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

COS 463: Wireless Networks
Lecture 3
Kyle Jamieson

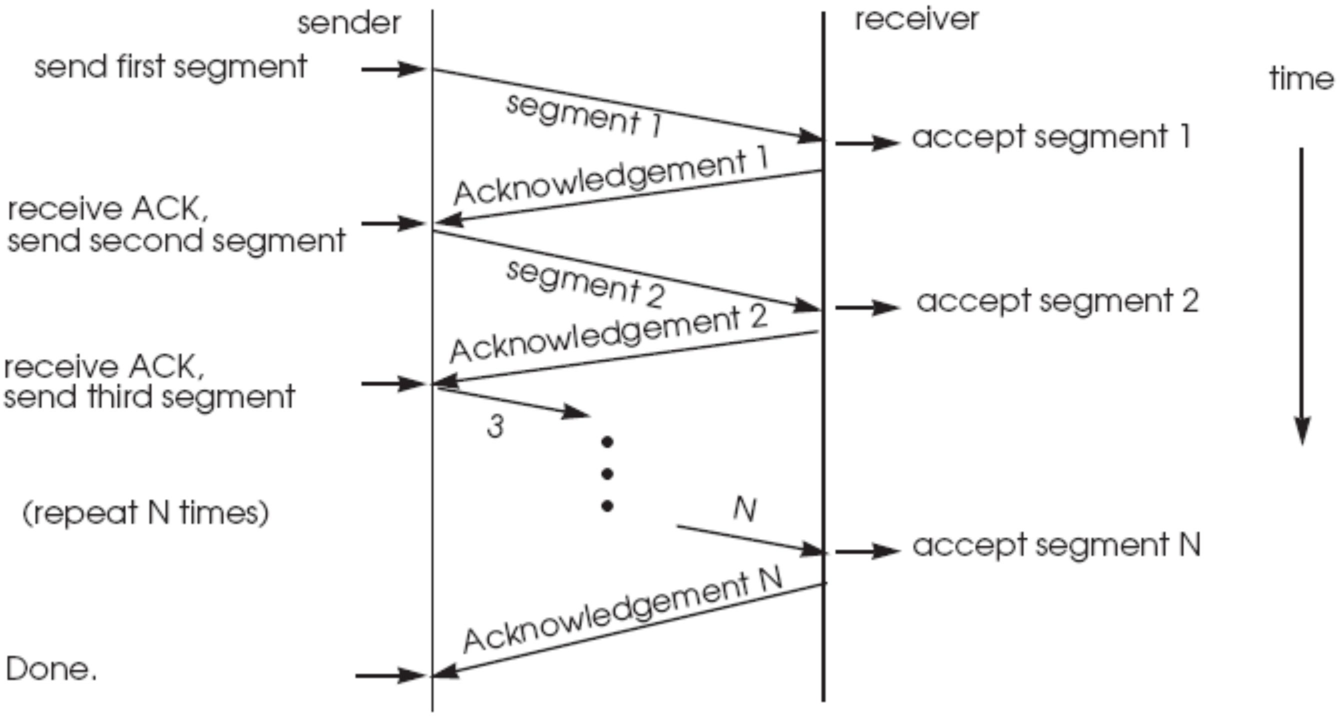
[Various parts adapted from S. Das, B. Karp, N. Vaidya]

Today

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

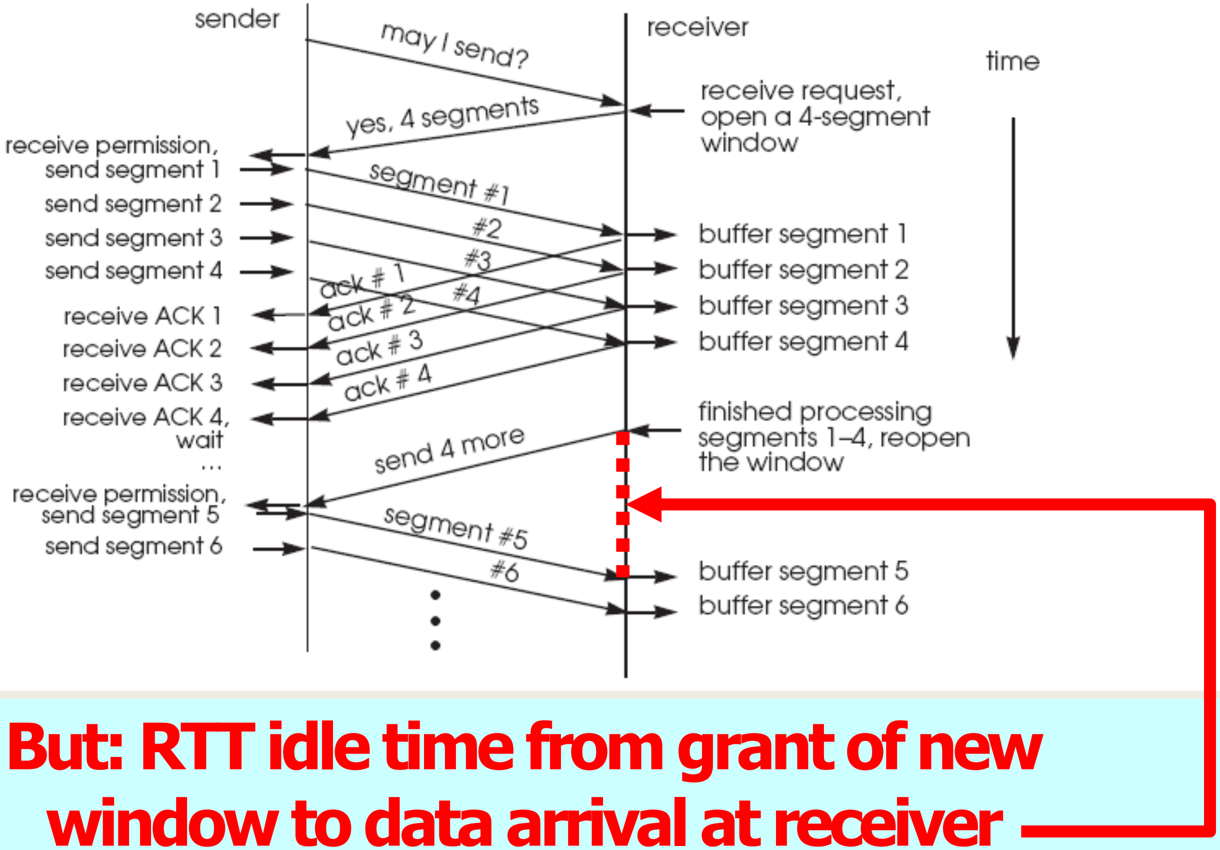
2. TCP over Wireless
    - TCP Snoop
    - Explicit Loss Notification

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

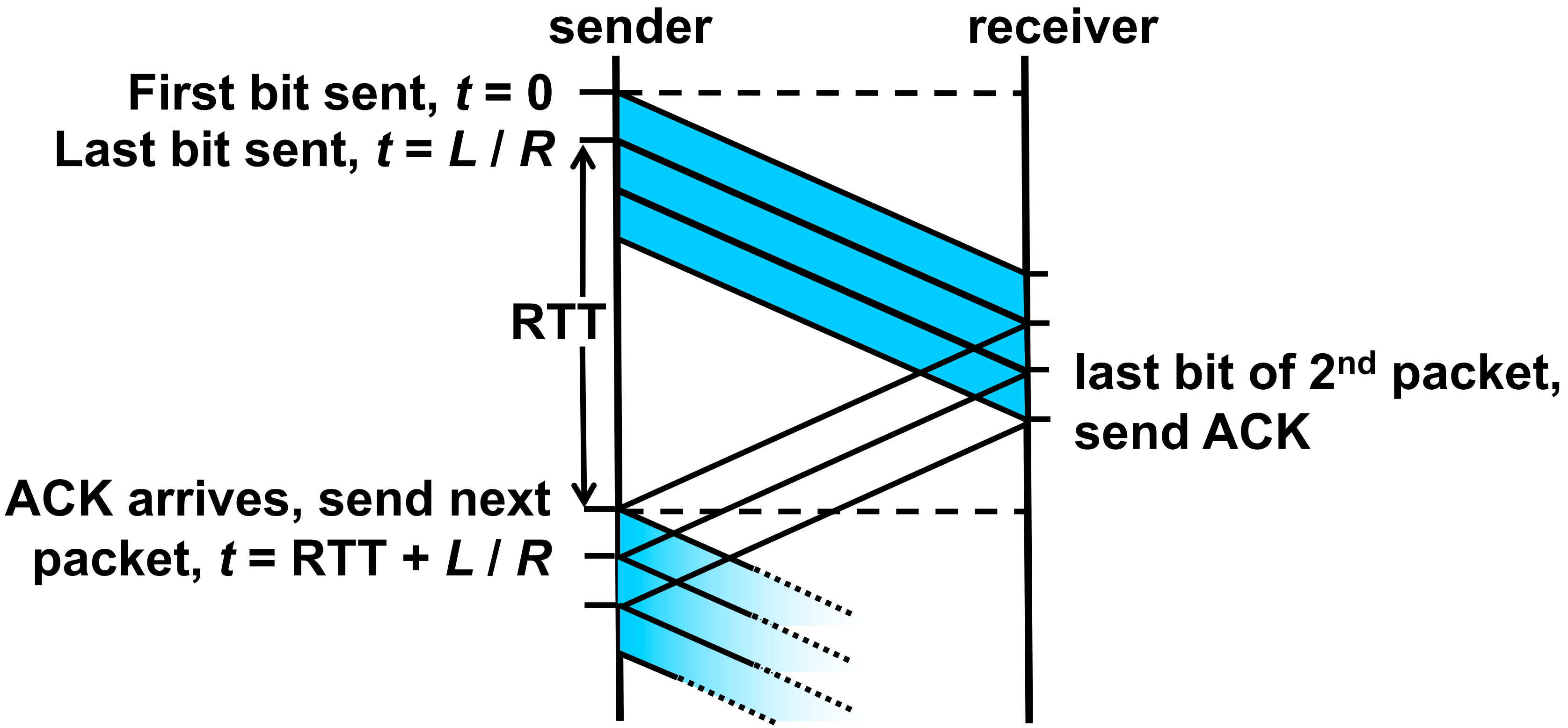
- **Network *bottleneck:*** point of **slowest rate** along path **between sender and receiver**

- 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

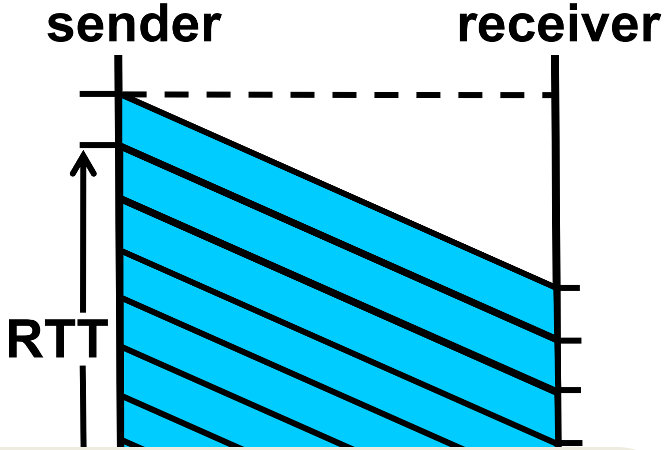


The bandwidth-delay product

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

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

$$\underbrace{(N-1)L}_{\text{Number of bits “in flight”}} = \underbrace{\mathrm{RTT}\cdot R}_{\text{Delay × Bandwidth product}}$$

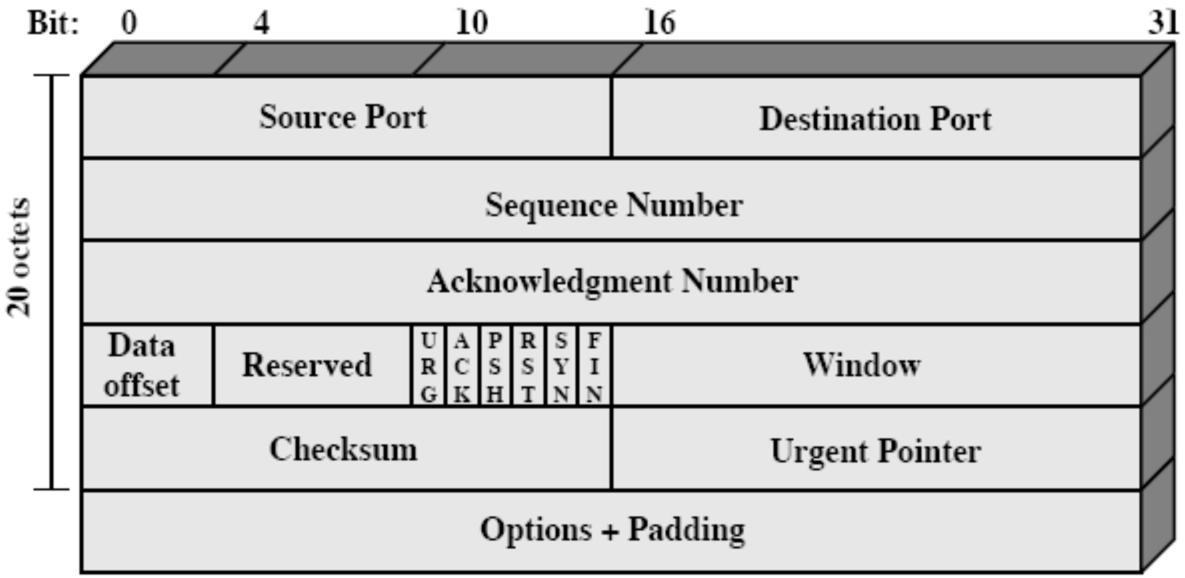


Goal: window size = RTT × bottleneck rate

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

W = (10^6 bps × .07 s) = 70 Kbits = 8.75 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

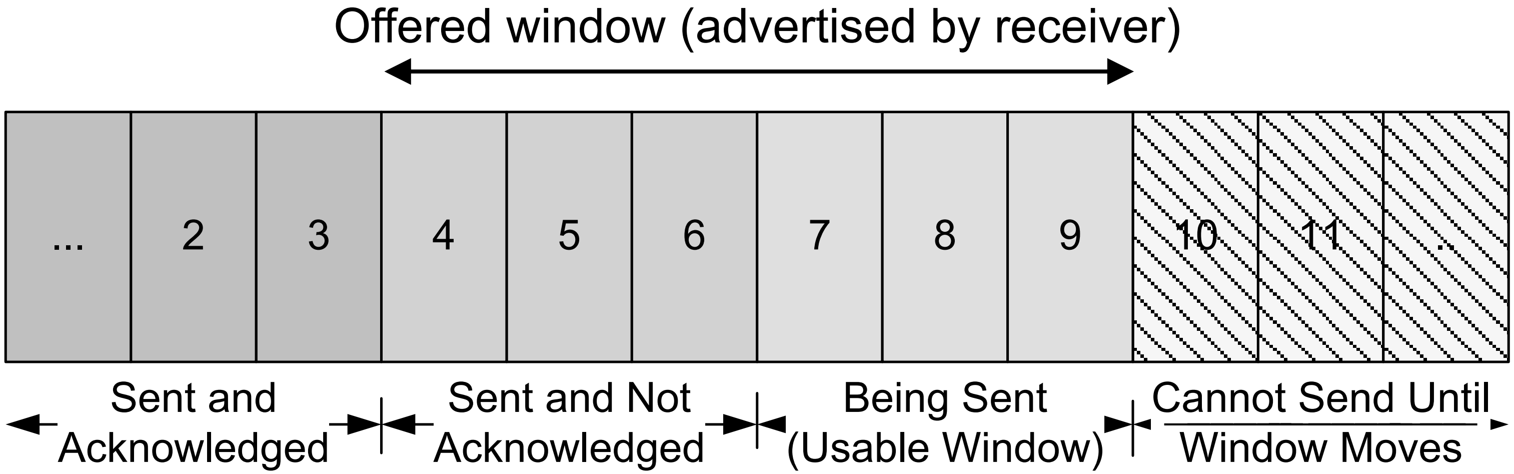
- **Sequence number (seqno) field** in TCP header indicates first user payload byte in packet

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



Offered window (advertised by receiver)

Sent and Acknowledged

Sent and Not Acknowledged

Being Sent (Usable Window)

Cannot Send Until Window Moves

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

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 mi from timed packet/ACK pairs
  - $RTTi = ((1 - \alpha) x RTTi - 1 + \alpha x mi)$

- **Original TCP retransmit timeout: $RTOi = \beta \times RTTi$**
  - **original TCP: $\beta = 2$**

Mean and Variance: Jacobson's RTT Estimator

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

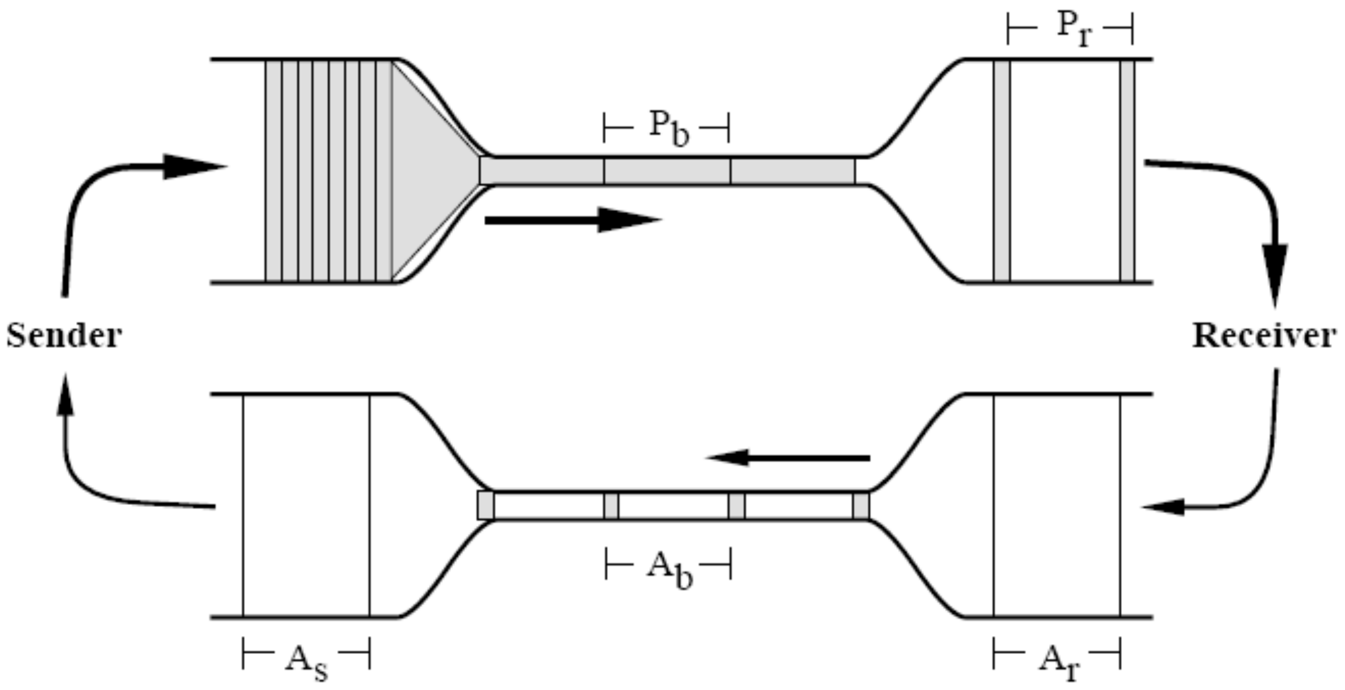
- **Idea** [Jacobson 88]: Estimate ***mean deviation*** **v_i,** (EWMA of $|m_i - \mathrm{RTT}_i|$), a stand-in for variance:

  $$v_i = v_{i-1} \times (1-\gamma) + \gamma \times |m_i - \mathrm{RTT}_i|$$

  - Then use retransmission timeout **$\mathrm{RTO}_i = \mathrm{RTT}_i + 4v_i$**

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

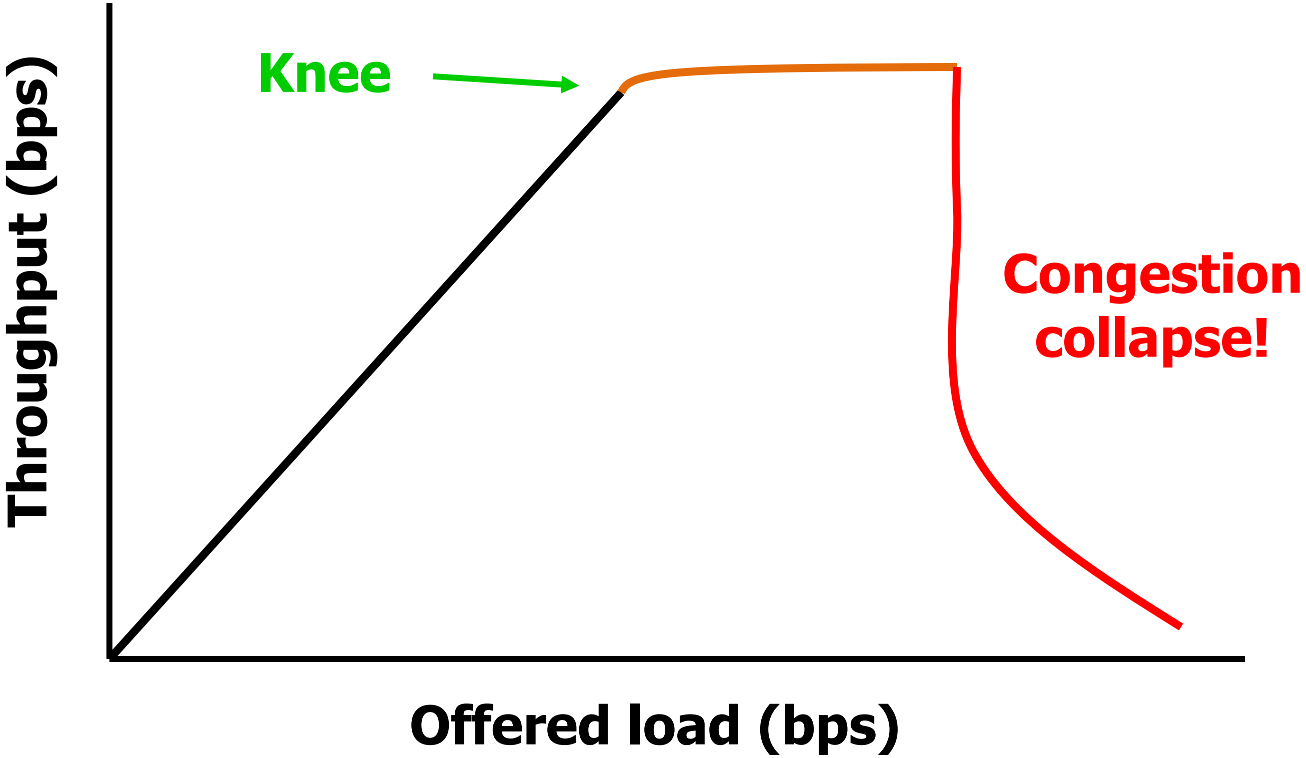
1. Achieve **high utilization** on links; don’t waste capacity!

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 $\log_2(W / \texttt{pktSize})$ 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 **timeout** for sent packet, sender **presumes packet lost to congestion,** and:
  1. sets `ssthresh = cwnd / 2`
  2. sets `cwnd = 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,
    `cwnd = cwnd + (pktSize × pktSize) / cwnd`
  - Increases cwnd by ≈ `pktSize` bytes per RTT

Combined algorithm: Additive Increase, Multiplicative 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**

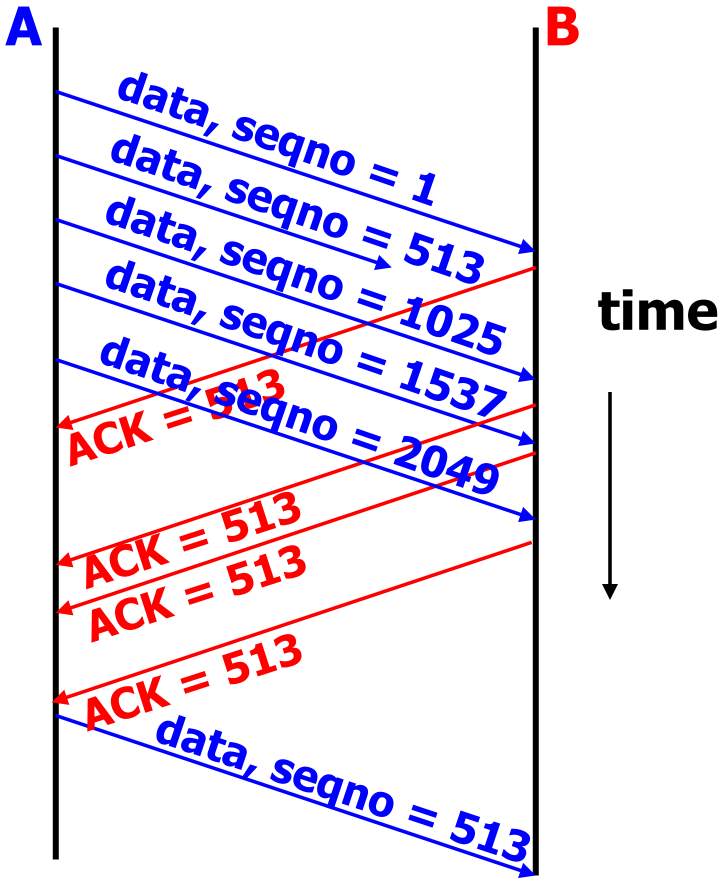
- **Another indicator of loss:**
  - Suppose sender sends: 1, 2, 3, 4, 5 (...but **2 is lost**)
  - Receiver receives: 1, 3, 4, 5
  - Receiver sends cumulative ACKs: 2, 2, 2, 2
    - Loss causes ***duplicate ACKs***

Fast Retransmit (II)

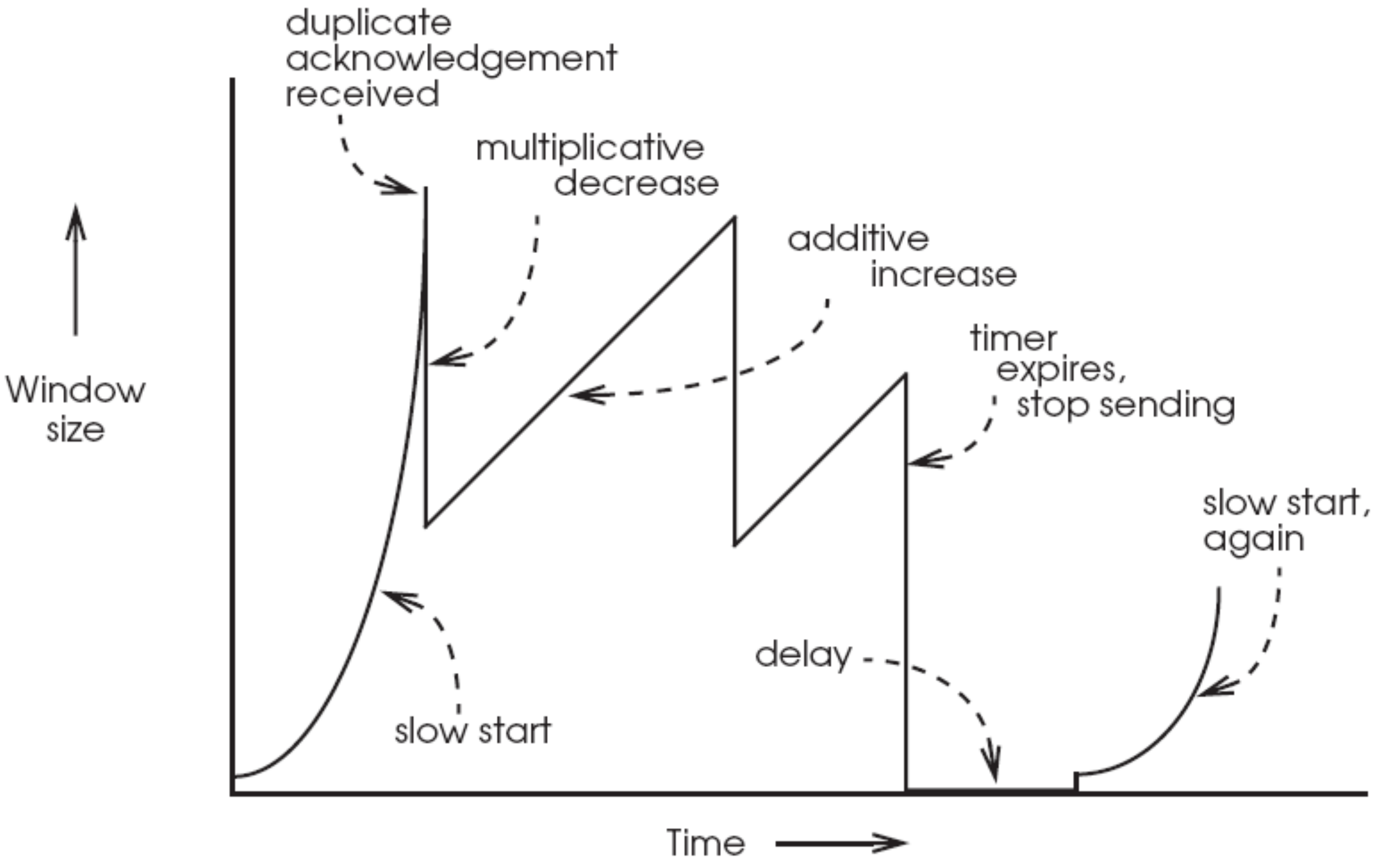
- Upon arrival of **three duplicate ACKs,** sender:

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



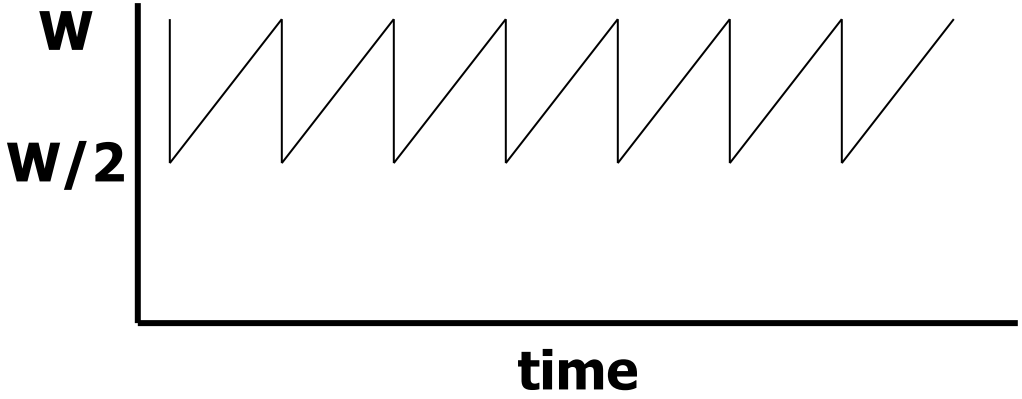
Modeling Throughput, Loss, and RTT

- *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: ¾W**
- One window of packets is sent per RTT
- Bandwidth:
  - **¾W** packets per RTT
  - **(¾W x packet size) / 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 **W/2 RTTs** to go from window size W/2 to window size W; this period is one ***cycle***
- How many packets sent in total, in a cycle?
  - $(\frac{3}{4}W / RTT) \times (W/2 \times RTT) = 3W^2/8$
- 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

- $W = \sqrt{(8/3p)} = (4/3) \times \sqrt{(3/2p)}$

- Recall, **bandwidth *B*** = (3W/4 x packet size) / RTT

$$B = \text{packet size} / (RTT \times \sqrt{(2p/3)})$$

- **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

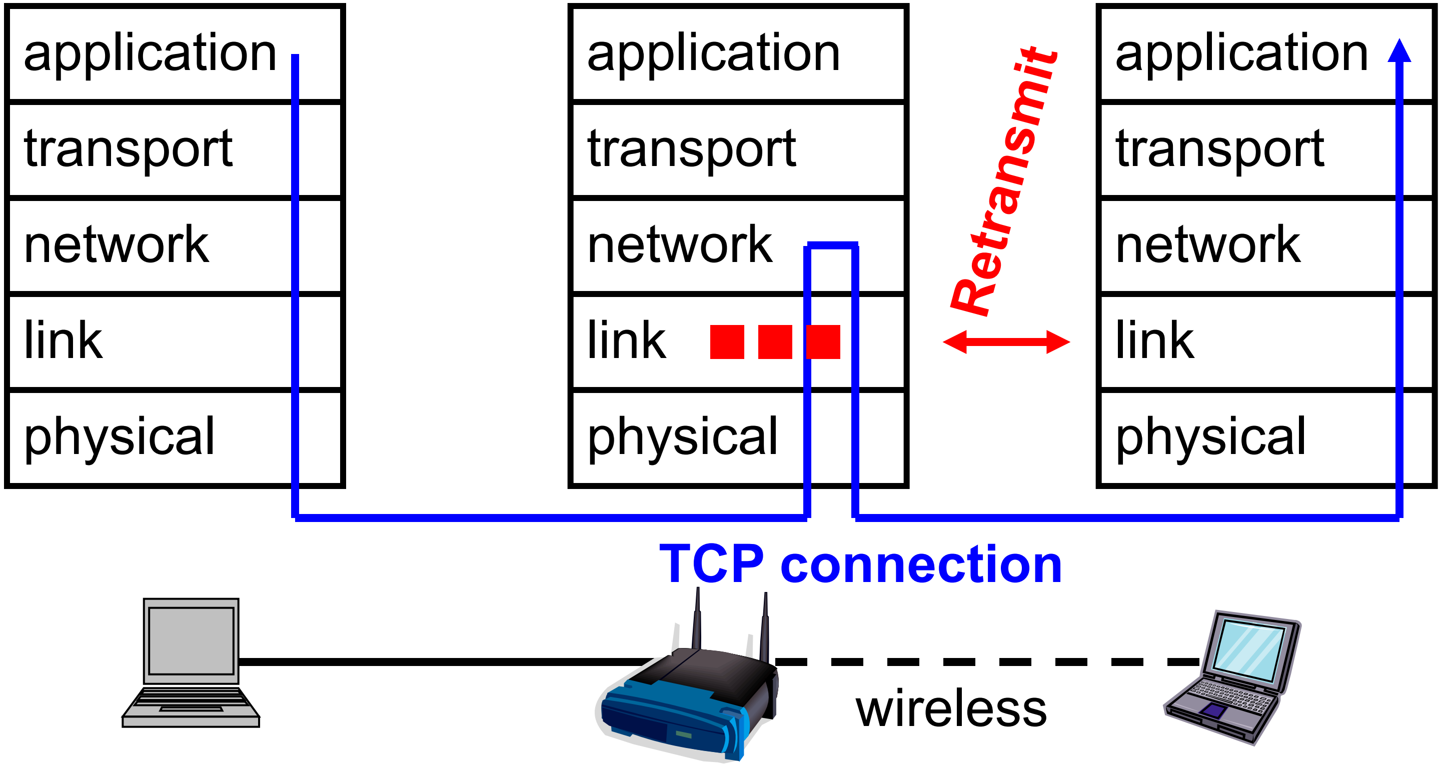
- Removes most significant problem of split connection: **breaking end-to-end semantics**
  - No more split connection
  - **Single end-to-end connection** like regular TCP

- TCP Snoop only modifies the AP

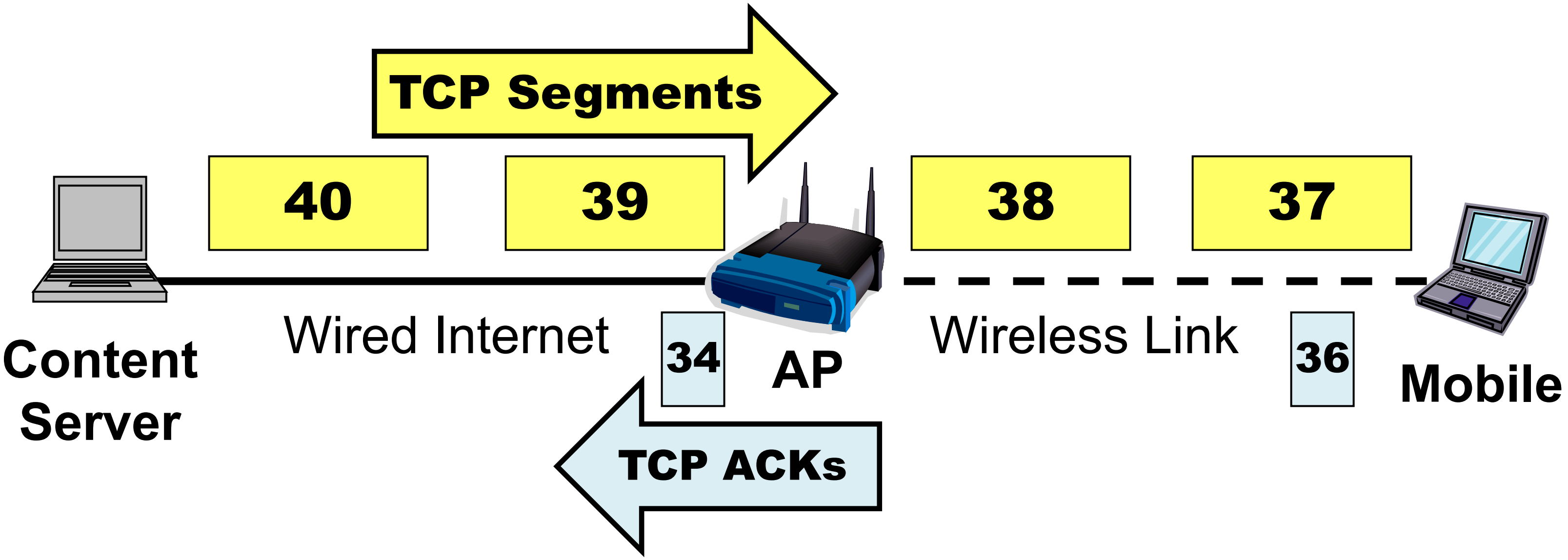
- **Basic Idea (Downlink traffic):**
  - AP **"snoops" on TCP traffic** to and from the mobile
    - **Quickly retransmits** packets it thinks may be lost over the wireless link

Snoop Protocol: High-level View

🟥 **Per TCP-connection state**



TCP Snoop: Downlink traffic case



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

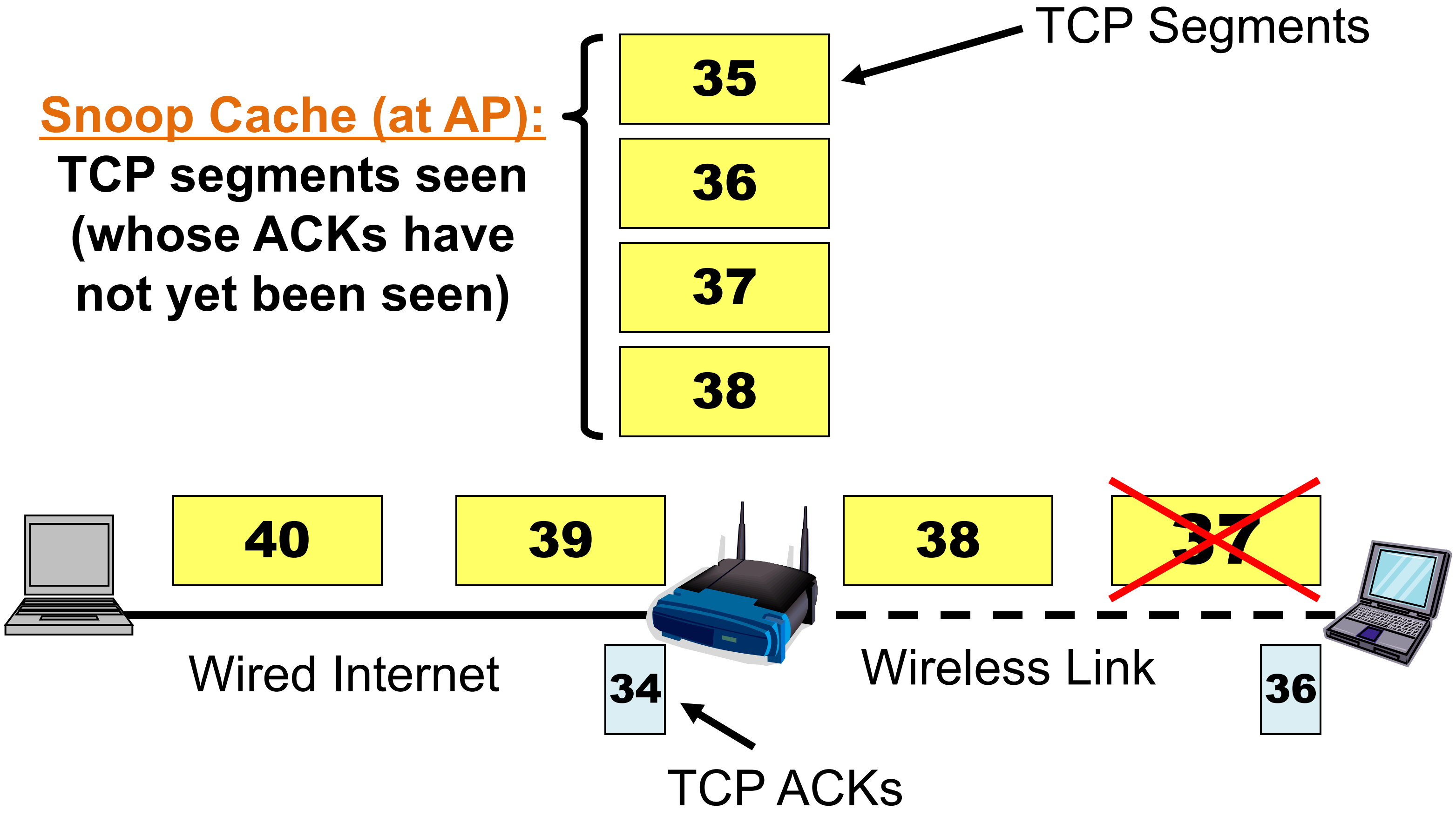
- AP **snoops on uplink** TCP acknowledgements
  - Detects downlink wireless TCP segment loss via duplicate ACKs or time-out

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

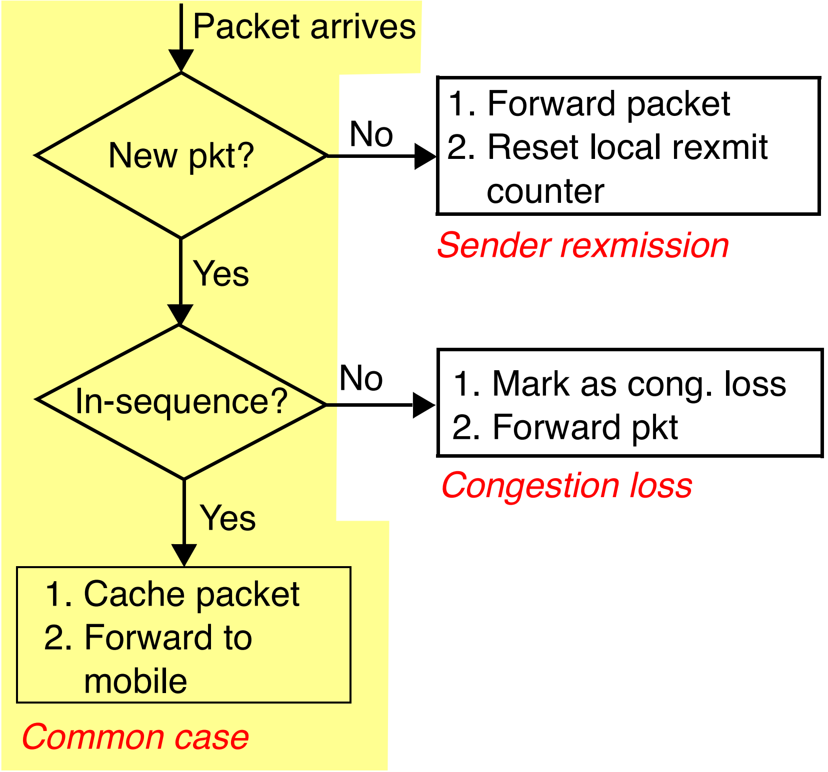
- **Goal:** Content server **unaware of wireless loss and retransmission**
  - **No reduction in `cwnd`**

TCP Snoop: Downlink Example

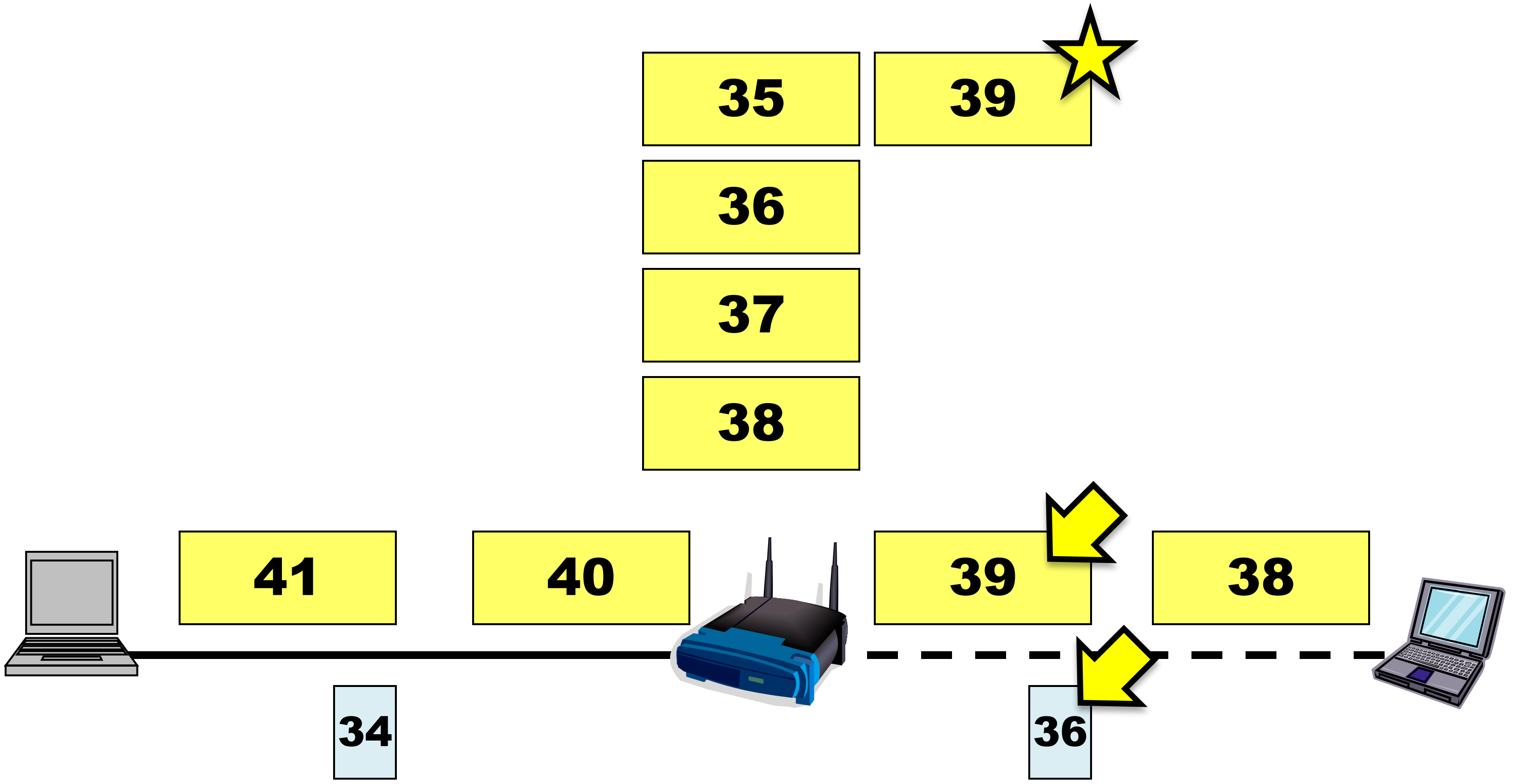


Downlink traffic operation, at Snoop AP

Downlink TCP segments:

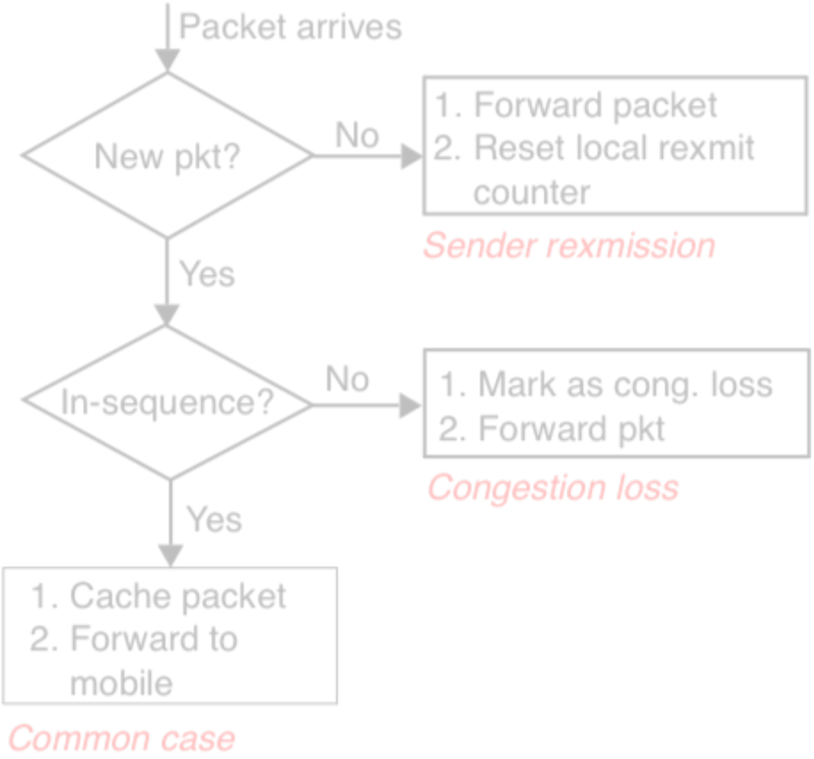


TCP Snoop: Downlink example



Downlink traffic operation, at Snoop AP

Downlink TCP segments:



Packet arrives

New pkt?

No

1. Forward packet
2. Reset local rexmit counter

Sender rexmission

Yes

In-sequence?

No

1. Mark as cong. loss
2. Forward pkt

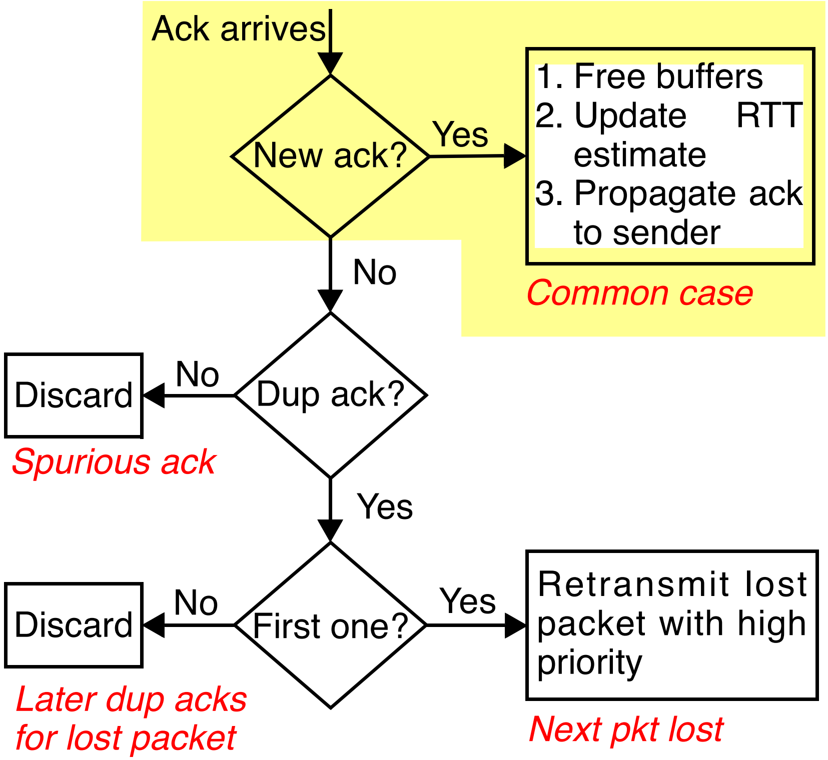
Congestion loss

Yes

1. Cache packet
2. Forward to mobile

Common case

Uplink TCP ACKs:



Ack arrives

New ack?

Yes

1. Free buffers
2. Update RTT estimate
3. Propagate ack to sender

Common case

No

Dup ack?

No

Discard

Spurious ack

Yes

First one?

No

Discard

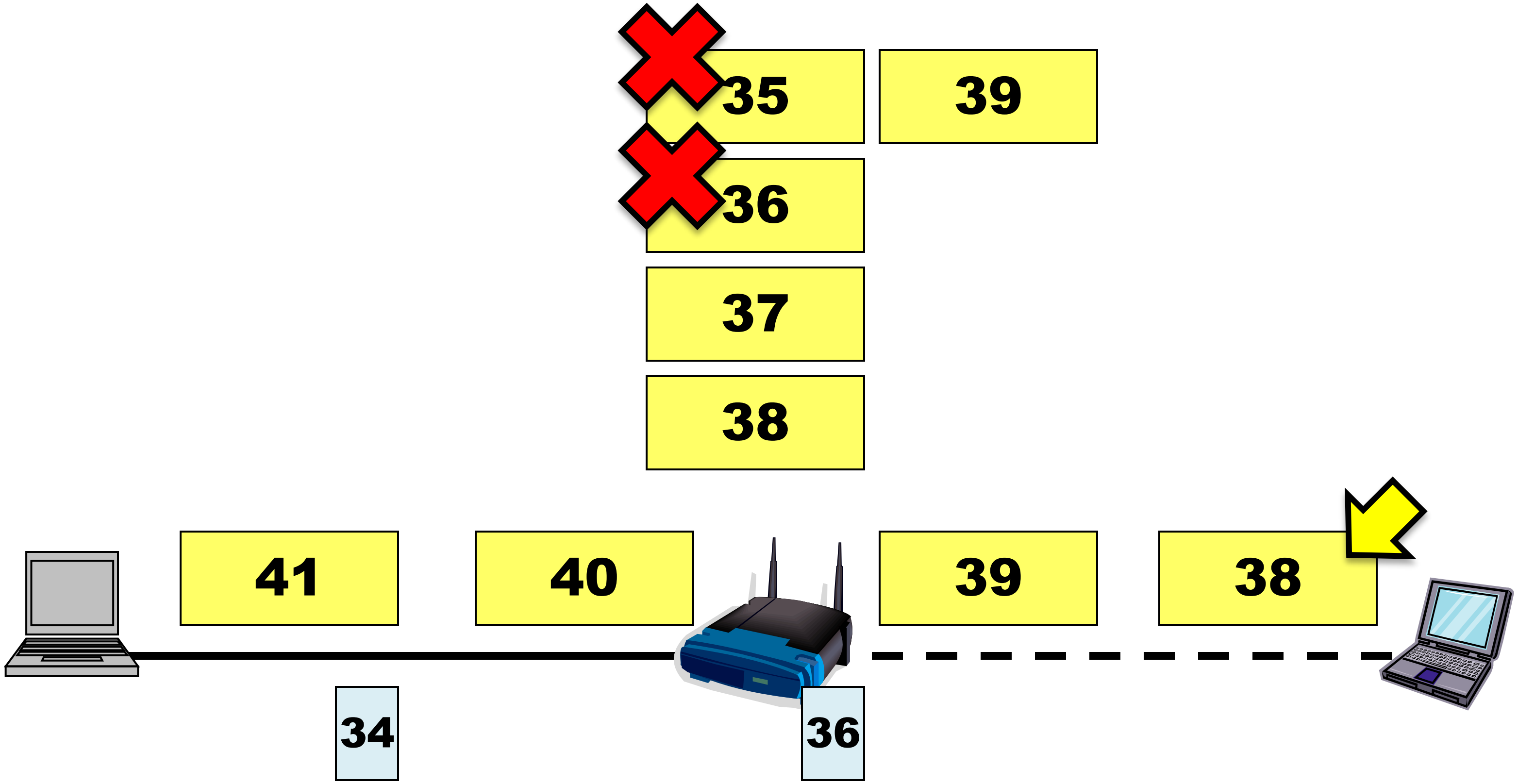
Later dup acks for lost packet

Yes

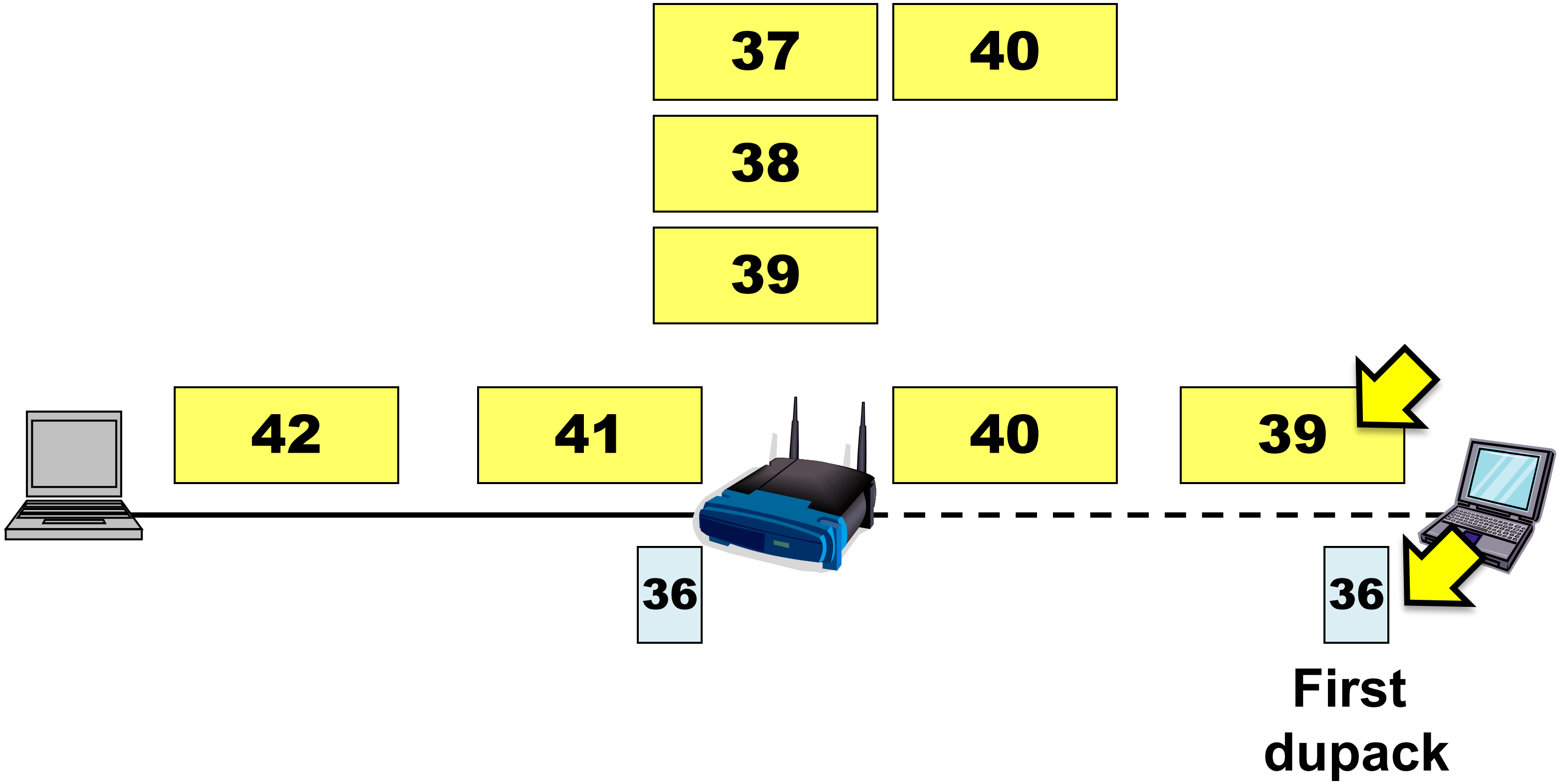
Retransmit lost packet with high priority

Next pkt lost

TCP Snoop: Downlink example

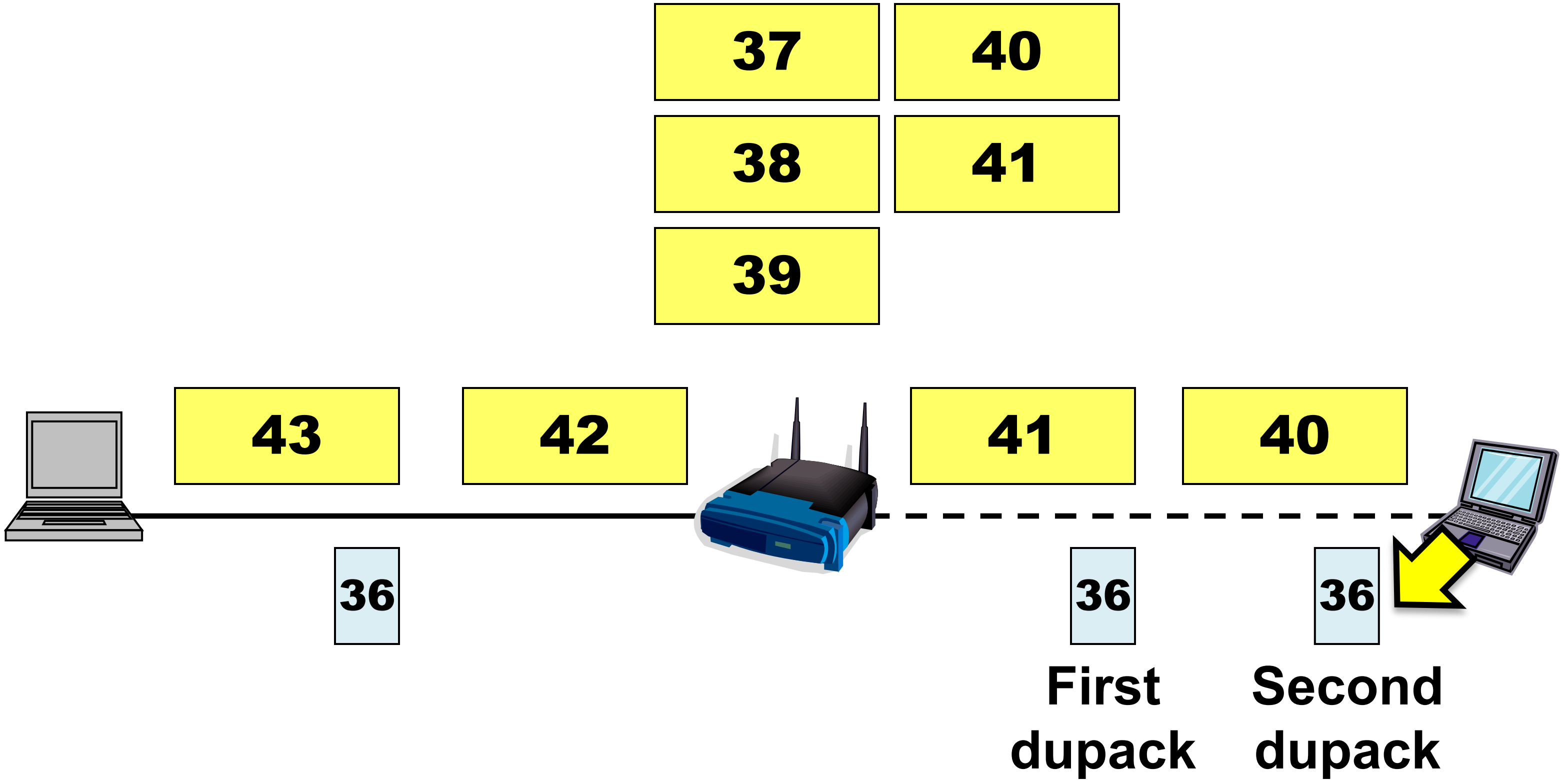


TCP Snoop: Downlink example



- TCP receiver does not delay duplicate ACKs (*dupacks*)

TCP Snoop: Downlink example

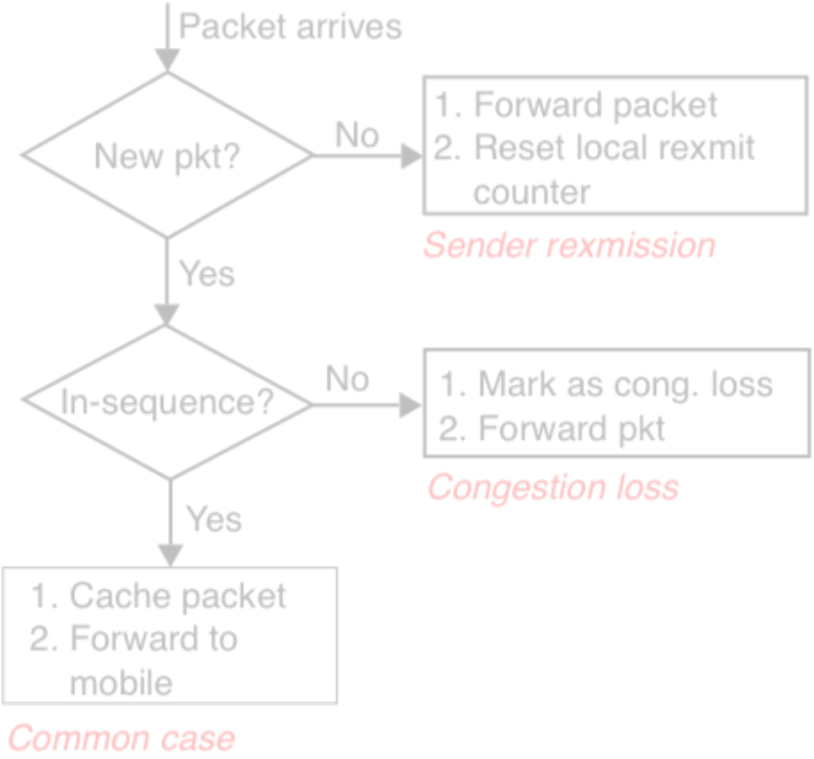


First dupack

Second dupack

Downlink traffic operation, at Snoop AP

Downlink TCP segments:



Packet arrives

New pkt?

No

1. Forward packet
2. Reset local rexmit counter

Sender rexmission

Yes

In-sequence?

No

1. Mark as cong. loss
2. Forward pkt

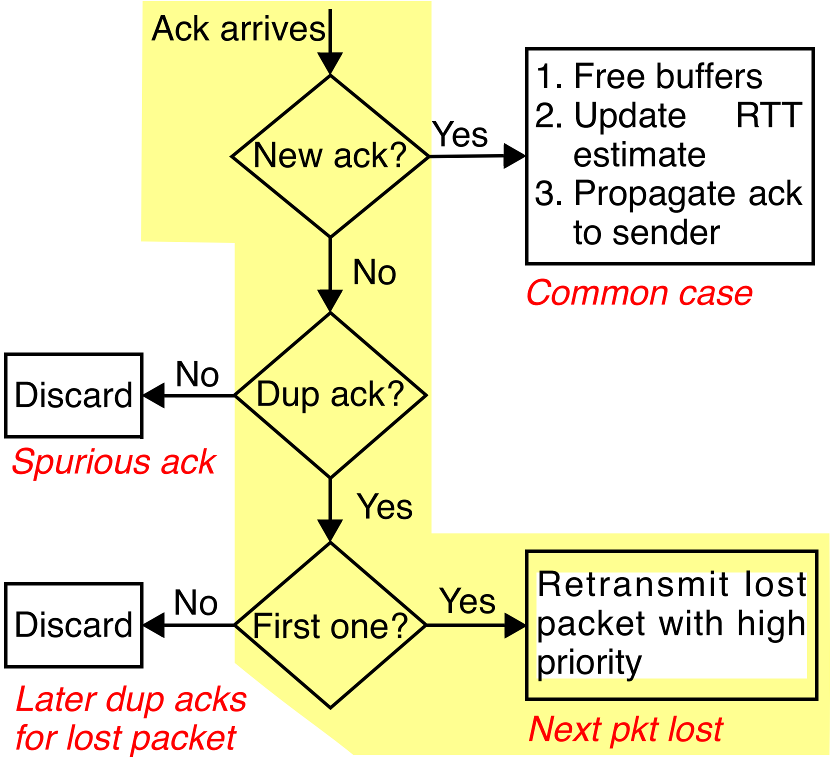
Congestion loss

Yes

1. Cache packet
2. Forward to mobile

Common case

Uplink TCP ACKs:



Ack arrives

New ack?

Yes

1. Free buffers
2. Update RTT estimate
3. Propagate ack to sender

Common case

No

Dup ack?

No

Discard

Spurious ack

Yes

First one?

No

Discard

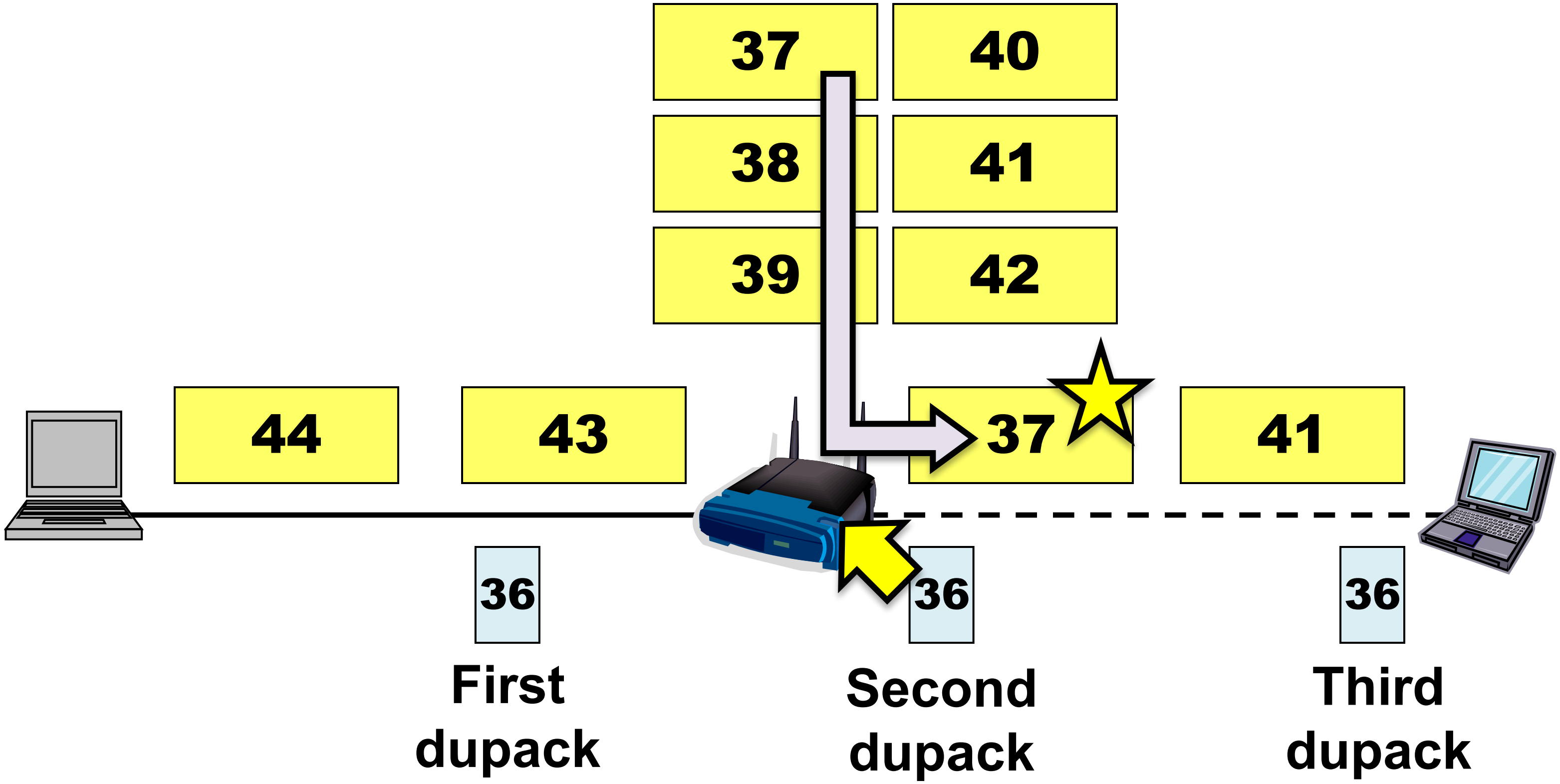
Later dup acks for lost packet

Yes

Retransmit lost packet with high priority

Next pkt lost

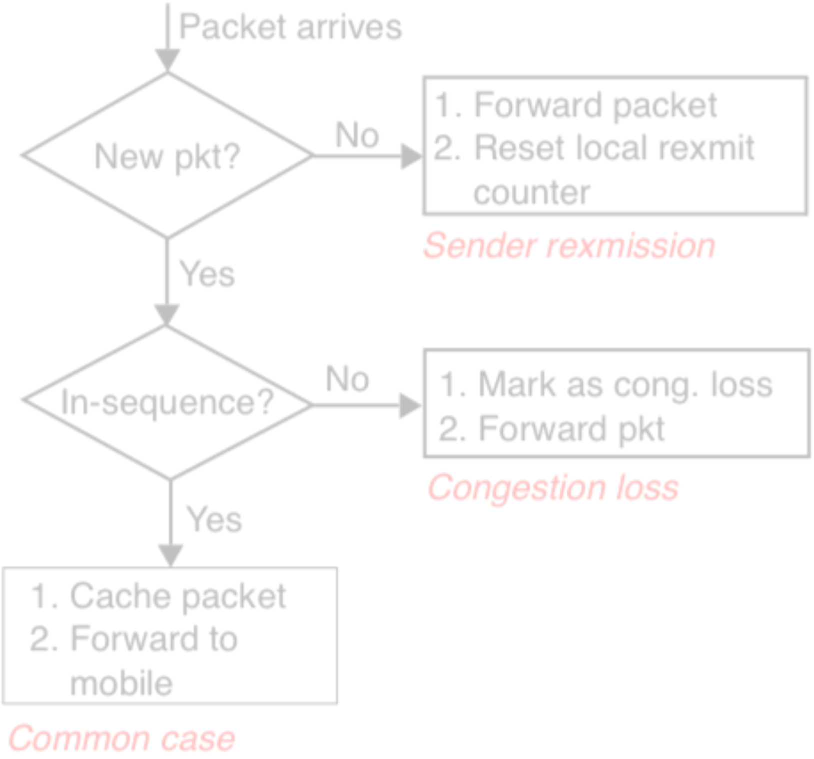
TCP Snoop: Downlink example



- Dupack triggers retransmission of packet 37 from AP

Downlink traffic operation, at Snoop AP

Downlink TCP segments:



Packet arrives

New pkt?

No

1. Forward packet
2. Reset local rexmit counter

Sender rexmission

Yes

In-sequence?

No

1. Mark as cong. loss
2. Forward pkt

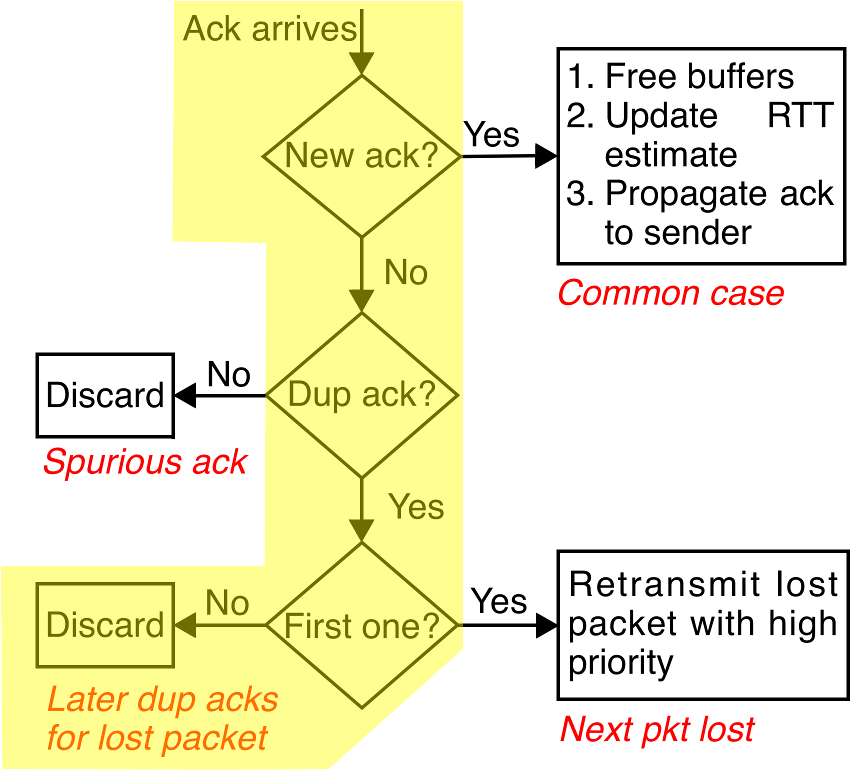
Congestion loss

Yes

1. Cache packet
2. Forward to mobile

Common case

Uplink TCP ACKs:



Ack arrives

New ack?

Yes

1. Free buffers
2. Update RTT estimate
3. Propagate ack to sender

Common case

No

Dup ack?

No

Discard

Spurious ack

Yes

First one?

No

Discard

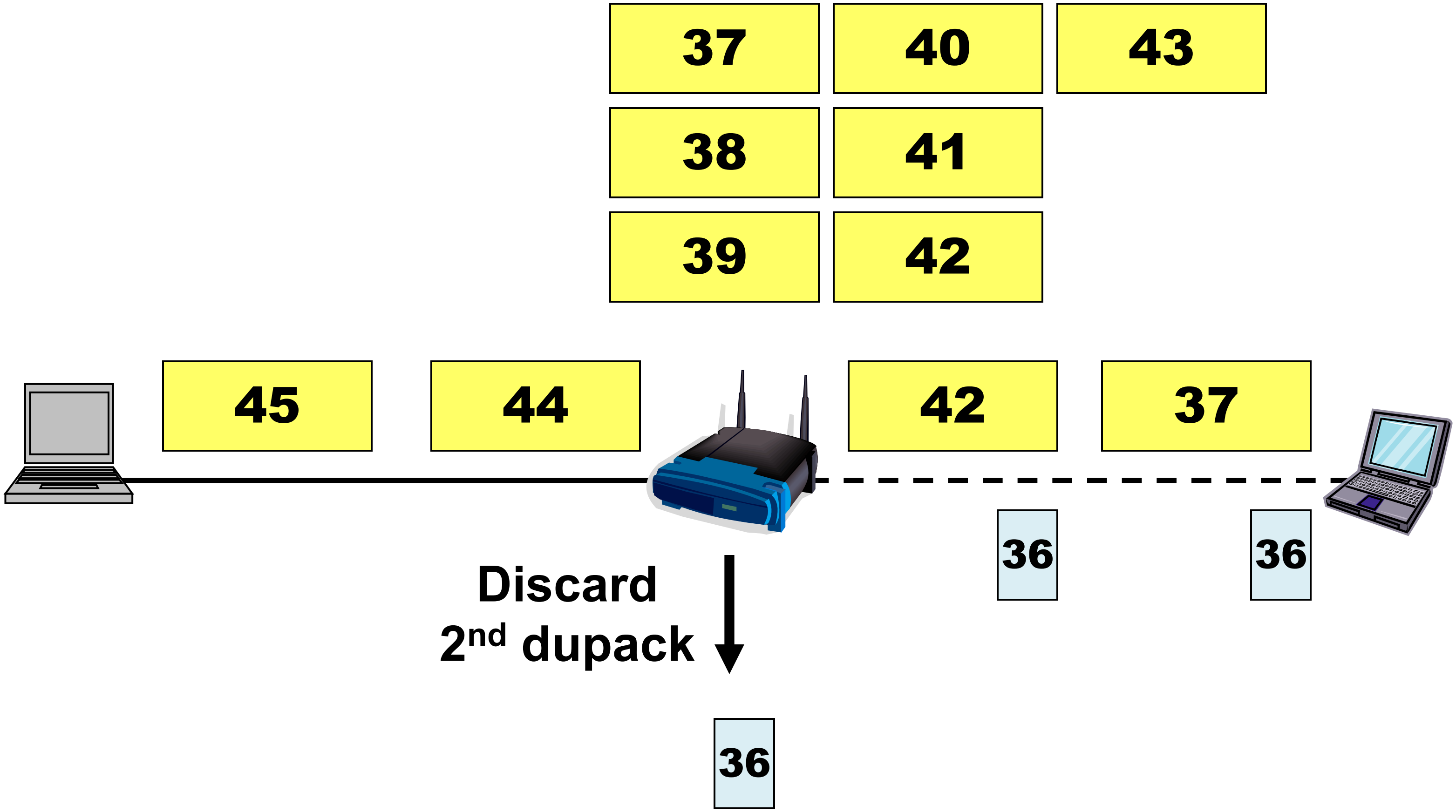
Later dup acks for lost packet

Yes

Retransmit lost packet with high priority

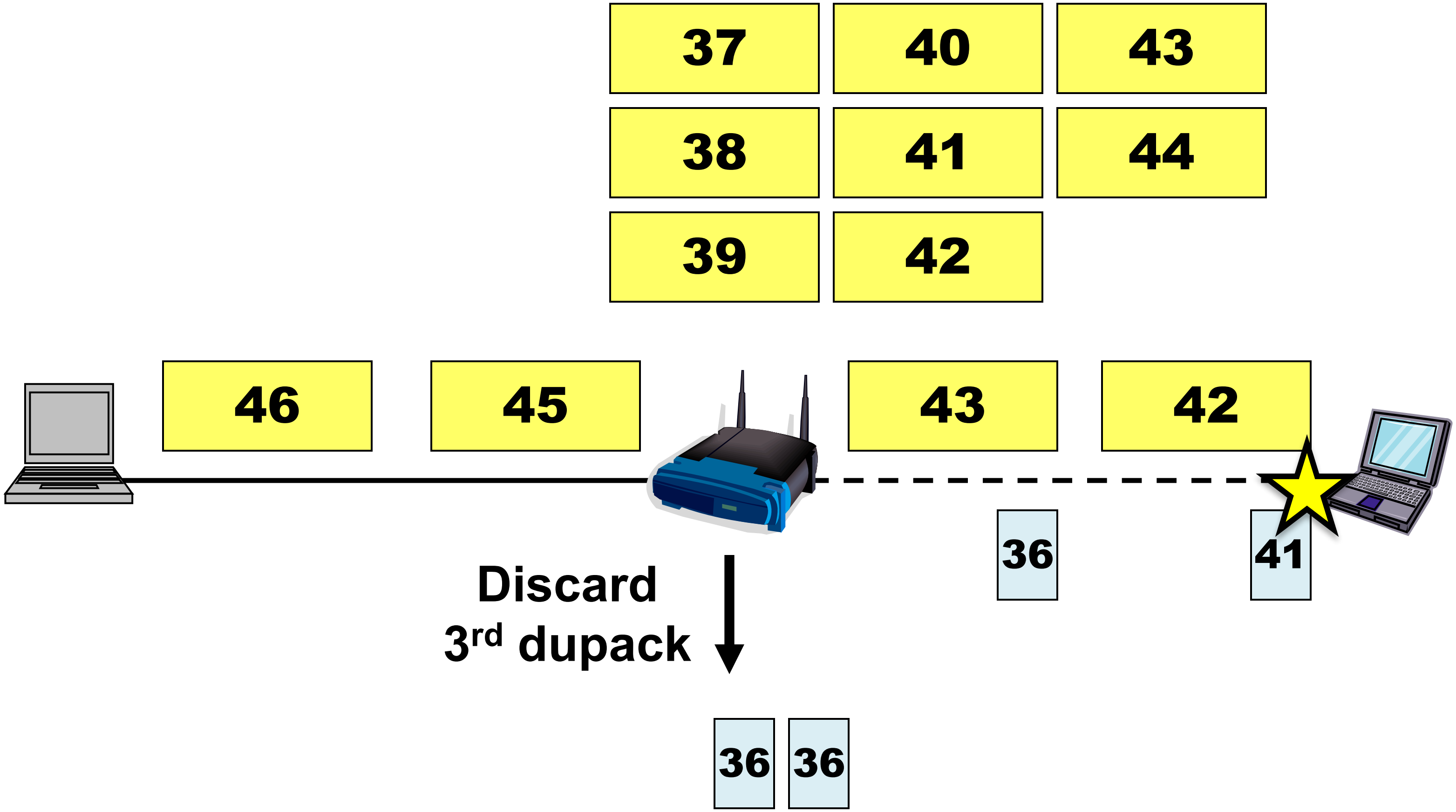
Next pkt lost

TCP Snoop: Downlink example



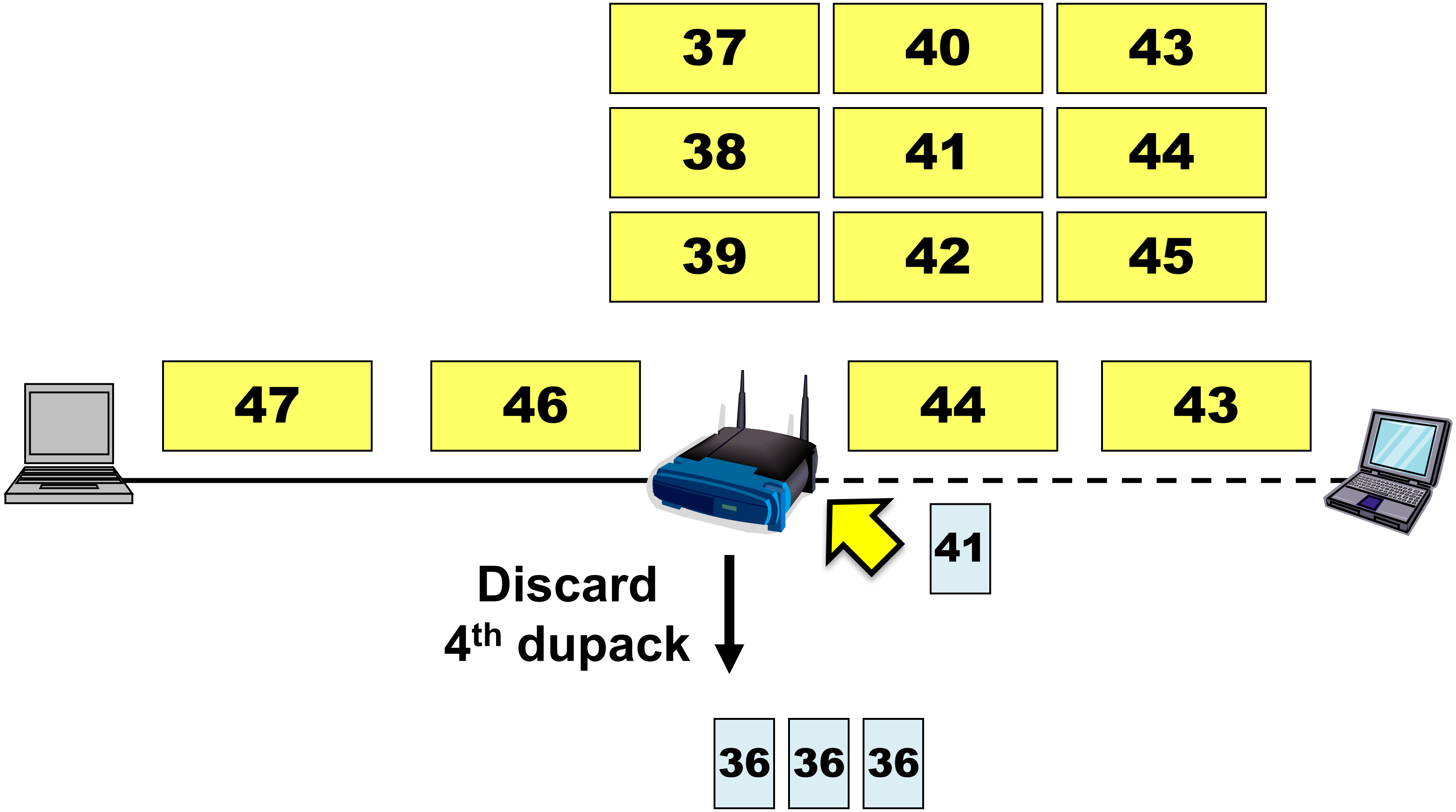
Discard
2nd dupack

TCP Snoop: Downlink example



Discard
3rd dupack

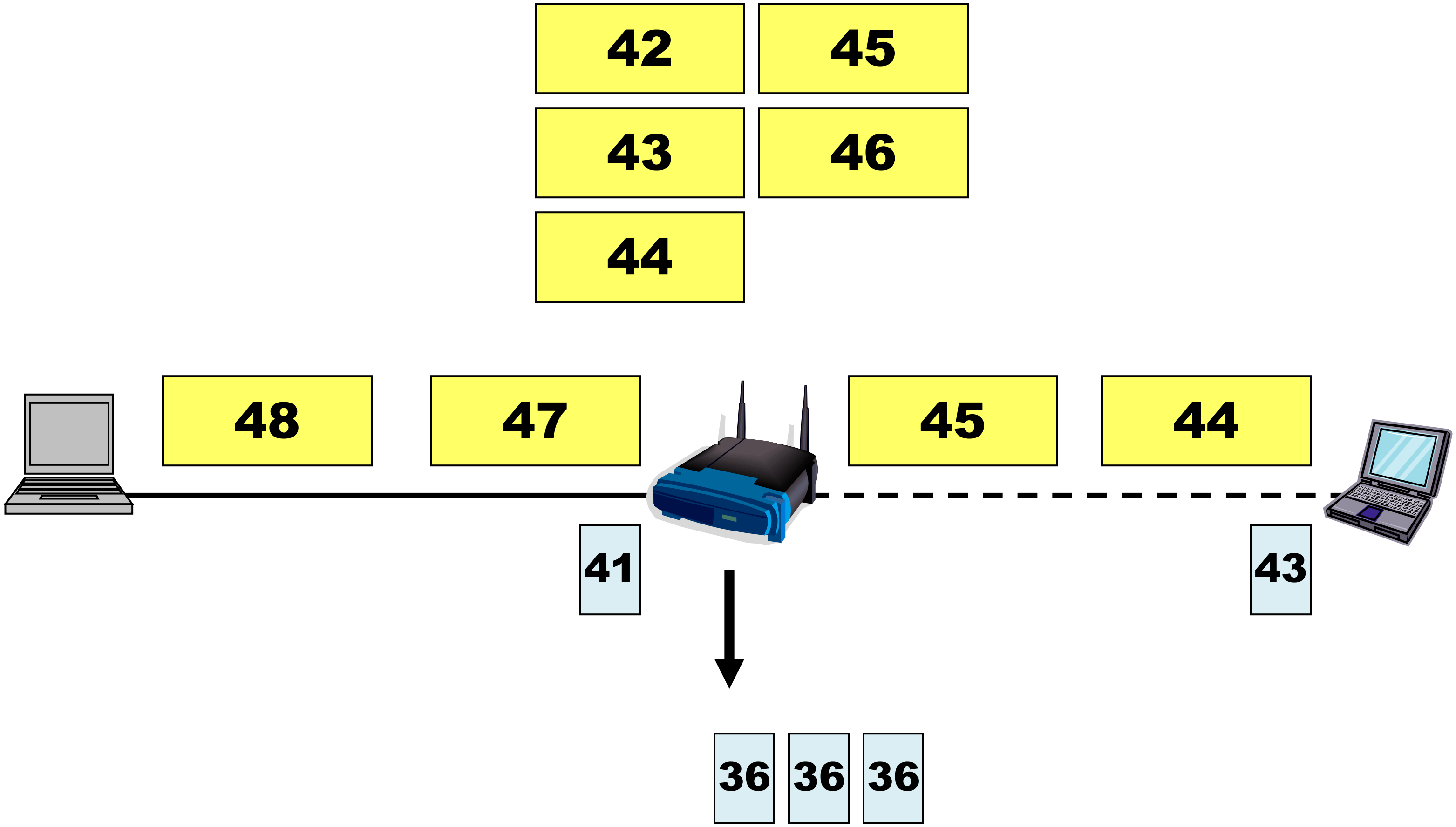
TCP Snoop: Downlink example



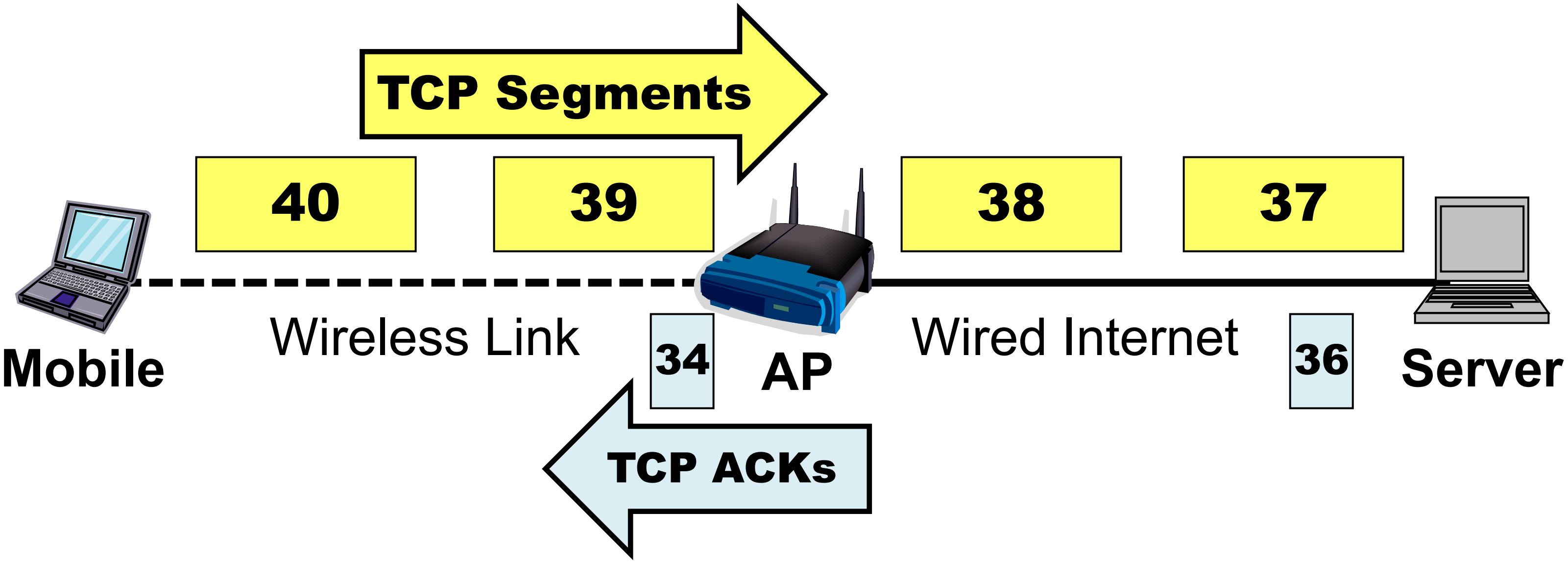
Discard
4th dupack

- TCP sender does not fast retransmit

TCP Snoop: Downlink example

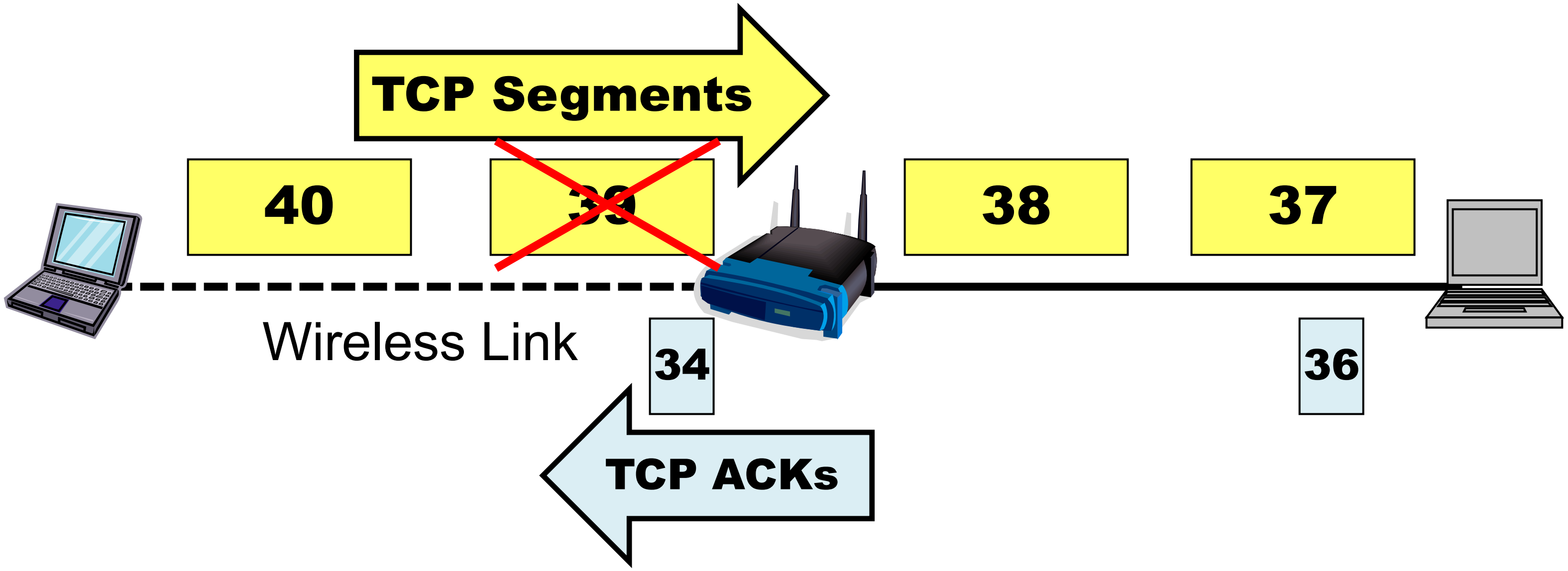


Uplink traffic case



- **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**

Negative ACKs: Recovering uplink loss


TCP Segments

40

39

38

37

Wireless Link

34

36

TCP ACKs

- **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)

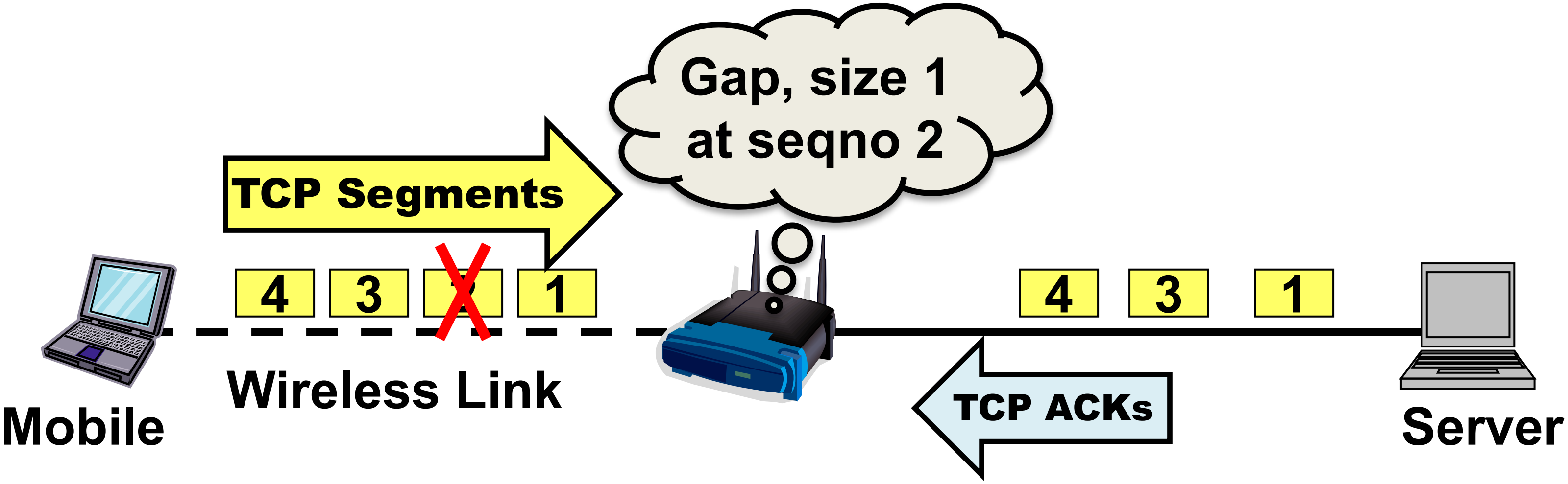
- Notify the TCP sender that a **wireless link** (not congestion) caused a certain packet loss

- 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

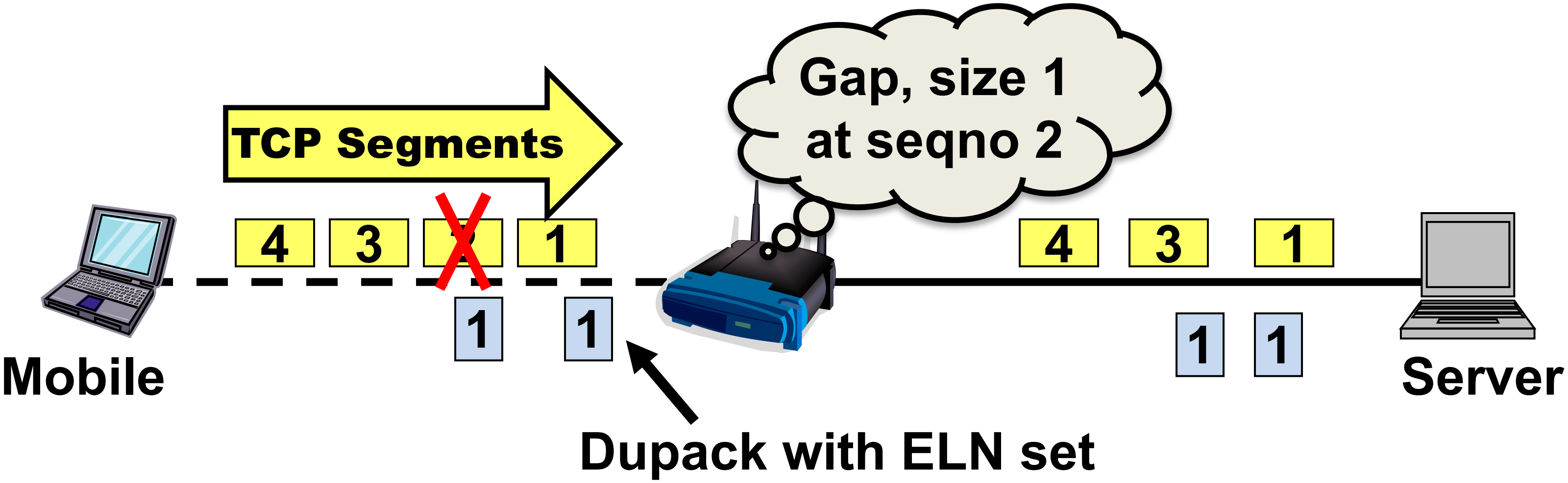
- AP keeps track of **gaps** in the TCP packet sequence received from the mobile sender



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



Thursday Topic:
Link Layer I: Time, Frequency, and Code Division

Friday Precept
Introduction to Lab 1