
  - Packet reordering, too
AIMD in Action
Modeling Throughput, Loss, and RTT

• How do packet loss rate and RTT affect throughput TCP achieves?

• Assume:
  1. Only fast retransmits
  2. No timeouts (so no slow starts in steady-state)
Evolution of Window Over Time

- Average window size: $\frac{3}{4}W$
- One window of packets is sent per RTT
- Bandwidth:
  - $\frac{3}{4}W$ packets per RTT
  - $(\frac{3}{4}W \times \text{packet size}) / \text{RTT}$ bytes per second
  - $W$ depends on loss rate…
Window Size Versus Loss

• Assume no delayed ACKs, fixed RTT

• $cwnd$ grows by one packet per RTT
  – So it takes $\frac{W}{2}$ RTTs to go from window size $\frac{W}{2}$ to window size $W$; this period is one cycle

• How many packets sent in total, in a cycle?
  – $(\frac{3}{4}W / \text{RTT}) \times (W/2 \times \text{RTT}) = 3W^2/8$

• One loss per cycle (as window reaches $W$)
  – So, the packet loss rate $p = 8/3W^2$
  – $W = \sqrt{(8/3p)}$
Throughput, Loss, and RTT Model

- \( W = \sqrt{\frac{8}{3p}} = \left(\frac{4}{3}\right) \times \sqrt{\frac{3}{2p}} \)

- Recall, bandwidth \( B = \left(\frac{3W}{4} \times \text{packet size}\right) / \text{RTT} \)

\[
B = \text{packet size} / \left(\text{RTT} \times \sqrt{\frac{2p}{3}}\right)
\]

- Consequences:
  1. Increased loss quickly reduces throughput
  2. At same bottleneck, flow with longer RTT achieves less throughput than flow with shorter RTT!
Today

1. Transmission Control Protocol (TCP) primer, cont’d

2. TCP over Wireless
   – TCP Snoop
   – Explicit Loss Notification
Review: TCP on Wireless Links

• TCP interprets any packet loss as a sign of congestion
  – TCP sender reduces congestion window

• On wireless links, packet loss can also occur due to random channel errors, or interference
  – Temporary loss not due to congestion
  – Reducing window may be too conservative
  – Leads to poor throughput
Review: Two Broad Approaches

1. Mask wireless losses from TCP sender
   – Then TCP sender will not reduce congestion window
     – Split Connection Approach
     – TCP Snoop

2. Explicitly notify TCP sender about cause of packet loss
TCP Snoop: Introduction

- Removes most significant problem of split connection: **breaking end-to-end semantics**
  - No more split connection
  - **Single end-to-end connection** like regular TCP

- TCP Snoop only modifies the AP

- **Basic Idea (Downlink traffic):**
  - AP “**snoops**” on TCP traffic to and from the mobile
    - **Quickly retransmits** packets it thinks may be lost over the wireless link
## Snoop Protocol: High-level View

<table>
<thead>
<tr>
<th>application</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
</table>

- **Per TCP-connection state**

![Diagram showing the layers of the protocol stack with a TCP connection and wireless link]

- Re-transmit

TCP connection

wireless
TCP Snoop: Downlink traffic case

- **AP buffers downlink** TCP segments
  - Until it receives **corresponding ACK** from mobile

- **AP snoops on uplink** TCP acknowledgements
  - Detects downlink wireless TCP segment loss via duplicate ACKs or time-out
TCP Snoop Goal: Recover wireless downlink loss

• When AP detects a lost TCP segment:
  – **Locally, quickly retransmit** that segment over the wireless link
  – **Minimize duplicate ACKs** flowing back to server

• **Goal:** Content server **unaware of wireless loss and retransmission**
  – No reduction in cwnd
TCP Snoop: Downlink Example

Snoop Cache (at AP):
TCP segments seen (whose ACKs have not yet been seen)

TCP Segments

TCP ACKs

Wired Internet

Wireless Link
Downlink traffic operation, at Snoop AP

Downlink TCP segments:

Packet arrives

New pkt?
Yes
No

In-sequence?
Yes
No

1. Cache packet
2. Forward to mobile

Common case

1. Forward packet
2. Reset local rexmit counter

Sender rexmission

1. Mark as cong. loss
2. Forward pkt

Congestion loss

Figure 1. Flowchart for snoop_data().
TCP Snoop: Downlink example
1. **A new ACK:** This is the common case (when the connection is fairly error-free and there is little user movement), and signifies an increase in the packet sequence received at the MH. This ACK initiates the cleaning of the snoop cache and all acknowledged packets are freed. The round-trip time estimate for the wireless link is also updated at this time. This estimate is not done for every packet, but only for one packet in each window of transmission, and only if no retransmissions happened in that window. The last condition is needed because it is impossible in general to determine if the arrival of an acknowledgment for a retransmitted packet was for the original packet or for the retransmission [14]. Finally, the acknowledgment is forwarded to the FH.

2. **A spurious ACK:** This is an acknowledgment less than the last acknowledgment seen by the snoop module and is a situation that rarely happens. It is discarded and the packet processing continues.

3. **A duplicate ACK (DUPACK):** This is an ACK that is identical to a previously received one. In particular, it is the same as the highest cumulative ACK seen so far. In this case the next packet in sequence from the DUPACK has not been received by the MH. However, some subsequent packets in the sequence have been received, since the MH generates a DUPACK for each TCP segment received out of sequence. One of several actions is taken depending on the type of duplicate acknowledgment and the current state of snoop:

   • **The first case** occurs when the DUPACK is for a packet that is either not in the snoop cache or has been marked as having been retransmitted by the sender. If the packet is not in the cache, it needs to be resent from the FH, perhaps after invoking the necessary congestion control mechanisms at the sender. If the packet was marked as a sender-retransmitted packet, the DUPACK needs to be routed to the FH because the TCP stack there maintains state based on the number of duplicate acknowledgments it receives when it retransmits a packet. Therefore, both these situations require the DUPACK to be routed to the FH.

   • **The second case** occurs when the snoop module gets a DUPACK that it does not expect to receive for the packet. This typically happens when the first DUPACK arrives for the packet, after a subsequent packet in the stream reaches the MH, following a packet loss. The arrival of each successive packet in the window causes a DUPACK to be generated for the lost packet. In order to make the number of such DUPACKs as small as possible, the lost packet is retransmitted as soon as the loss is detected, and at a higher priority than normal packets. This is done by maintaining two queues at the link layer for high and normal priority packets. In addition, snoop also estimates the maximum number of duplicate acknowledgments that can arrive for this packet. This is done by counting the number of packets that were transmitted after the lost packet prior to its retransmission.
TCP Snoop: Downlink example
TCP Snoop: Downlink example

- TCP receiver does not delay duplicate ACKs (dupacks)
TCP Snoop: Downlink example
1. A **new** ACK: This is the common case (when the connection is fairly error-free and there is little user movement), and signifies an increase in the packet sequence received at the MH. This ACK initiates the cleaning of the snoop cache and all acknowledged packets are freed. The round-trip time estimate for the wireless link is also updated at this time. This estimate is not done for every packet, but only for one packet in each window of transmission, and only if no retransmissions happened in that window. The last condition is needed because it is impossible in general to determine if the arrival of an acknowledgment for a retransmitted packet was for the original packet or for the retransmission [14]. Finally, the acknowledgment is forwarded to the FH.

2. A **spurious** ACK: This is an acknowledgment less than the last acknowledgment seen by the snoop module and is a situation that rarely happens. It is discarded and the packet processing continues.

3. A **duplicate** ACK (DUPACK): This is an ACK that is identical to a previously received one. In particular, it is the same as the highest cumulative ACK seen so far. In this case the next packet in sequence from the DUPACK has not been received by the MH. However, some subsequent packets in the sequence have been received, since the MH generates a DUPACK for each TCP segment received out of sequence. One of several actions is taken depending on the type of duplicate acknowledgment and the current state of snoop:

   - **The first case occurs when the DUPACK is for a packet that is either not in the snoop cache or has been marked as having been retransmitted by the sender.** If the packet is not in the cache, it needs to be resent from the FH, perhaps after invoking the necessary congestion control mechanisms at the sender. If the packet was marked as a sender-retransmitted packet, the DUPACK needs to be routed to the FH because the TCP stack there maintains state based on the number of duplicate acknowledgments it receives when it retransmits a packet. Therefore, both these situations require the DUPACK to be routed to the FH.

   - **The second case occurs when the snoop module gets a DUPACK that it does not expect to receive for the packet.** This typically happens when the first DUPACK arrives for the packet, after a subsequent packet in the stream reaches the MH, following a packet loss. The arrival of each successive packet in the window causes a DUPACK to be generated for the lost packet. In order to make the number of such DUPACKs as small as possible, the lost packet is retransmitted as soon as the loss is detected, and at a higher priority than normal packets. This is done by maintaining two queues at the link layer for high and normal priority packets. In addition, snoop also estimates the maximum number of duplicate acknowledgments that can arrive for this packet. This is done by counting the number of packets that were transmitted after the lost packet prior to its retransmission.
TCP Snoop: Downlink example

- Dupack triggers retransmission of packet 37 from AP
1. A new ACK: This is the common case (when the connection is fairly error-free and there is little user movement), and signifies an increase in the packet sequence received at the MH. This ACK initiates the cleaning of the snoop cache and all acknowledged packets are freed. The round-trip time estimate for the wireless link is also updated at this time. This estimate is not done for every packet, but only for one packet in each window of transmission, and only if no retransmissions happened in that window. The last condition is needed because it is impossible in general to determine if the arrival of an acknowledgment for a retransmitted packet was for the original packet or for the retransmission [14]. Finally, the acknowledgment is forwarded to the FH.

2. A spurious ACK: This is an acknowledgment less than the last acknowledgment seen by the snoop module and is a situation that rarely happens. It is discarded and the packet processing continues.

3. A duplicate ACK (DUPACK): This is an ACK that is identical to a previously received one. In particular, it is the same as the highest cumulative ACK seen so far. In this case the next packet in sequence from the DUPACK has not been received by the MH. However, some subsequent packets in the sequence have been received, since the MH generates a DUPACK for each TCP segment received out of sequence. One of several actions is taken depending on the type of duplicate acknowledgment and the current state of snoop:

   - The first case occurs when the DUPACK is for a packet that is either not in the snoop cache or has been marked as having been retransmitted by the sender. If the packet is not in the cache, it needs to be resent from the FH, perhaps after invoking the necessary congestion control mechanisms at the sender. If the packet was marked as a sender-retransmitted packet, the DUPACK needs to be routed to the FH because the TCP stack there maintains state based on the number of duplicate acknowledgments it receives when it retransmits a packet. Therefore, both these situations require the DUPACK to be routed to the FH.

   - The second case occurs when the snoop module gets a DUPACK that it does not expect to receive for the packet. This typically happens when the first DUPACK arrives for the packet, after a subsequent packet in the stream reaches the MH, following a packet loss. The arrival of each successive packet in the window causes a DUPACK to be generated for the lost packet. In order to make the number of such DUPACKs as small as possible, the lost packet is retransmitted as soon as the loss is detected, and at a higher priority than normal packets. This is done by maintaining two queues at the link layer for high and normal priority packets. In addition, snoop also estimates the maximum number of duplicate acknowledgments that can arrive for this packet. This is done by counting the number of packets that were transmitted after the lost packet prior to its retransmission.
TCP Snoop: Downlink example

Discard 2\textsuperscript{nd} dupack

\begin{itemize}
\item 45
\item 44
\item 2nd dupack
\item 36
\end{itemize}
TCP Snoop: Downlink example

Discard 3rd dupack

36 36

37 40 43
38 41 44
39 42

46 45 43 42

41

36
TCP Snoop: Downlink example

- TCP sender does not fast retransmit

Discard 4th dupack

36 36 36

• TCP sender does not fast retransmit
TCP Snoop: Downlink example
Uplink traffic case

- Less-common case but becoming more prevalent
- Buffer & retransmit TCP segments at AP? Not likely useful
- Run Snoop agent on the Mobile? Not likely useful
AP detects wireless uplink loss via **missing sequence numbers**

AP immediately sends L2 **negative ACK (NACK)** to mobile
- Mobile quickly & selectively **retransmits data**
  - Requires modification to AP and mobile’s link layer
Snoop TCP: Advantages

• Downlink works without modification to mobile or server

• Preserves end-to-end semantics. Crash does not affect correctness, only performance.

• After an AP handoff: New AP needn’t Snoop TCP
  – Can automatically fall back to regular TCP operation
  – No state need be migrated (but if done, can improve performance)
  – Note such “state” is called soft state
    • Good if available, but correct functionality otherwise
Negative ACKs: Critique

- Mobile host still needs to be modified at L2 and L4
  - This applies to NACK scheme for uplink traffic, **not** Snoop for downlink traffic

- Violates the layering principle

- *Almost* violates the end-to-end principle
Two Broad Approaches

1. Mask wireless losses from TCP sender
   – Then TCP sender will not reduce congestion window
   – Split Connection Approach
   – TCP Snoop

2. Explicitly notify TCP sender about cause of packet loss
Explicit Loss Notification (ELN)

• Notify the TCP sender that a wireless link (not congestion) caused a certain packet loss.

• Upon notification, TCP sender retransmits packet, but doesn’t reduce congestion window.

• Many design options:
  – Who sends notification? How is notification sent? How is notification interpreted at sender?
  • We’ll discuss one example approach.
ELN for uplink TCP traffic

- AP keeps track of gaps in the TCP packet sequence received from the mobile sender
ELN for uplink TCP traffic

- When **AP** sees a **dupack**:  
  - **AP compares** dupack seqno with its recorded gaps  
  - **If match**: **AP sets ELN bit** in dupack and forwards it

- When **mobile** receives **dupack with ELN bit set**:  
  - Resends packet, but **doesn’t reduce** congestion window

---

**Diagram:**

- Mobile sends TCP segments: 4, 3, 1 (with gaps marked)
- Server receives segments 4, 3, 1
- AP sees dupack with ELN set: 4, 3, 1
- Mobile receives dupack with ELN set and resends packet.
Thursday Topic:
Link Layer I: Time, Frequency, and Code Division

Friday Precept
Introduction to Lab 1