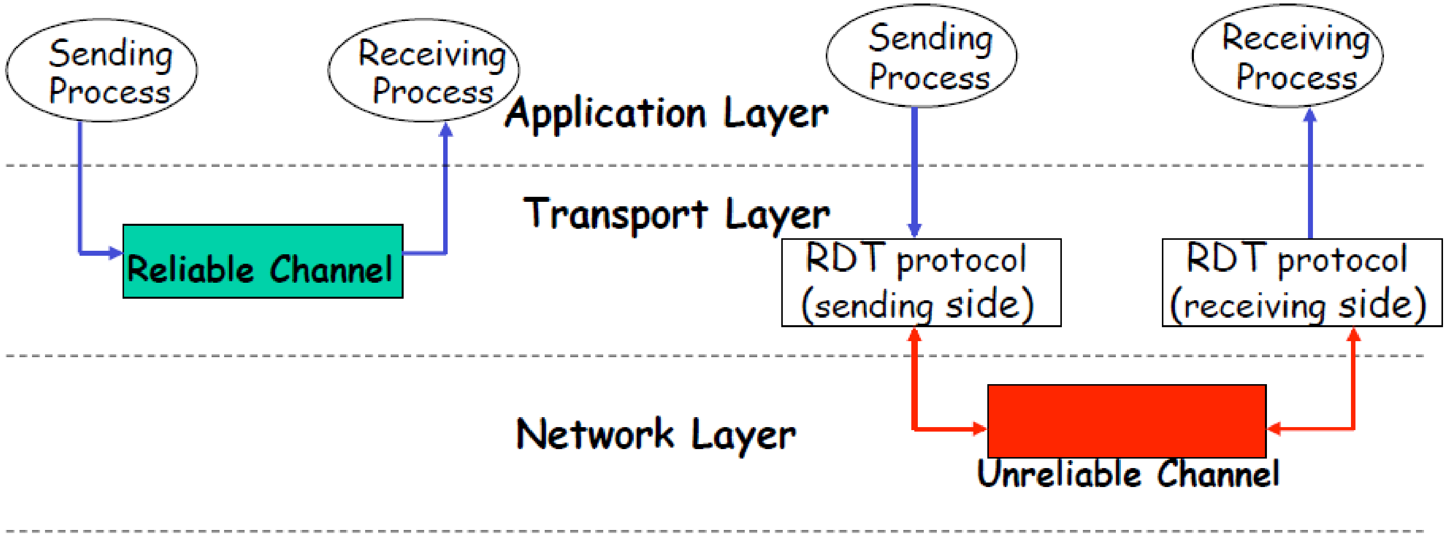
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Principles of Reliable Data Transfer

- Fundamentally important networking topic!
- Why do we need reliable data transfer protocol?



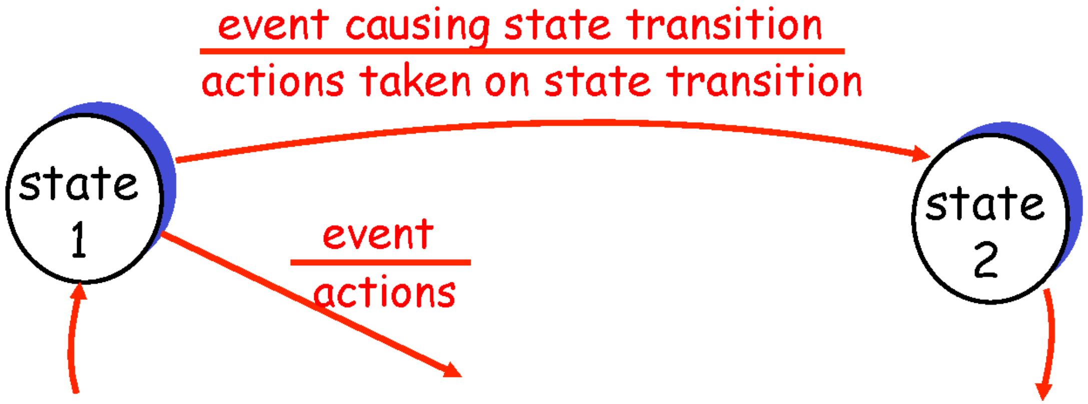
- What can happen over unreliable channel?
  - **Message error**
  - **Message loss**

Let's Build *simple* Reliable Data Transfer Protocol

We'll:

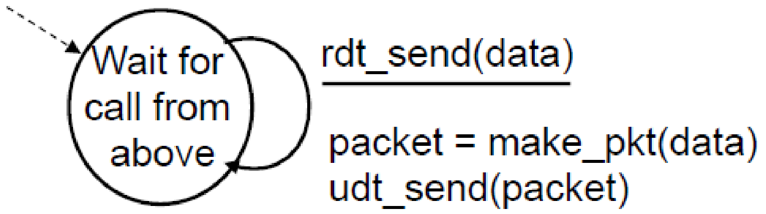
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only stop-and-wait protocol
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

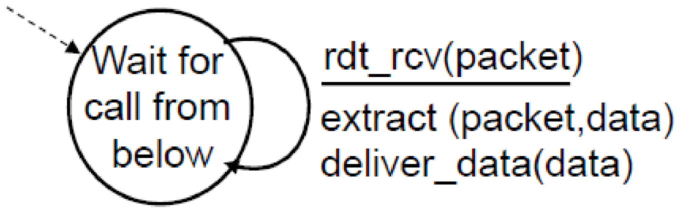


Rdt1.0: Data Transfer over a Perfect Channel

- underlying channel is perfectly reliable
  - no packet errors
  - no packet loss
- What mechanisms do we need for reliable transfer?
  - Nothing! Underlying channel is reliable!



sender



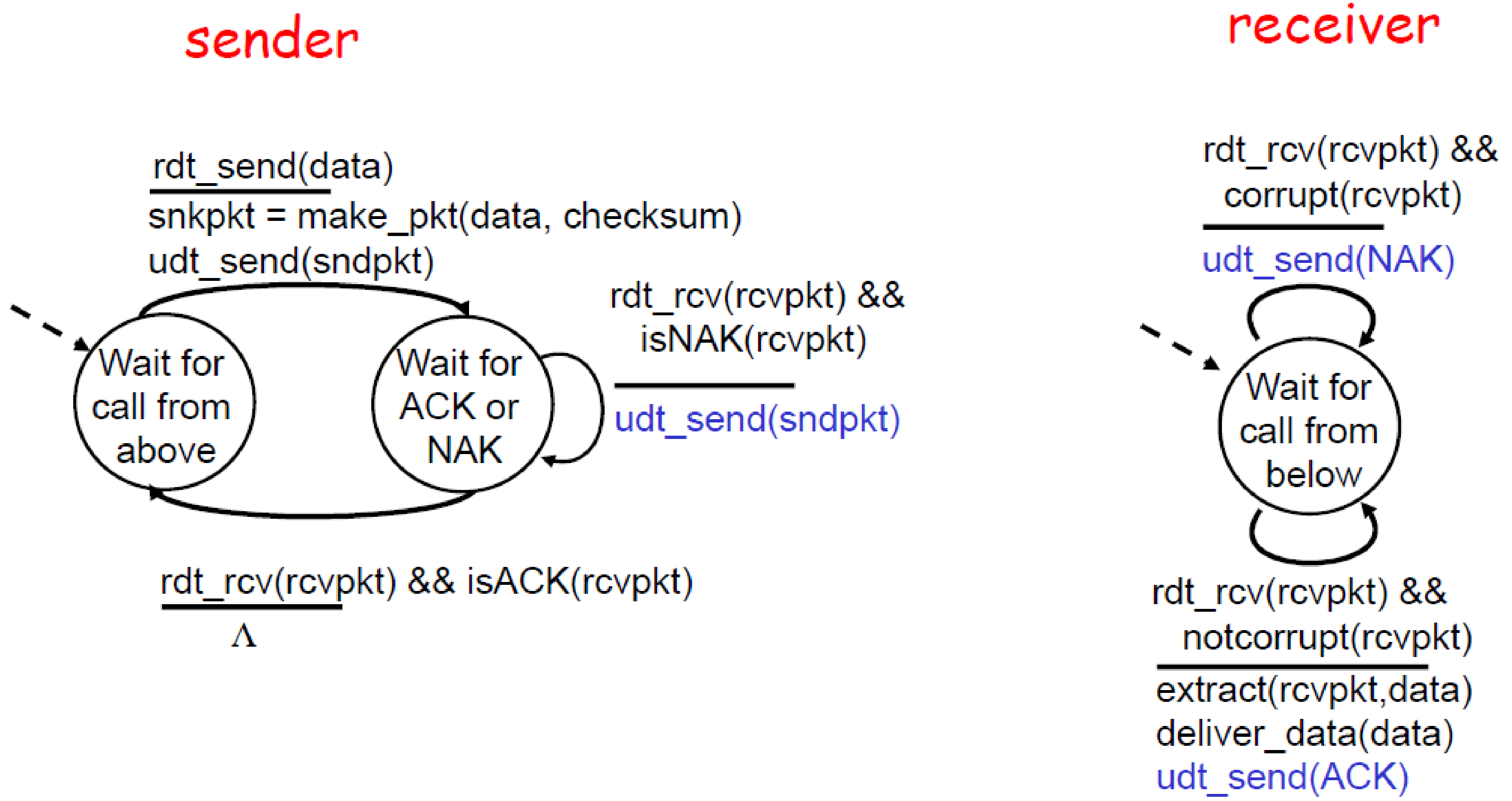
receiver

Rdt2.0: channel with packet errors (no loss!)

- What mechanisms do we need to deal with error?
  - Error detection
    - Add checksum bits
  - Feedback
    - *Acknowledgements (ACKs):* receiver explicitly tells sender that packet received correctly
    - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that packet had errors
  - Retransmission
    - sender retransmits packet on receipt of NAK

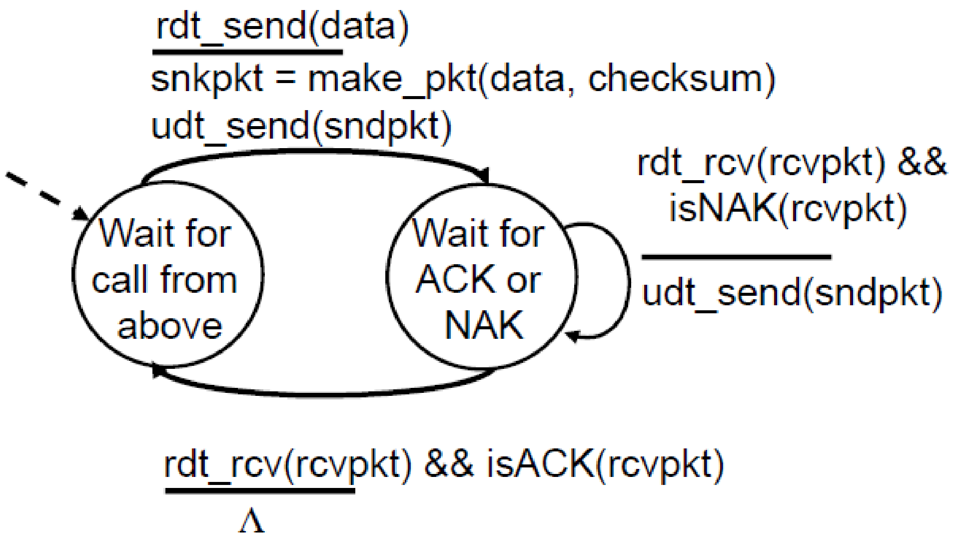
- So, we need the following mechanisms:
  - Error detection, Feedback (ACK/NACK), Retransmission

rdt2.0: FSM specification



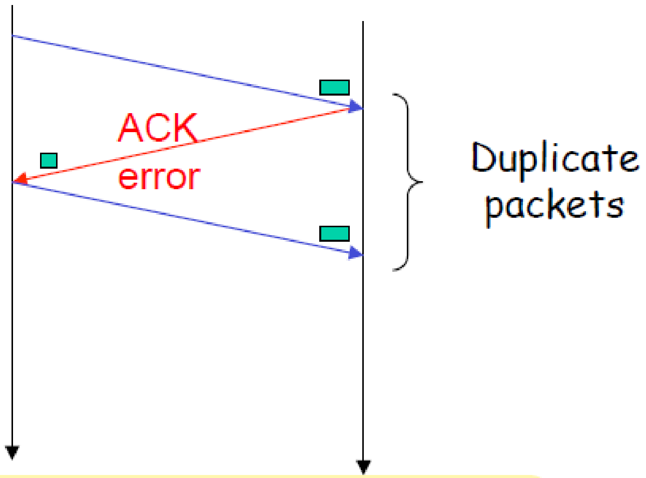
rdt2.0: Can this completely solve errors?

sender



What happens when ACK or NAK has errors?

Approach: resend the current data packet?

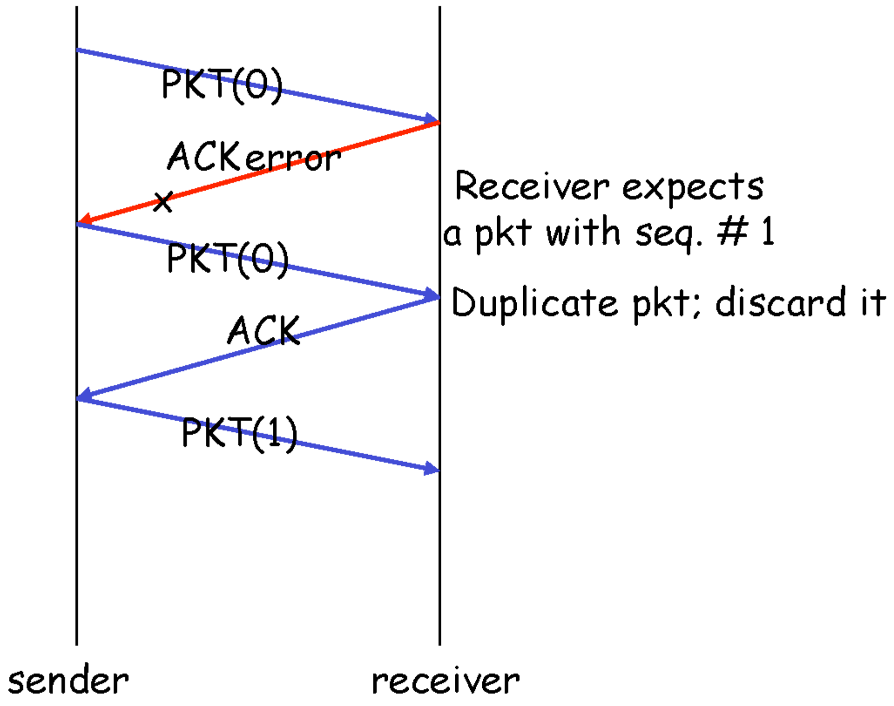


The received packet is new or duplicate?

Handling Duplicate Packets

- Sender adds *sequence number* to each packet
- Sender retransmits current packet if ACK/NAK garbled
- Receiver discards duplicate packet

rtd2.1: examples



rdt2.1: summary for packet error

- Mechanisms for channel with packet errors
  - Error detection, Feedback, Retransmission, Sequence#

Sender:

- seq # added to pkt
- must check if received ACK/NAK corrupted
- Retransmit on NAK or corrupted feedback

Receiver:

- must check if received packet is duplicate
- send NAK if received packet is corrupted
  - Send ACK otherwise

rdt2.2: a NAK-free protocol

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last correctly received packet
  - Receiver must *explicitly* include seq # of pkt being ACKed
- Duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

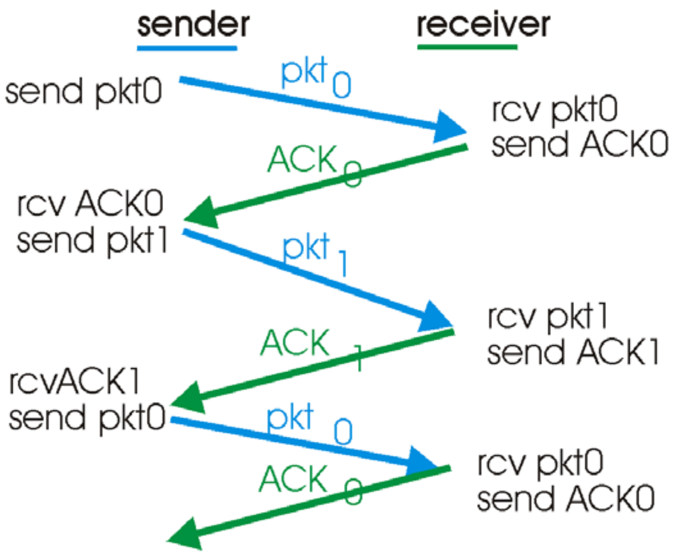
rdt3.0: channel with loss & packet errors

- What mechanisms do we need for packet loss?
  - Timer!

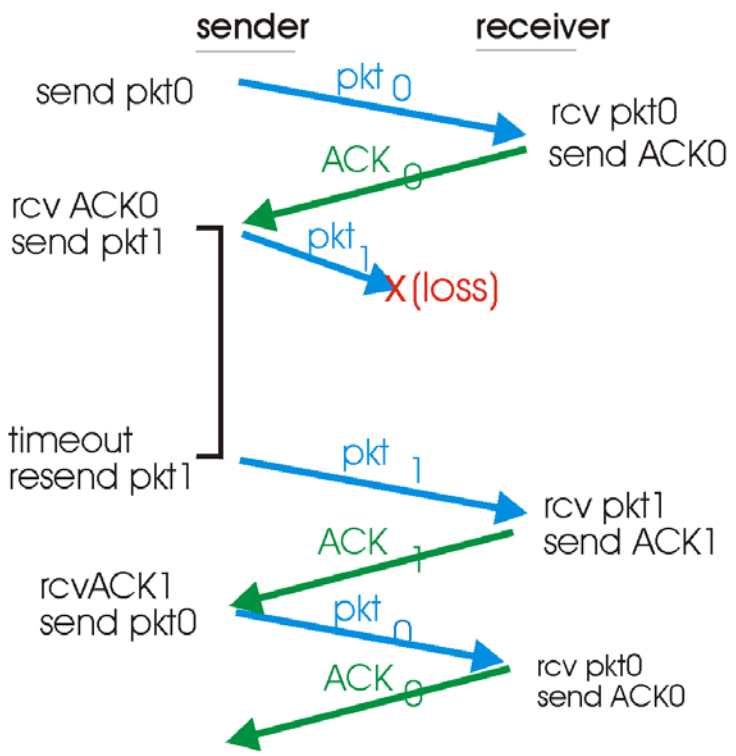
- Sender waits "reasonable" amount of time for ACK (a Time-Out)

- If packet (or ACK) is just delayed (not lost):
  - Retransmission will be duplicate, but use of seq. #'s already handles this

rdt3.0 in action

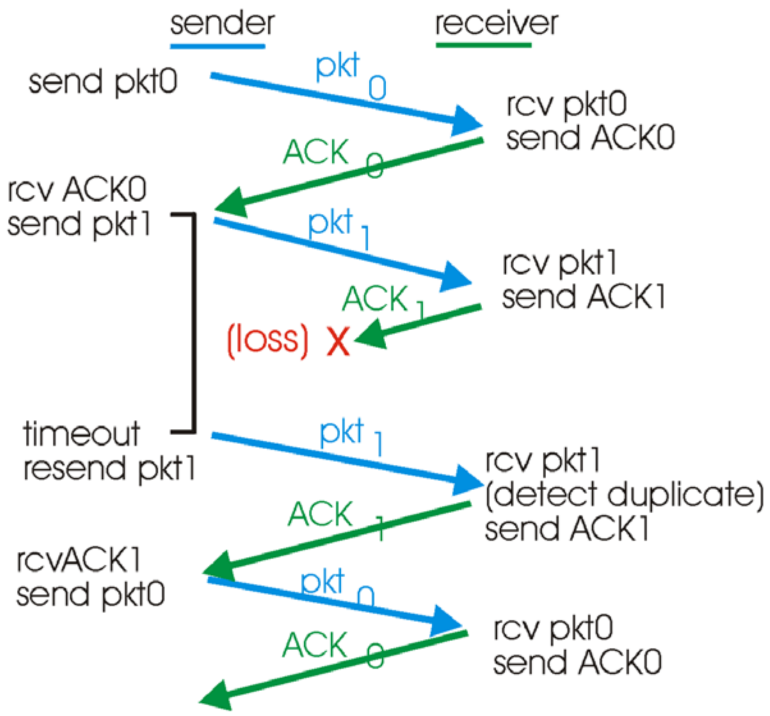


(a) operation with no loss

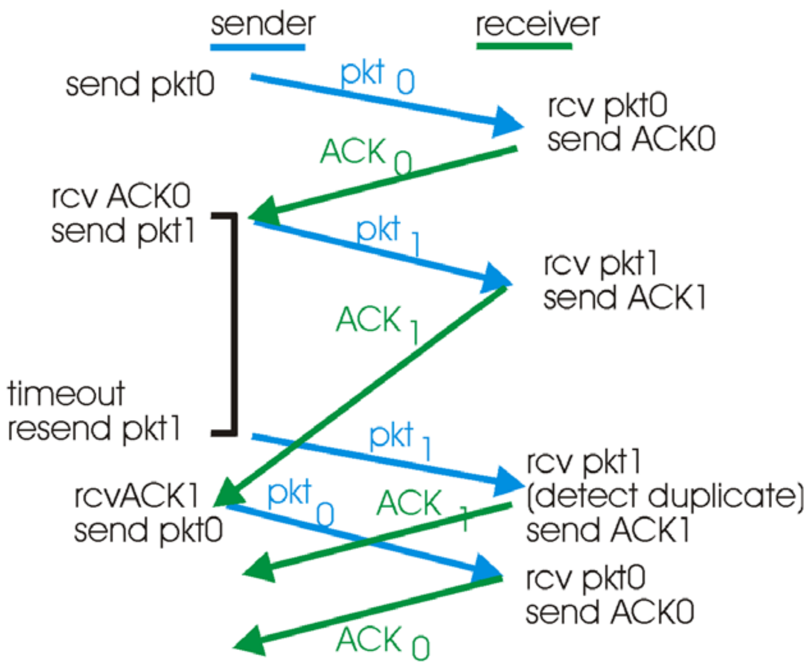


(b) lost packet

rdt3.0 in action



(c) lost ACK



(d) premature timeout

Recap: Principles of Reliable Data Transfer

- What can happen over unreliable channel?
  - Packet error, packet loss
- What mechanisms for **packet error**?
  - Error detection, feedback, retransmission, sequence#
- What mechanisms for **packet loss**?
  - Timeout!
- We built simple reliable data transfer protocol
  - Real-world protocol (e.g., TCP) is more complex, but *with same principles!*

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^{**}9 \; b/sec} = 8 \text{ microsec}$$

  - U_{sender}: utilization – fraction of time sender busy sending

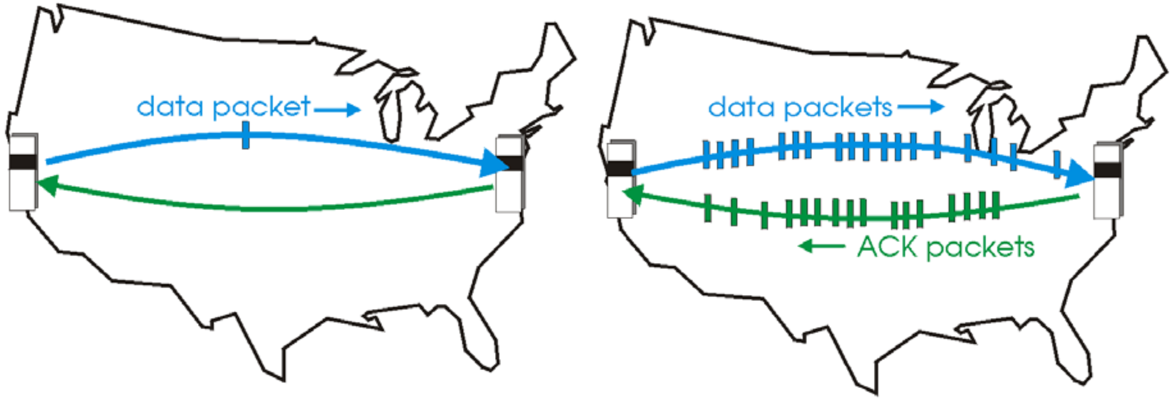
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver



(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat*