Transport Layer

Application / Transport Interface

Application requests service from transport layer

Application Layer	Prepare Transport service requirements Data for transport Local endpoint — node address + service port Remote endpoint — node address + service port
Transport Layer	Receives Application layer PDU — data for transport Interface Control Information (ICI) Service requirements Local + remote endpoints

Transport Layer Services

Connection management

- Maintain separate sessions between various user applications
- Label sessions with client / server port numbers
- Provide reliability services according to connection type

Multiplexing

- Divide outgoing data stream into segments
- Combine segments from multiple sessions into single output

Demultiplexing

- Separate incoming segments by sessions
- Combine incoming segments for each session into user data stream

Transport Layer Connections

Reliable transport (TCP)

- Connection-oriented
  - TCP connection established before data transfer
- Error-free delivery
  - Data delivered
    - In original order
    - No errors, duplications, omissions
- Flow control
  - Control sender rate to prevent buffer overflow in receiver
- Congestion control
  - Control sender rate to prevent buffer overflow in network

Unreliable transport (UDP)

- Connectionless
- Lower overhead ⇒ faster but no guarantees
- Segments with errors discarded with no warning to application

Source / Destination Ports

Client

Opens socket to send requests

Client / OS binds port number to socket

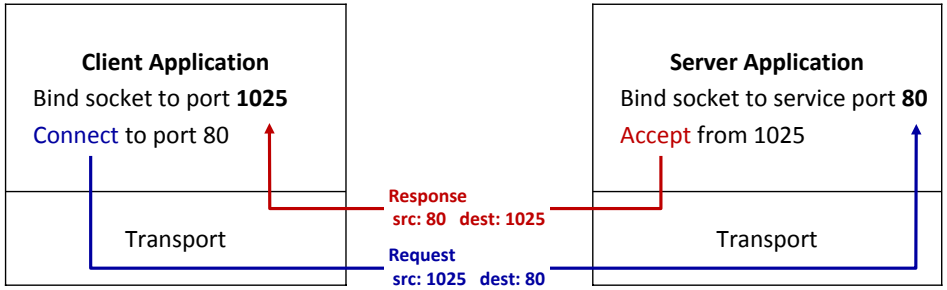
1024 ≤ client port ≤ 65,535 identifies client application

Server

Opens listen socket mapped to accept sockets for requests

Binds well-known port to service socket

0 ≤ well-known port ≤ 1023 identifies service application



Multiplexing / Demultiplexing

Applications send / receive data on sockets

Multiple sockets ⇒ multiple conversations

Transport layer segment

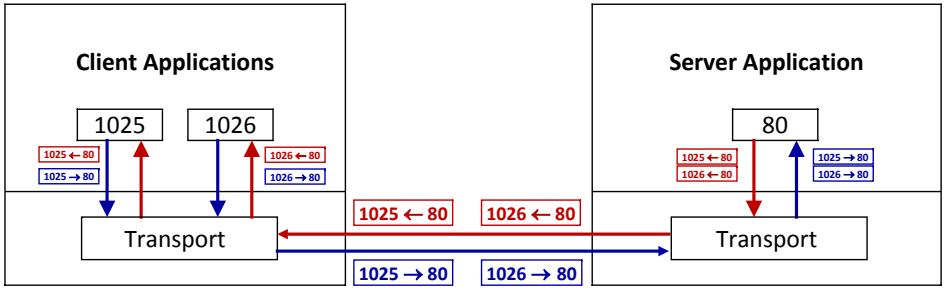
Transport header + application data (PDU)

TCP / UDP headers carry source + destination ports

Multiplexing / demultiplexing

Segments transmitted on same infrastructure

Sorted by destination port at destination



User Datagram Protocol (UDP)

Internet unreliable transport protocol

Defined in RFC 768

Used when low delay / jitter more important than error control

Streaming multimedia, multiplayer games, …

UDP segment

header	application data

UDP header

32 bits

source port	destination port
length	checksum

Length

Number of bytes in datagram < 2^{16} = 65,536

Maximum length = 64 KB – lengths of all headers

UDP / TCP Checksum

Calculation at source

Break UDP segment into sequence of 16-bit words

Add IP pseudo-header (IP src/dest addresses, protocol, length)

Pad with zeros if necessary

Add all 16 bit words (not counting checksum field)

Add carry-out to 0-order bit (rotate)

Perform 1's complement

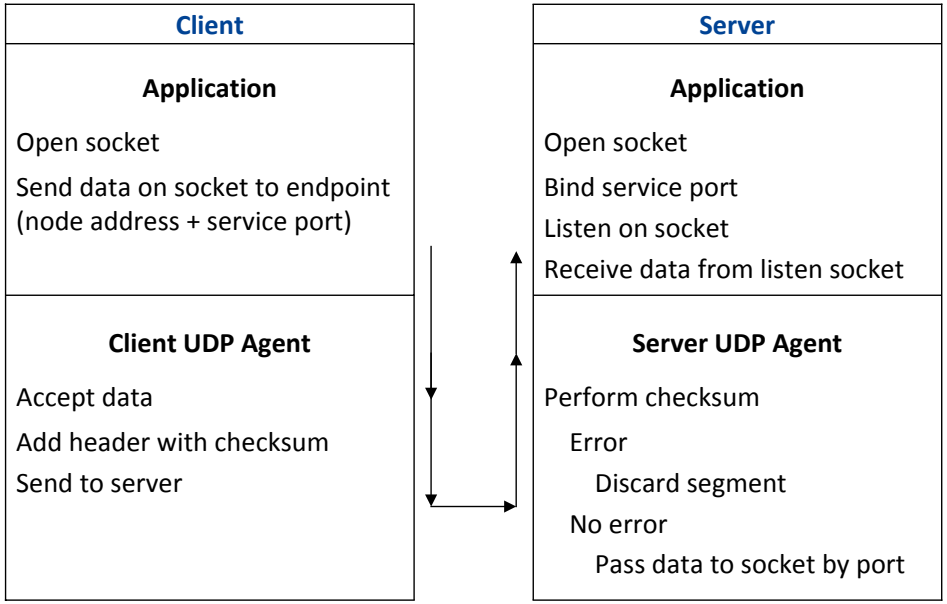
Calculation at destination

Repeat calculation and compare

Example

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
rotate	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
																1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Using UDP



If required — add reliability features at client / server application level

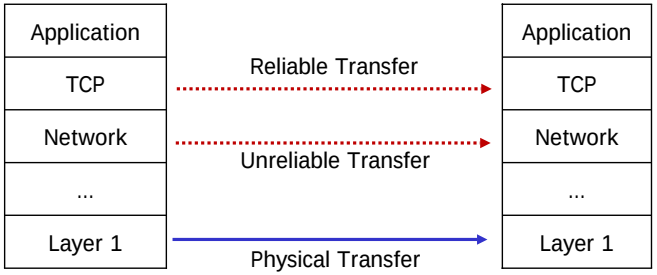
Transport Control Protocol (TCP)

Reliable transport over unreliable layers

Network + infrastructure layers can introduce errors

- Bit errors (1 ↔ 0)
- Lost packets
- Duplicate packets
- Out-of-order packets

TCP detects and corrects errors



Basic Theory of Reliable Data Transfer (RDT)

Bottom-up approach

Version 1.0 for error-free channel

No error checking necessary

Version 2.0

Handles bit errors

Version 2.1

Bug fix of version 2.0

Version 2.2

Simplification of version 2.1

Version 3.0

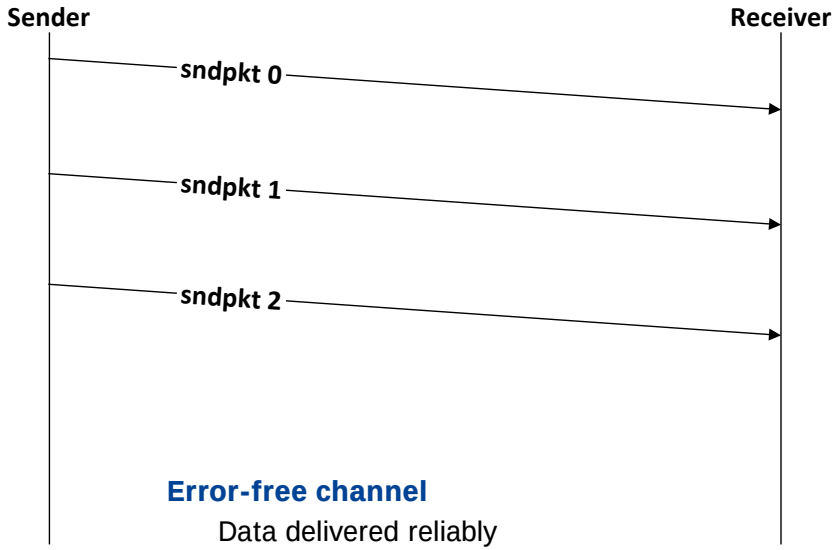
Handles missing packets

Version 4.0

Improved performance

Theory of Reliable Data Transfer (RDT)

Version 1.0 on error-free channel



Error-free channel

Data delivered reliably

Handling Bit Errors

Error detection

Sender adds checksum to packet header

Receiver

- Re-computes checksum
- Compares with checksum in header

Corrupt packets (packets with errors) discarded

Error control

Receiver provides sender with feedback about received data

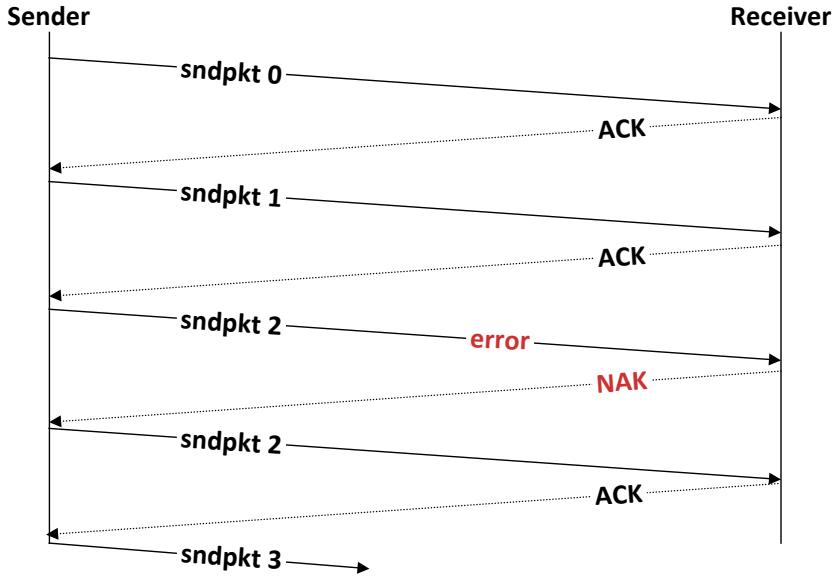
ACK	Acknowledgement	Data received without bit errors
NAK	Negative acknowledgement (reject)	Data received with bit errors

Automatic Repeat Request (ARQ)

Sender re-transmits on NAK

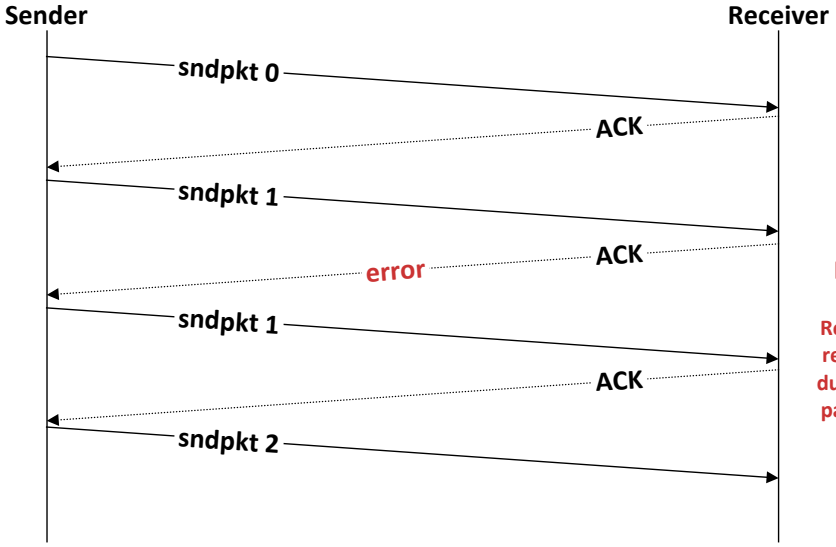
Theory of Reliable Data Transfer (RDT)

Version 2.0 on channel with bit errors



Theory of Reliable Data Transfer (RDT)

Version 2.0 on channel with bit errors



Bug in Version 2.0

Corrupt ACK / NAK packet

Option 1

- Sender interprets corrupt ACK / NAK = ACK
- Receiver misses data packet

Option 2

- Sender interprets corrupt ACK / NAK = NAK
- Sender retransmits packet
- Receiver may receive duplicate packet

Bug fix

Label packet with sequence number (SEQ)

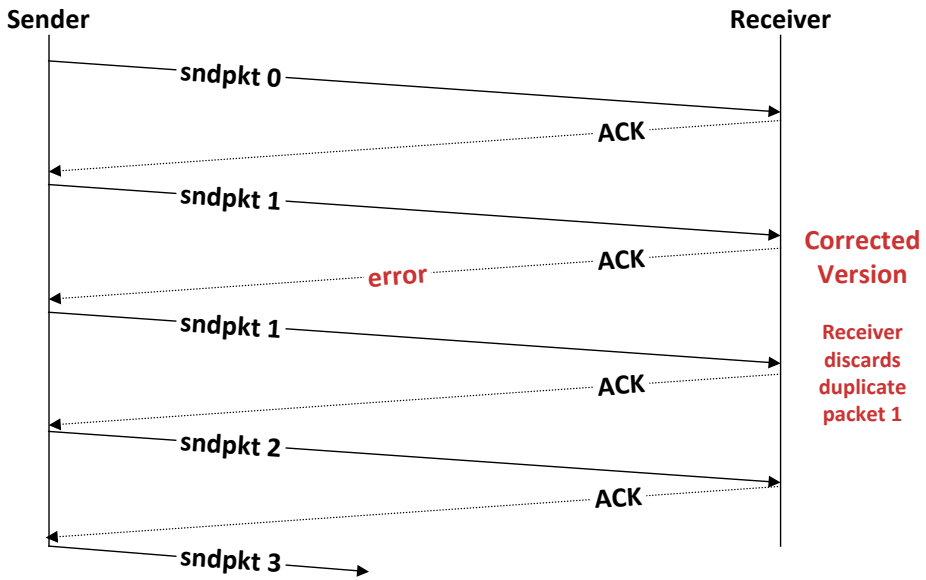
- SEQ = packet_number mod 2 = 0 or 1 (sufficient for stop and wait)

Sender follows option 2

Receiver gets duplicate packet

- Re-transmits ACK
- Discards duplicate packet

Theory of Reliable Data Transfer (RDT)
Version 2.1 with SEQ



Complication in Version 2.1

Version 2.1 — SEQ + ACK + NAK

Sender
- ACK or NAK = response to last packet
- Corrupt ACK or NAK = NAK
- NAK ⇒ re-transmit packet

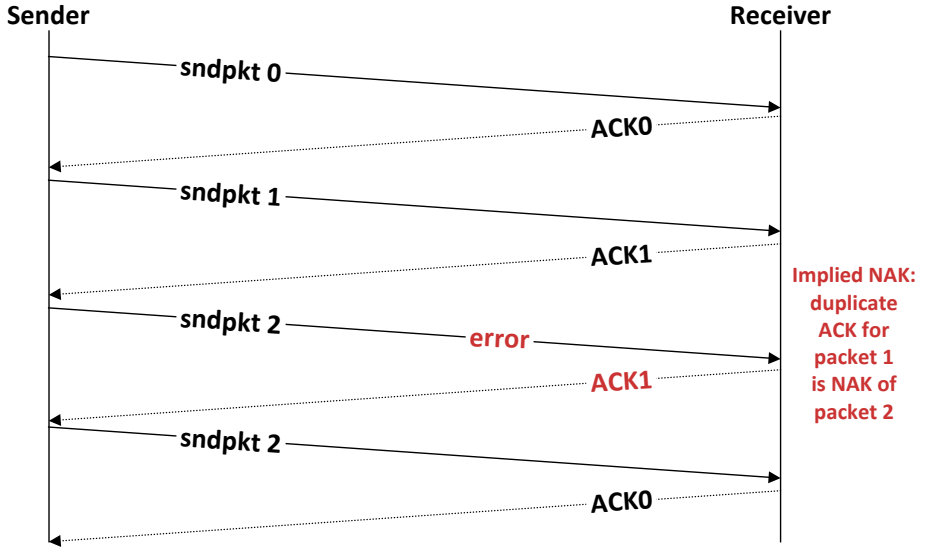
Receiver
- ACK or NAK for each packet
- Corrupt packet ⇒ re-transmit packet
- Duplicate SEQ ⇒ re-transmit ACK but discard packet

Version 2.2 — SEQ + ACK without transmitted NAK

Sender
- ACK + SEQ = response to packet SEQ
- Corrupt ACK || ACK to previous SEQ = implied NAK
- Implied NAK ⇒ re-transmit packet

Receiver
- ACK + SEQ for each packet
- Corrupt packet ⇒ re-transmit ACK for **previous SEQ**
- Duplicate SEQ ⇒ re-transmit ACK but discard packet

Theory of Reliable Data Transfer (RDT)
Version 2.2 with SEQ + ACK without NAK



Lost Packets

Packet loss

Lost data packet or ACK

Discarded by intermediate system or lower network layer

Buffer overflow

At receiver — no error or indication of lost packet

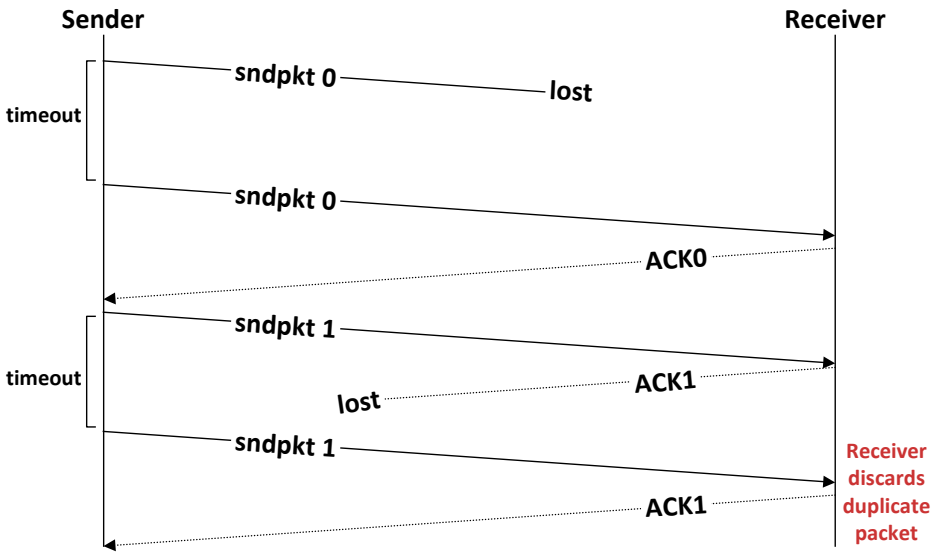
Handling lost packets

Sender sets timeout counter

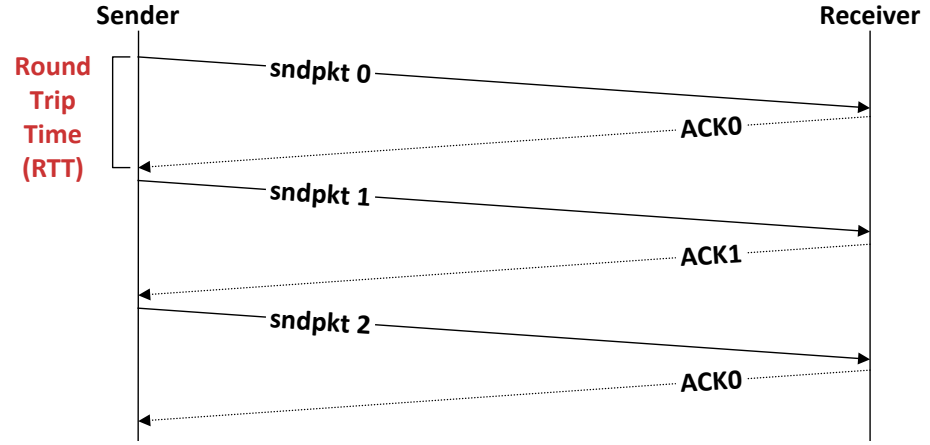
Timeout || corrupt ACK || ACK to previous SEQ = implied NAK

Duplicates handled by SEQ

Theory of Reliable Data Transfer (RDT)
Version 3.0 with SEQ + ACK + Timeout



Performance Problem with Stop-and-Wait
Wait too long



$$T_T = \text{transmission delay} \approx \frac{64\ \text{KB} \times 8\ \text{bits/B}}{10\ \text{Mbps}} = 5\ \text{ms}$$

Typical RTT > 30 ms

$$\text{utilization} = \frac{T_T}{T_T + \text{network latency}} = \frac{T_T}{T_T + \text{RTT}} \approx \frac{5\ \text{ms}}{5\ \text{ms} + 30\ \text{ms}} = 0.14$$

Theory of Reliable Data Transfer (RDT)
Improving performance

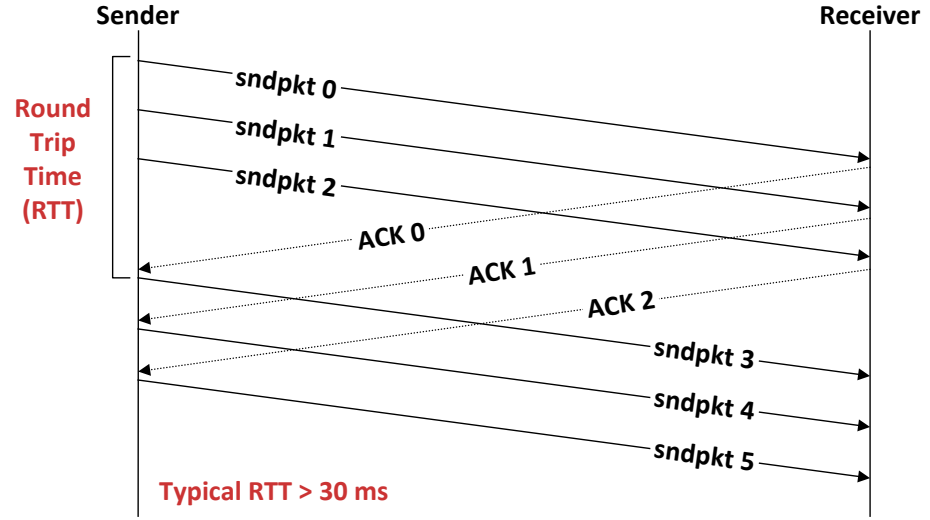
Sender transaction
- Send packet
- Receive ACK

Stop-and-wait protocol
- Send packet → wait → receive ACK
- Finish transaction n → begin transaction n + 1
- Transmitter idle while waiting for ACK

Pipelined protocol
- Begin new transaction before previous finishes
- Reduce transmitter idle time

Source window
- N = window size
- Transmit N packets before stop-and-wait
- N packets in process ("in flight" or "in pipeline") at any time

Theory of Reliable Data Transfer (RDT)
Pipelining with window size = 3



Typical RTT > 30 ms

$$\text{utilization} = \frac{N \times T_T}{T_T + \text{RTT}} \approx \frac{3 \times 5\ \text{ms}}{5\ \text{ms} + 30\ \text{ms}} = 0.43$$

Theory of Reliable Data Transfer (RDT)
Pipelined protocol — error handling method 1

Selective repeat

Sender

- Buffers packet until ACK
- Timeout counter for each packet
- Timeout ⇒ sender re-transmits "unACKed" packet

Receiver

- ACKs received packets
- Received in order — passed to application
- Received out-of-order — held in buffer

Window size = N

"unACKed" packets < N

Send N packets without ACK ⇒ stop sending

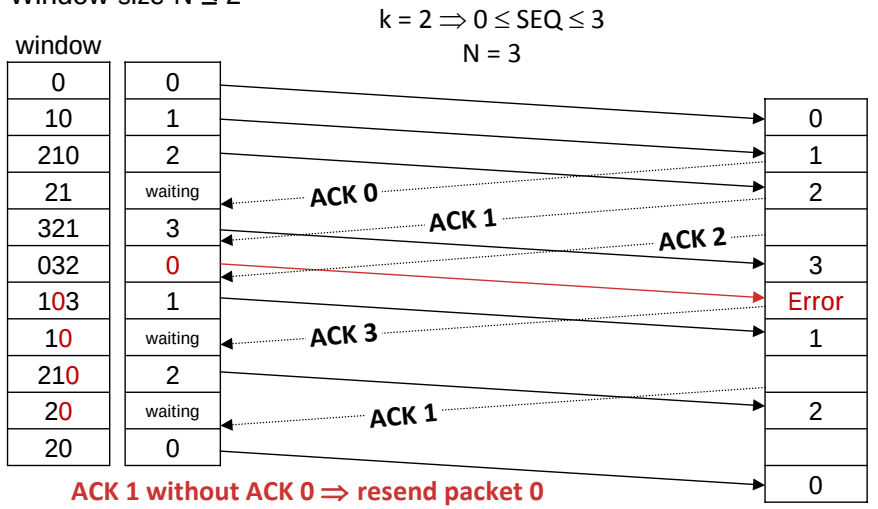
N < receiver buffer size

Theory of Reliable Data Transfer (RDT)
Selective repeat

Sequence number (SEQ)

k bit SEQ ⇒ $0 \le SEQ \le 2^k - 1$ (modulo 2^k)

Window size $N \le 2^k$

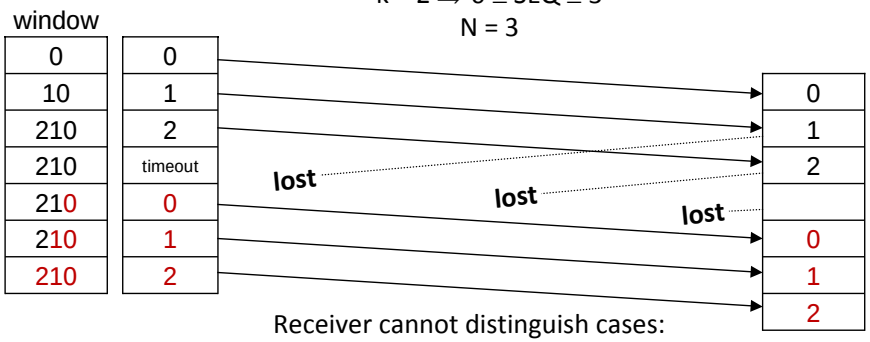


Theory of Reliable Data Transfer (RDT)
Bug in selective repeat

Sequence number (SEQ)

k bits ⇒ $0 \le SEQ \le 2^k - 1$ (modulo 2^k)

Window size $N \le 2^k$



Receiver cannot distinguish cases:

1. Re-transmission of duplicates
2. New packets

Solution — long SEQ (≥ 32 bits)

Theory of Reliable Data Transfer (RDT)
Pipelined protocol — error handling method 2

Go Back N (GBN)

Sender

- Buffers packet until ACK
- Timeout counter for oldest "unACKed" packet
- Timeout ⇒ sender re-transmits all "unACKed" packets from buffer

Receiver sends **CUMULATIVE ACK**

- ACK for last packet
  - Implies ACK for all previous packets
- No buffer for out-of-order packets
- Error or missing packet ⇒ no ACK packet or any subsequent packet

Window size = N

"unACKed" packets < N

Send N packets without ACK ⇒ stop sending

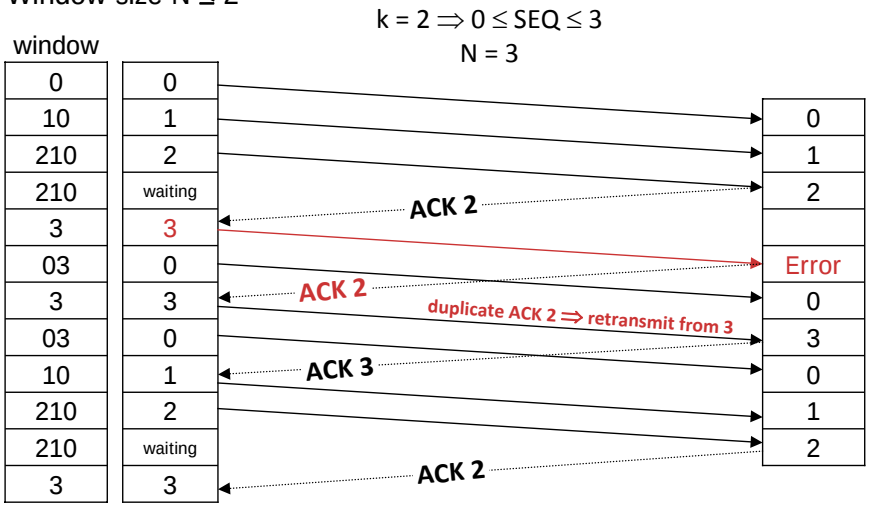
N < transmit buffer size

Theory of Reliable Data Transfer (RDT)
Version 4.0 — Go Back N

Sequence number (SEQ)

k bits ⇒ $0 \le SEQ \le 2^k - 1$ (modulo 2^k)

Window size $N \le 2^k$



TCP as Reliable Transport Protocol

Connection-oriented

Set up connection before data transfer

Maximum Segment Size (MSS)

64 KB (including IP header)

Error detection

Checksum as in UDP

Error control

ARQ with ACK + SEQ + timeout

No corrupt, missing, duplicate, or out-of-order data

Piggybacking — send ACK within data segment

Pipelining

Variable window size at sender and receiver

GBN cumulative ACK with optional selective repeat

Flow /congestion control

Dynamic window size ⇒ control of sender utilization

TCP Header

32 bits

source port			destination port
sequence number (SEQ)			
acknowledgement number (ACK)			
HLEN	not used	flags	window size
checksum			urgent pointer
Options			

HLEN (data offset)	4 bits	Length of TCP header in 32-bit words
Reserved	3 bits	Not used
Flags	9 bits	Control bits
Window size	16 bits	Number of bytes receiver can receive now
Urgent pointer	16 bits	Offset from SEQ points to last urgent data byte
Options	10 – 320 bits	Options fields + padding for multiple of 32 bits

TCP Header
Flags

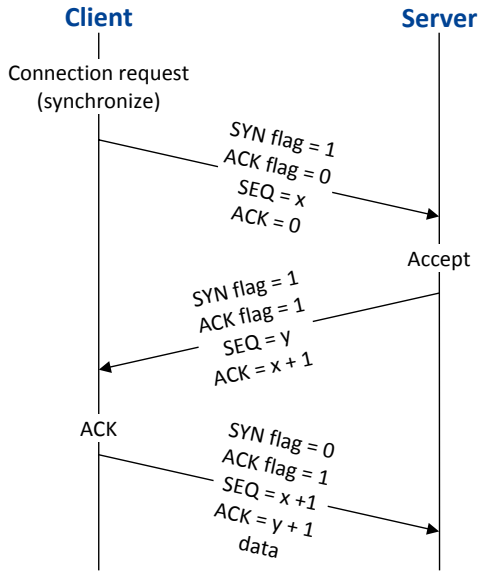
NS	ECN-nonce concealment protection
CWR	Congestion Window Reduced (CWR) flag Sender indicates receiving segment with ECE flag set
ECE	ECN-Echo If SYN = 1 peer is ECN capable If SYN = 0 packet with Congestion Experienced flag in IP header received during normal transmission
URG	Urgent pointer field valid
ACK	**Acknowledgment field valid**
PSH	Push buffered data to receiving application
RST	**Reset connection**
SYN	**Synchronize sequence numbers**
FIN	**No more data from sender**

TCP Connection Set-up
Three-way handshake

Client SYN segment
- SYN flag = 1
- SEQ = random number x
- No data

Server SYN-ACK segment
- SYN flag = ACK flag = 1
- SEQ = random number y
- ACK = x + 1
- No data

Client ACK segment
- SYN flag = 0
- ACK flag = 1
- SEQ = random number x + 1
- ACK = y + 1
- May contain data

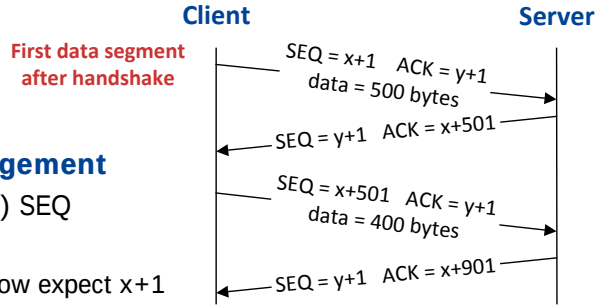


TCP SEQ + ACK

Initial sequence number (ISN)
- Random SEQ in 3-way handshake
  - Prevents counterfeit segments
- At end of handshake SEQ = ISN + 1

SEQ — byte sequencing
SEQ = previous SEQ + length(data bytes in previous segment)
= ISN + 1 + data bytes sent in all **previous** segments

ACK — byte acknowledgement
ACK = next (expected) SEQ
ACK = x + 1
Acknowledge x — now expect x+1



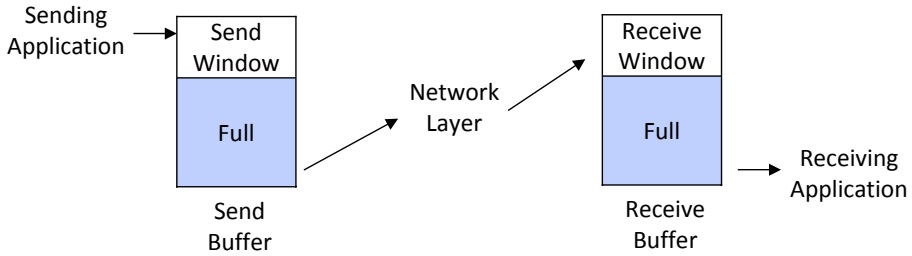
Send and Receive Windows

Send buffer
- Holds sent segments until ACKed
- On timeout resend segments from send buffer

Send window
- Available space in send buffer

Receive buffer
- Holds received segments until requested by application

Send window
- Available space in send buffer



Basic TCP Sender Behavior

Start
SEQ = SendBase = ISN + 1

Receive data from application
- Prepare segment with SEQ
- Send segment when unACKed bytes < SendWindow
- SEQ = SEQ + length(segment data)
- Start timer

Receive ACK = y
- y > SendBase ⇒ SendBase = y
- unACKed bytes > 0 ⇒ restart timer

Timeout
- Resend segments starting from SendBase
- Restart timer

Typical Control Program

```
default_inits ;
while ( true ) {                /* main loop */
    if ( condition_1 ) {
      handle_condition_1 ;
    }

    if ( condition_2 ) {
      handle_condition_1 ;
    }

    if ( condition_3 ) {
      handle_condition_1 ;
    }

    default_updates ;
}
```


Simplified TCP Sender — 1

```
// initialize
    SEQ = ISN + 1
    SendBase = ISN + 1              // last byte ACKed
    InFlight = 0                    // unACKed bytes sent
    RTO                             // timeout interval
    SendWindow, RecvWindow          // send & receive windows

// main loop
    if (new data from application)
        Prepare data segment
            sequence number = SEQ
            SEQ = SEQ + length(data)
        if InFlight < min{SendWindow,RecvWindow}
            Pass segment to IP
            InFlight = InFlight + length(data)
            if !(timer running) timer = RTO
```


Simplified TCP Sender — 2

```
    if (receive ACK = y)
        stop timer
        if (y > SendBase)
            newACKs = y – SendBase        // bytes ACKed
            SendBase = y
            InFlight = InFlight – newACKs
            if (InFlight > 0) timer = RTO

    if (timeout)
            SEQ = SendBase = min{unACKed SEQ} and resend
            timer = RTO
```


Basic TCP Receiver Behavior

Start

Expected = ISN + 1 // sender ISN

Receive error-free segment with SEQ < Expected

Resend ACK = Expected

Ignore packet

Receive segment with error || SEQ > Expected

Resend ACK = Expected (implied NACK)

Ignore packet

Receive error-free segment with SEQ = Expected

Expected = Expected + length(segment data)

Buffer ACK

Send ACK = Expected if

- Delayed ACK timer > maximum
- Buffered ACKs > maximum
- ACK following implied NACK

Simplified TCP Receiver — 1

```
// initialize
    Set RecvWindow = receiver buffer size
    expected = Sender ISN + 1
    ack_buffer = 0          // received unACKed segments
    ack_max                 // delayed ACK trigger
    ack_delay = 250 msec    // local policy: < 500 msec
    Start ACK delay timer = ack_delay

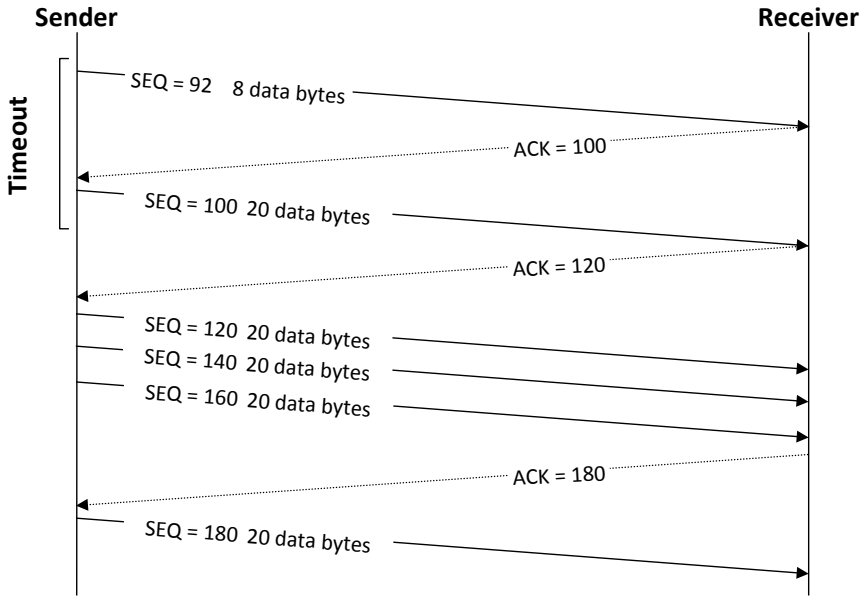
// main loop
if (ACK delay timer = 0 && ack_buffer > 0)
    Send ACK = expected with updated RecvWindow
    ACK delay timer = ack_delay
    ack_buffer = 0
```


Simplified TCP Receiver — 2

```
if (receive SEQ = x)                    // new packet arrives
    if (x = expected && error-free)
        expected = expected + length(data)
        if (NACK = 1)
            Send ACK = expected with updated RecvWindow
            ACK delay timer = ack_delay
            ack_buffer = 0
            NACK = 0
        else if (ack_buffer < ack_max)
            nextACK = expected
            ack_buffer++
        else if (ack_buffer = ack_max)
            Send ACK = expected with updated RecvWindow
            ACK delay timer = ack_delay
            ack_buffer = 0
    else
        Send ACK = expected with updated RecvWindow
        ACK delay timer = ack_delay
        NACK = 1
```

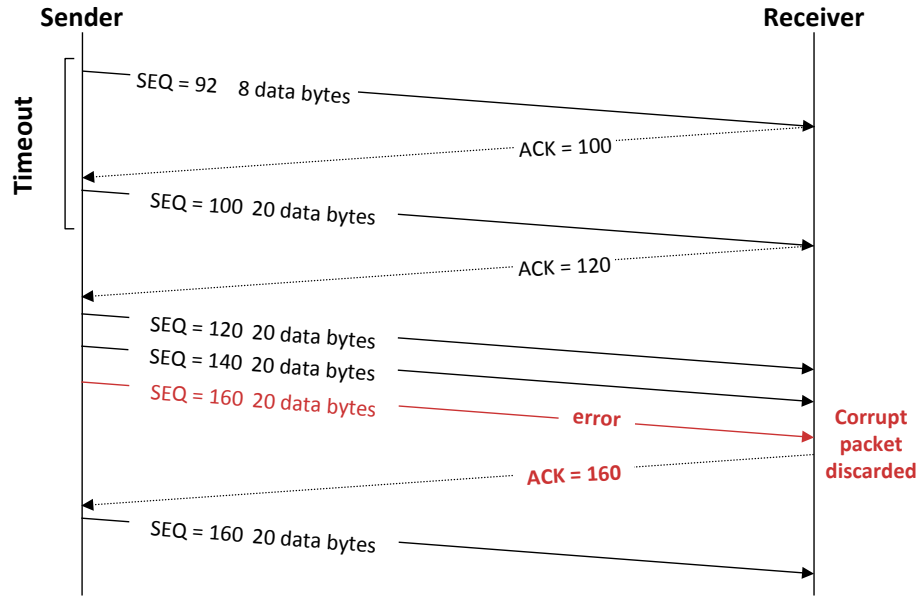

TCP SEQ + ACK

No errors

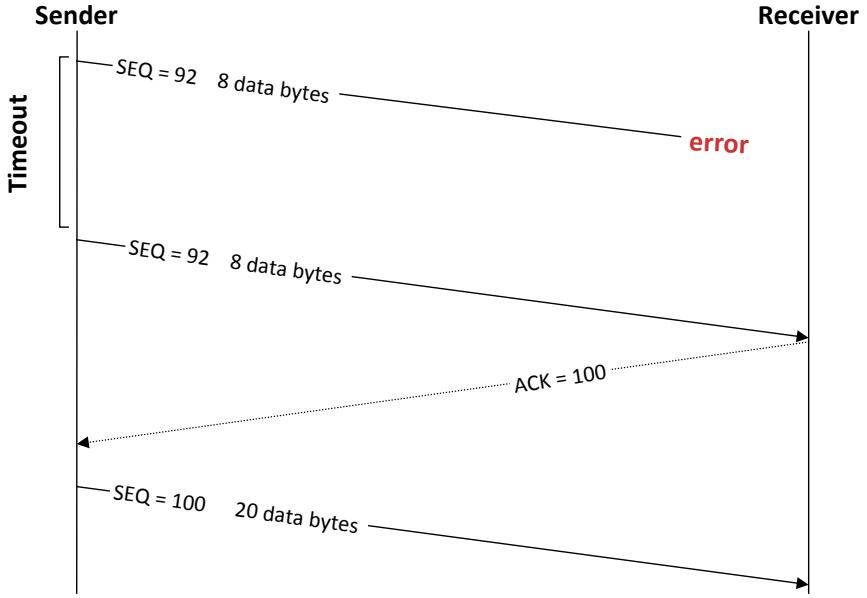


TCP SEQ + ACK

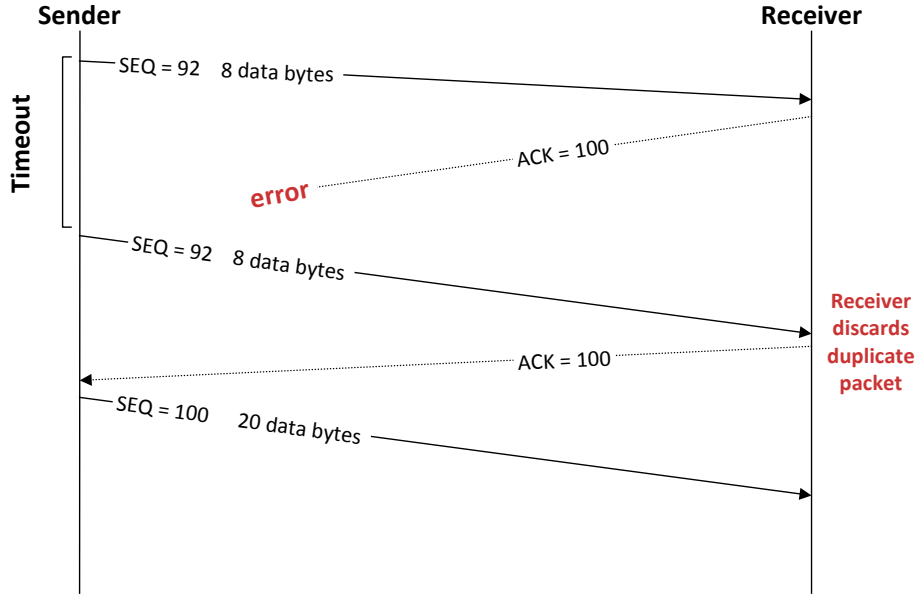
Bit errors



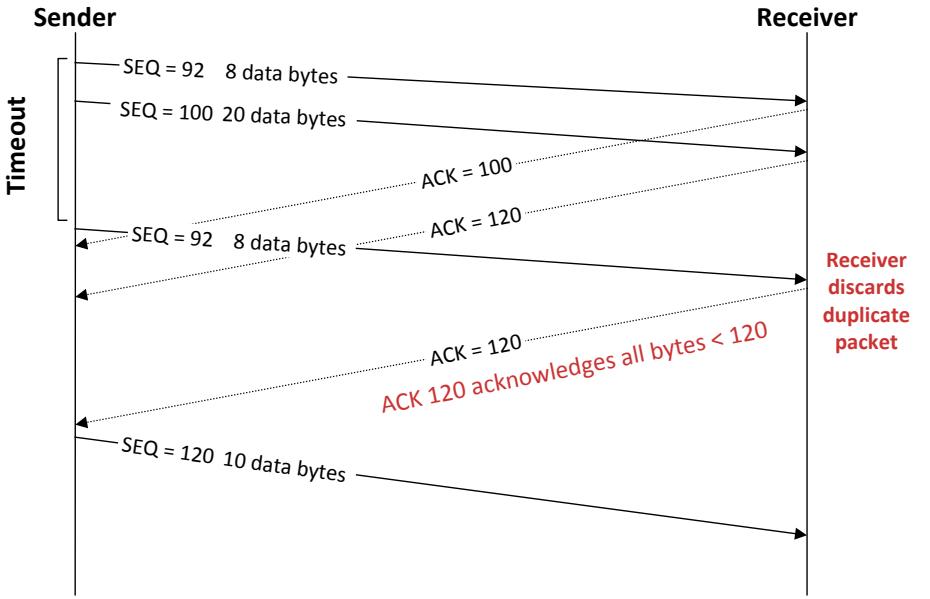
TCP SEQ + ACK
Timeout



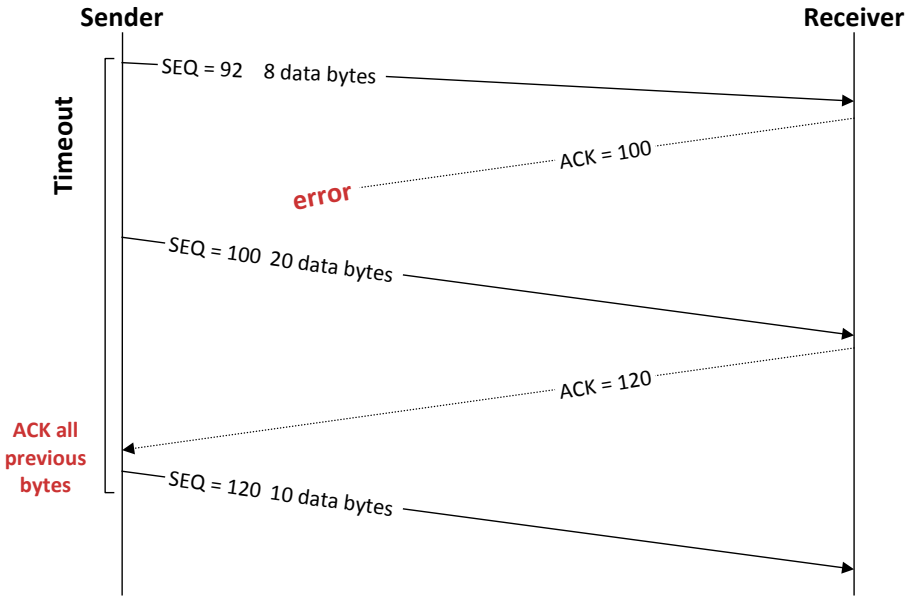
TCP SEQ + ACK
Lost ACK



TCP SEQ + ACK
Missed ACK



TCP SEQ + ACK
Cumulative ACK



Selective Acknowledgment Option

Selective ACK (SACK)

Permits ACK for segments with gaps

Option negotiated between hosts

Defined in RFC 2018

Example

Last ACK = 5000

Send 8 segments × 500 data bytes / segment

Case 1

First 4 segments received and last 4 dropped

Receiver returns normal ACK = 5000 + 4 * 500 = 7000

No SACK option field

Case 2

First segment lost and 7 segments received

For each segment receiver returns segment with

ACK = 5000

SACK option field with start + end ACK

Data	ACK	Option Field	
		Start	End
5000	—	—	—
5500	5000	5500	6000
6000	5000	5500	6500
6500	5000	5500	7000
7000	5000	5500	7500
7500	5000	5500	8000
8000	5000	5500	8500
8500	5000	5500	9000

Setting Retransmission Timeout (RTO) Value

RTO > RTT

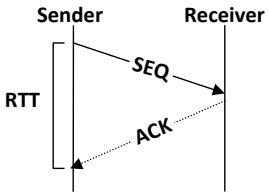
RTT = round trip time = minimum time to receive ACK

Timeout too short

Too many missed ACKs

Timeout too long

Waste too much time before re-transmission



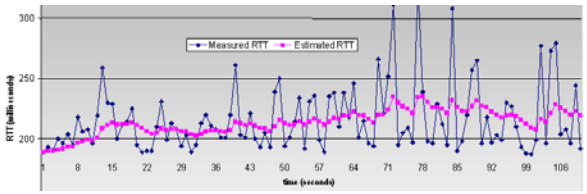
Method

Measure **RTT**

On each new ACK update

RTO ← max{(1 - α)* RTO + α * measured_RTT, 1 sec}

Typical: **α = 0.125**



TCP Connection Close

Symmetric

Client or server may close connection

FIN segment

FIN flag = 1

SEQ = cumulative SEQ number

ACK segment

ACK flag = 1

ACK = SEQ + 1

FIN segment

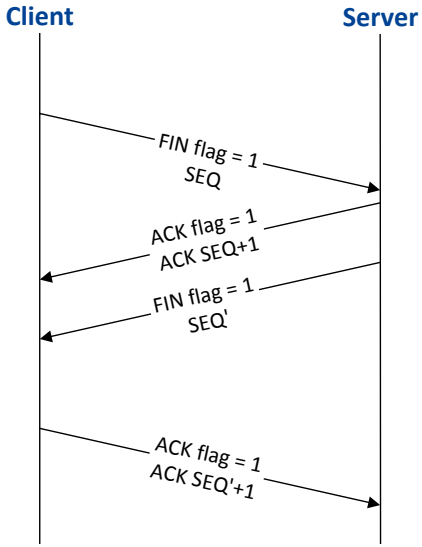
FIN flag = 1

SEQ' = cumulative SEQ number

ACK segment

ACK flag = 1

ACK = SEQ' + 1



Flow Control and Congestion Control

Flow control

Sender avoids overflow of receiver buffer

Congestion control

All sender avoid overflow of intermediate network buffers

Buffer fill rate

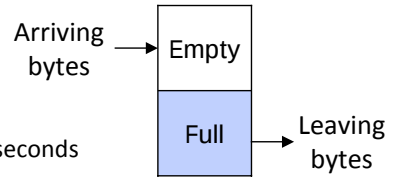
Bytes / second arriving from network

Buffer empty rate

Bytes / second leaving to network or application layer

Buffer file time

$$T_{overflow} = \frac{\text{buffer size}}{\text{buffer fill rate} - \text{buffer empty rate}}$$



Example

$$T_{overflow} = \frac{64\ \text{KB}}{8\ \text{KB/sec} - 4\ \text{KB/sec}} = \frac{64\ \text{KB}}{4\ \text{KB/sec}} = 16\ \text{seconds}$$

Flow Control

Source window

Initial source window = maximum number of "unACKed" bytes

Determined by congestion + flow control

Destination window

Number of bytes receiver can accept

Determined by available space in receiver buffer

Buffer level = Previous level + arriving bytes – bytes read by App

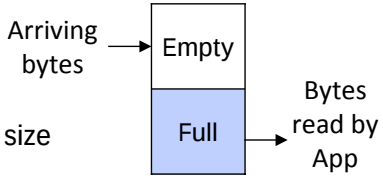
Application reads too slowly ⇒ decrease destination window

Sliding window

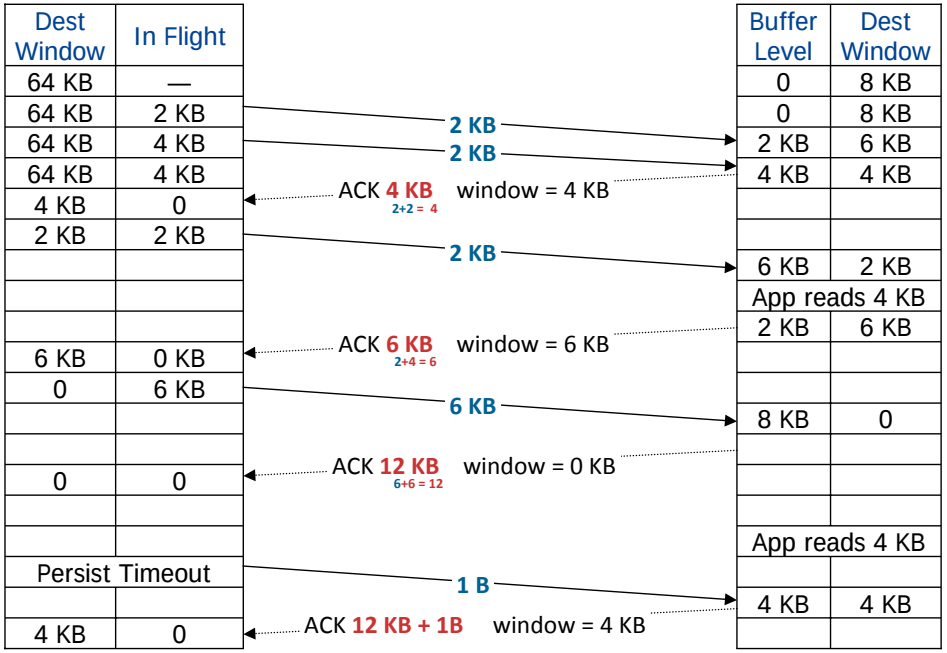
Windows field in TCP header

Number of bytes receiver will accept

Receiver discards bytes above window size



Flow Control Example



Receive Window Bugs — 1

Bug — deadlock

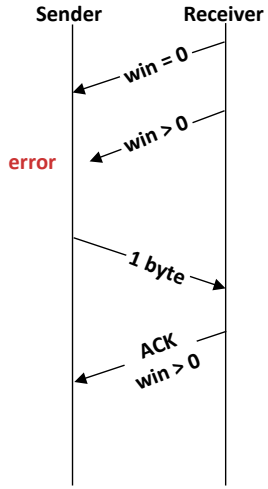
Receiver advertises window = 0

Window update with window > 0 is lost → deadlock

Fix — persist timeout

Sender attempts small segment

ACK contains new window size



Receive Window Bugs — 2

Silly Window Problem

Application reads received data slowly

Receiver advertises small window

Data bytes ~ header bytes

More segments / file transfer ⇒ larger total traffic (data + headers)

Nagle Algorithm — bug fix for Silly Window

Sender accumulates application data — sends large segments

Works badly with Telnet (requires small segments)

Receiver side bug fix

Receiver keeps 0 window size until it can advertise large window

Congestion Control

Queuing theory

Assumptions

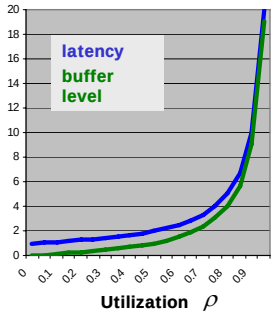
- Segments arrive independently (Poisson statistics)
  - Random length (bytes)
  - Average arrival rate in steady state
- Segments leave independently (Poisson statistics)
  - Average emptying rate in steady state

Results

$$\rho = \text{Utilization} = \frac{\text{arrival rate}}{\text{empty rate}}$$

$$\text{Latency} = \frac{1}{\text{empty rate} - \text{arrival rate}} = \frac{1}{\text{empty rate}}\left(\frac{1}{1-\rho}\right)$$

$$\text{Buffer Level} = \text{Latency} \times \text{arrival rate} = \frac{\rho}{1-\rho}$$



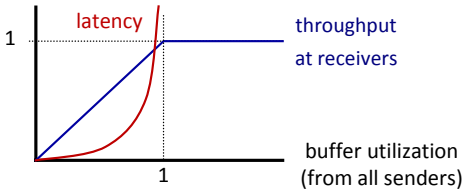
Congestion Control

Buffer throughput

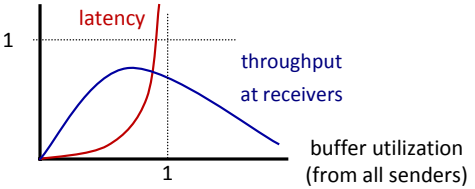
(Over)-simplified throughput model

$$\text{throughtput} = \frac{\text{receive rate}}{\text{maximum receive rate}}$$

$$\text{buffer utilization} = \frac{\text{arrival rate}}{\text{empty rate}}$$



Realistic throughput behavior



- High arrival rate at buffer
- Longer latency + overflow
- Sender timeouts
- Re-transmit ⇒ more segments ⇒ higher arrival rate at buffer

TCP Congestion Control

End-to-end congestion control

- Based on host estimates
- No feedback from intermediate network nodes

Slow-start

- Begin session with low transmission rate
- Increase rate until timeouts begin

Fast retransmit

- Do not wait for timeout
- Re-transmit after duplicate ACKs (dupACKs)

Congestion avoidance

- Limit transmission rate after duplicate ACKs
- Transmission rate → initial slow-start rate

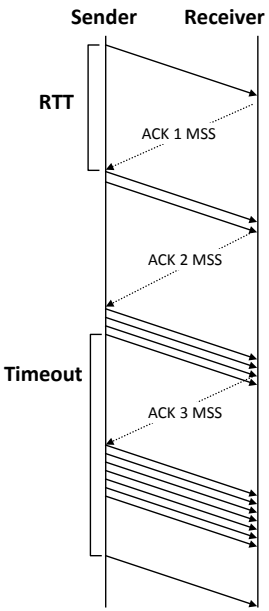
Fast recovery

- Congestion avoidance with larger transmission rate

Slow-Start

Congestion window (cwnd)

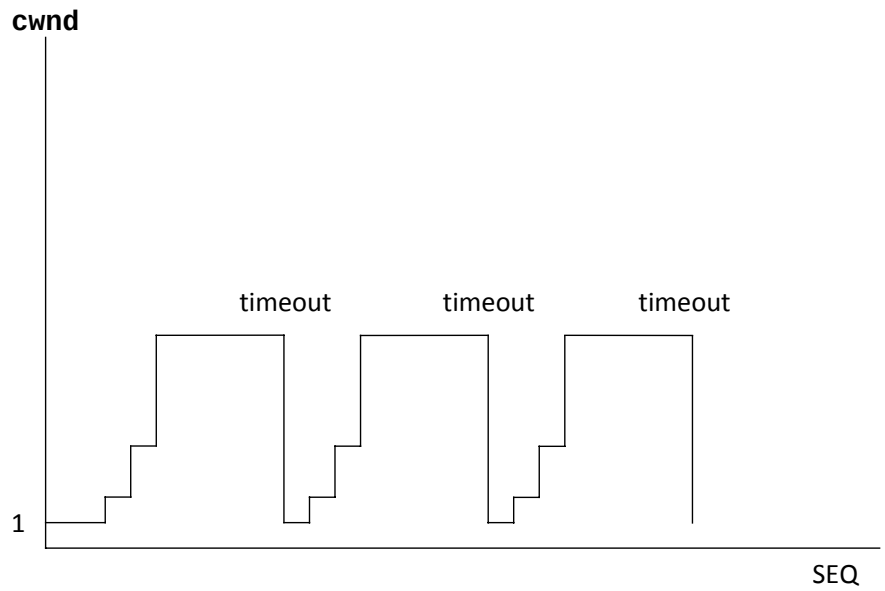
- Source window
  - Maximum number of "unACKed" bytes
- Initial **cwnd** = 1 MSS (maximum segment size)
- Data rate = 1 MSS / RTT
- Maximum **cwnd** = destination window



Exponential growth

```
On (ACK)
    cwnd ← cwnd + size of data ACKed
    if (cwnd > maximum cwnd)
        cwnd ← max cwnd
On (ACK timeout)
    cwnd ← initial cwnd = 1 MSS
```

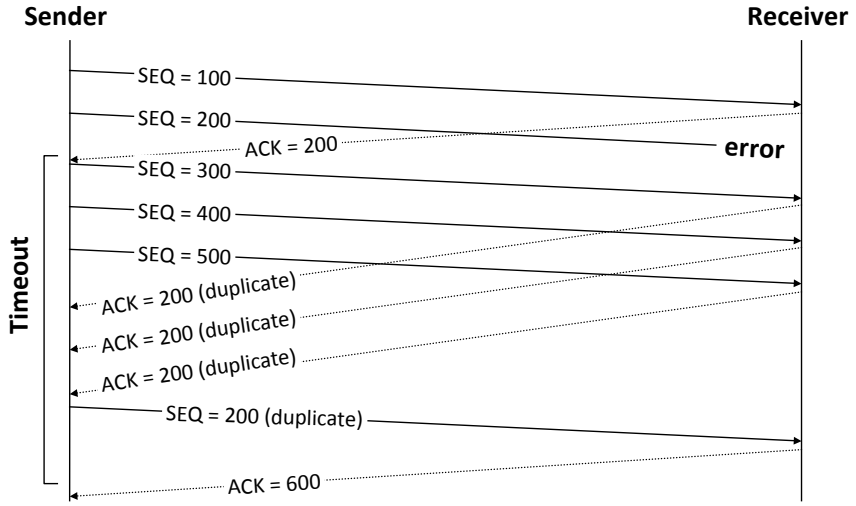
Behavior of **cwnd** in Slow Start



Fast Retransmit

Better performance with RTO >> RTT

3 duplicate ACKs (dupACKs) for segment ⇒ re-send segment



Congestion Avoidance

Tahoe protocol

Slow start threshold

ssthresh ← large initial value (possibly maximum **cwnd**)

Slow start phase

```
On (ACK && cwnd < ssthresh)
   cwnd ← cwnd + size of data ACKed
```

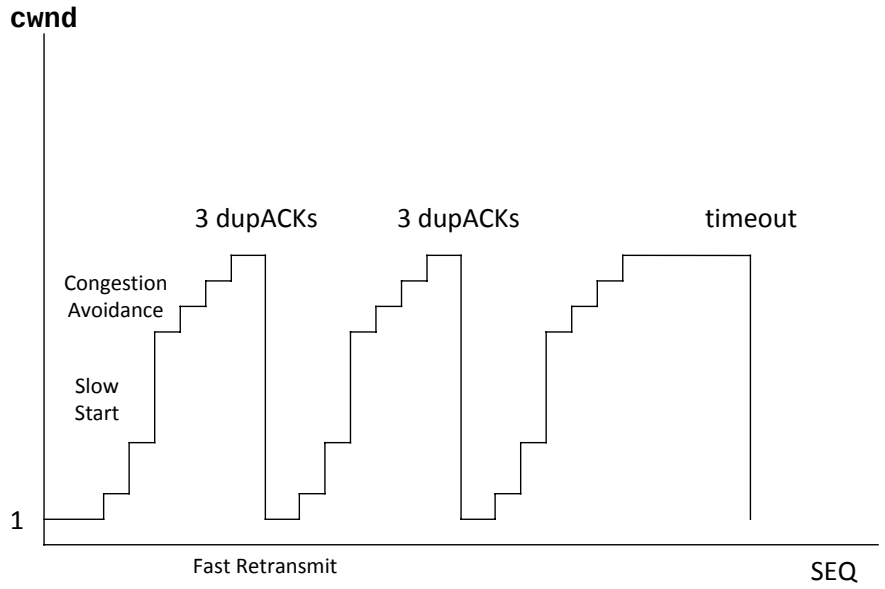
Congestion avoidance phase

```
On (ACK && cwnd > ssthresh)
   cwnd ← cwnd + 1 MSS
```
(exponential → linear growth)

Fast retransmit

```
On (ACK timeout || 3 dupACKs)
   ssthresh ← cwnd
```
(pre-timeout value)
```
   cwnd ← initial cwnd = 1 MSS
```


Behavior of **cwnd** in Tahoe



Congestion Avoidance

Reno protocol

Slow start phase

```
On (ACK && cwnd < ssthresh)
    cwnd ← cwnd + size of data ACKed
On (ACK timeout)
    ssthresh ← cwnd
    cwnd ← initial cwnd = 1 MSS
    RTO ← 2 * RTO
```

Congestion avoidance phase

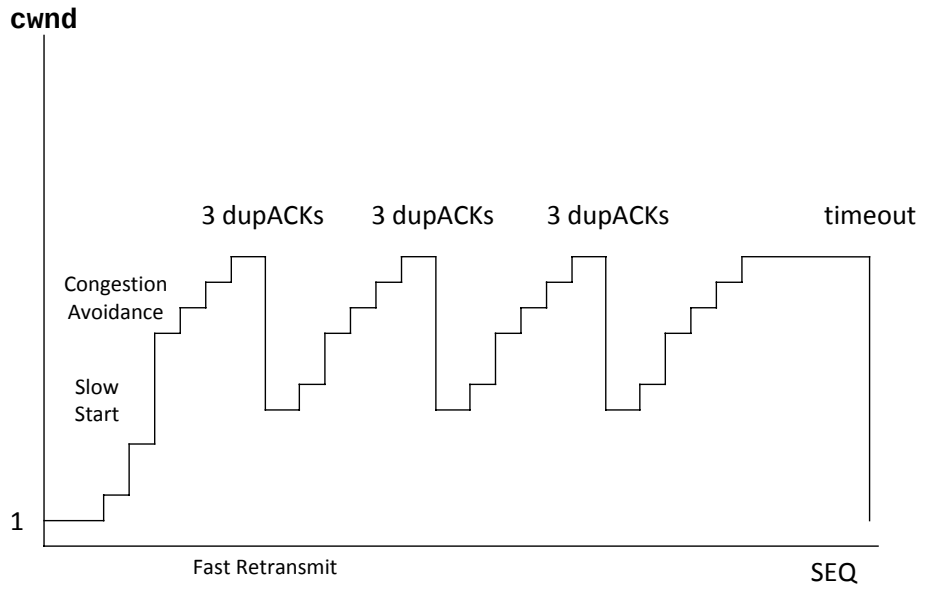
```
On (ACK && cwnd > ssthresh)
    cwnd ← cwnd + 1 MSS
```

Fast retransmit with fast recovery

```
On (3 dupACKs)
    ssthresh ← cwnd
    cwnd ← cwnd / 2
```

Retransmit lost packet
Wait 1 RTT → continue sending
For > 3 dupACKs
cwnd++ on each new dupACK

Behavior of **cwnd** in Reno



TCP Sender with Reno — 1

```
// initialize
    SEQ = ISN + 1
    SendBase = ISN + 1
    InFlight = 0
    cwnd = 1 MSS
    Set ssthreshold large (local policy)
    RTO = timeout

// main loop
on (new data from application)
    Prepare data segment:sequence number = SEQ
    if InFlight < min{cwnd,SendWindow,RecvWindow)
        Pass segment to IP
        SEQ = SEQ + length(data)
        InFlight = InFlight + length(data)
        if !(timer running) timer = RTO
```


TCP Sender with Reno — 2

```
if (receive ACK = y)
    stop timer
    if (y > SendBase)
        dupACK = 0
        newACKs = y – SendBase            // bytes ACKed
        SendBase = y
        InFlight = InFlight – newACKs
        if (cwnd < ssthresh) cwnd = cwnd + newACKs
        else cwnd = cwnd + 1 MSS
        if (InFlight > 0) timer = RTO
```

TCP Sender with Reno — 3

```
    // if (y > SendBase)
    else
        dupACK++
        if (dupACK = 3)
            SEQ = SendBase = min{unACKed SEQ} and resend
            timer = RTO
            ssthresh = cwnd
            cwnd = cwnd / 2
            wait 1 RTT   // wait for ACK of resent packet
        if (dupACK > 3)
            cwnd = cwnd + 1 MSS
if (timeout)
        SEQ = SendBase = min{unACKed SEQ} and resend
        ssthresh = cwnd
        cwnd = initial cwnd = 1 MSS
        RTO = 2 * RTO
        timer = RTO
```


TCP Receiver with Reno — 1

```
// initialize
    Set RecvWindow = receiver buffer size
    expected = Sender ISN + 1
    ack_buffer = 0
    ack_max (local policy: delayed ACK trigger)
    ack_delay = 250 msec (local policy: < 500 msec)
    Start ACK delay timer = ack_delay

// main loop
if (ACK delay timer = 0 && ack_buffer > 0)
    Send ACK = expected with updated RecvWindow
    ACK delay timer = ack_delay
    ack_buffer = 0
```


TCP Receiver with Reno — 2

```
if (receive SEQ = x)
    if (x = expected && error-free)
        expected = expected + length(data)
        if (NACK = 1)
            Send ACK = expected with updated RecvWindow
            ACK delay timer = ack_delay
            ack_buffer = 0
            NACK = 0
        else if (ack_buffer < ack_max)
            nextACK = expected
            ack_buffer++
        else if (ack_buffer = ack_max)
            Send ACK = expected with updated RecvWindow
            ACK delay timer = ack_delay
            ack_buffer = 0
    else
        Send ACK = expected with updated RecvWindow
        ACK delay timer = ack_delay
        NACK = 1
```