CS 31006: Computer Networks – The Internet Transport Protocols

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY
KHARAGPUR

Rajat Subhra Chakraborty
rschakraborty@cse.iitkgp.ac.in

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Transmission Control Protocol (TCP): History

- TCP was specifically designed to provide a reliable, end-to-end byte stream over an unreliable **internetwork**.
- **Internetwork** – different parts may have widely different topologies, bandwidths, delays, packet sizes and other parameters.
- **TCP** dynamically adapts to properties of the internetwork and is robust in the face of many kinds of failures.
- First proposed in 1974 by Cerf (Stanford) and Kahn (DARPA) as a monolithic combination of of what is now TCP and IP, separated in 1981
  - Cerf and Kahn received Turing Award in 2004
- **RFC 793 (September 1981) – Base protocol**
  - RFC 1122 (clarifications and bug fixes), RFC 1323 (High performance), RFC 2018 (SACK), RFC 2581 (Congestion Control), RFC 3168 (Explicit Congestion Notification)
- BSD UNIX started supporting TCP/IP in 1983

TCP Service Model

- Uses **Sockets** to define an end-to-end connection (Source IP, Source Port, Source Initial Sequence Number, Destination IP, Destination Port, Destination Initial Sequence Number)

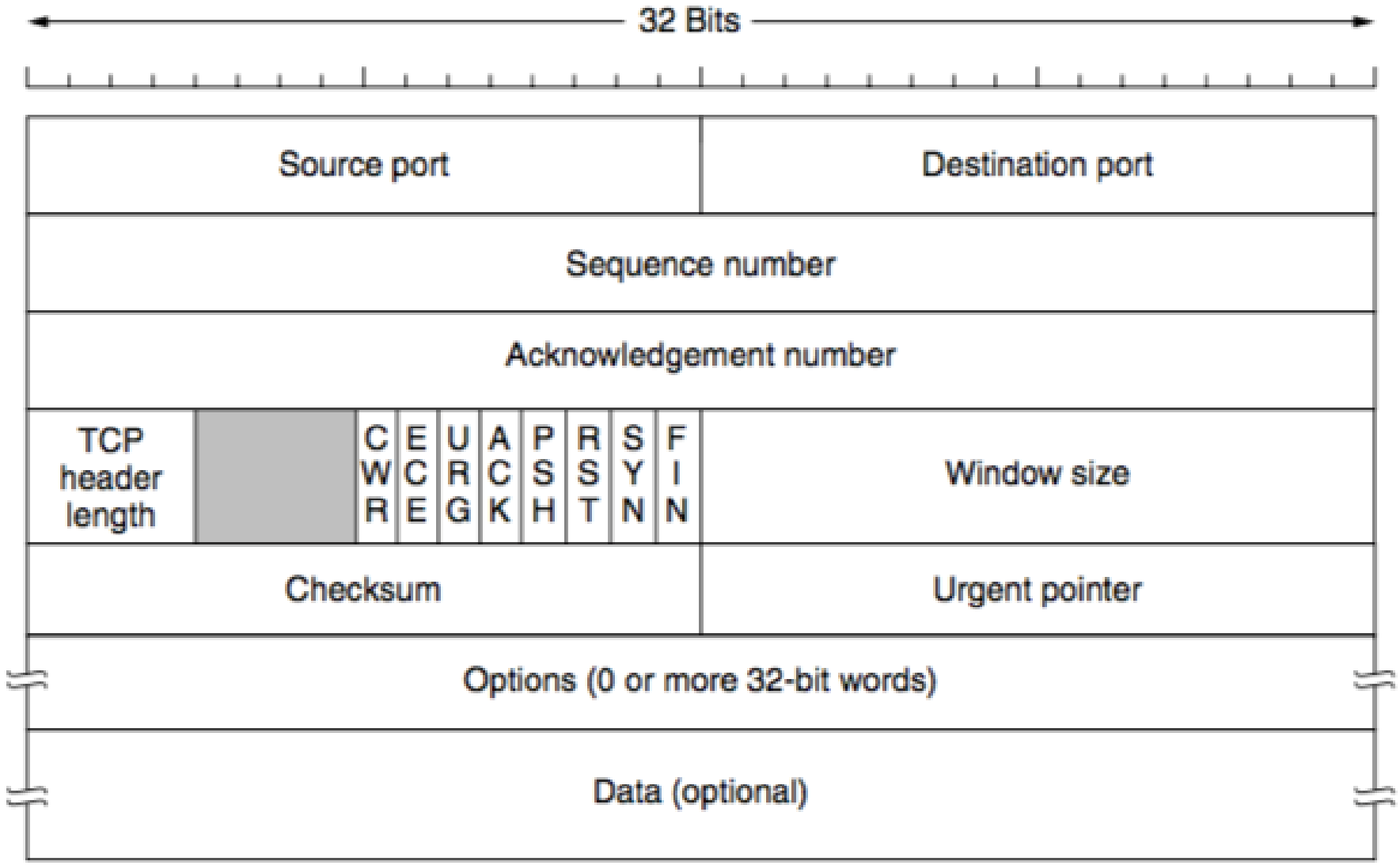
- **Unix Model of Socket Implementation:**
  - A single *super-server daemon* process, called **Internet Daemon (inetd)** runs all the times at different **well-known ports**, and wait for the first incoming connection
  - When a first incoming connection comes, *inetd* forks a new process and starts the corresponding daemon (for example *httpd* at port 80, *ftpd* at port 21 etc.)

- All TCP connections are full-duplex and point-to-point *logical connections*.
- TCP does not support multicasting or broadcasting.

TCP Service Model

- A TCP connection is a **byte stream**, not a message stream
- Message boundaries are not preserved end-to-end
- Example:
  - The sending process does four 512 byte writes to a TCP stream – through the `write()` call to the TCP socket
  - These data may be delivered as – four 512 byte chunks, two 1024 byte chunks, one 2048 byte chunk or some other way
  - There is no way for the receiver to detect the unit(s) in which the data were originally written by the sending process.

The TCP Protocol – The Header (at least 20 bytes)



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

TCP Sequence Number and Acknowledgement Number

- 32 bits sequence number and acknowledgement number
- Every byte on a TCP connection has its own 32 bit sequence number – a **byte stream** oriented connection
- TCP uses sliding window based flow control – the acknowledgement number contains next expected byte in order, which acknowledges the **cumulative bytes** that has been received by the receiver.
  - ACK number 31245 means that the receiver has correctly received up to 31244 bytes and expecting for byte 31245

TCP Segments

- The sending and receiving TCP entities exchange data in the form of **segments**
- A TCP segment consists of a fixed 20 byte header (plus an optional part) followed by zero or more data bytes
- TCP can accumulate data from several `write()` calls into one segment, or split data from one `write()` into multiple segments
- A segment size is restricted by two parameters
  - IP datagram payload (stuff other than the header): (65535 – 20) = 65515 bytes
  - Theoretical max. amount of data (in bytes):

    65535 (IP datagram size limit) - 20 (TCP header) – 20 (IP header) = 65495 bytes
  - Practically, limited by Maximum Transmission Unit (MTU) of the link, which often limits amount of data in a TCP segment to 1460 bytes

How a TCP Segment is Created

- `Write()` calls from the applications write data to the TCP sender buffer.
- sender maintains a dynamic window size based on the flow and congestion control algorithm

- Modern implementations of TCP uses **path MTU discovery** to determine the MTU of the end-to-end path (uses ICMP protocol), and sets up the **Maximum Segment Size** (**MSS**) during connection establishment
  - May depend on other parameters (buffer implementation).

- Check the sender window after receiving an ACK. If the window size is less than MSS, construct a single segment; otherwise construct multiple segments, each equals to the MSS

Challenges in TCP Design

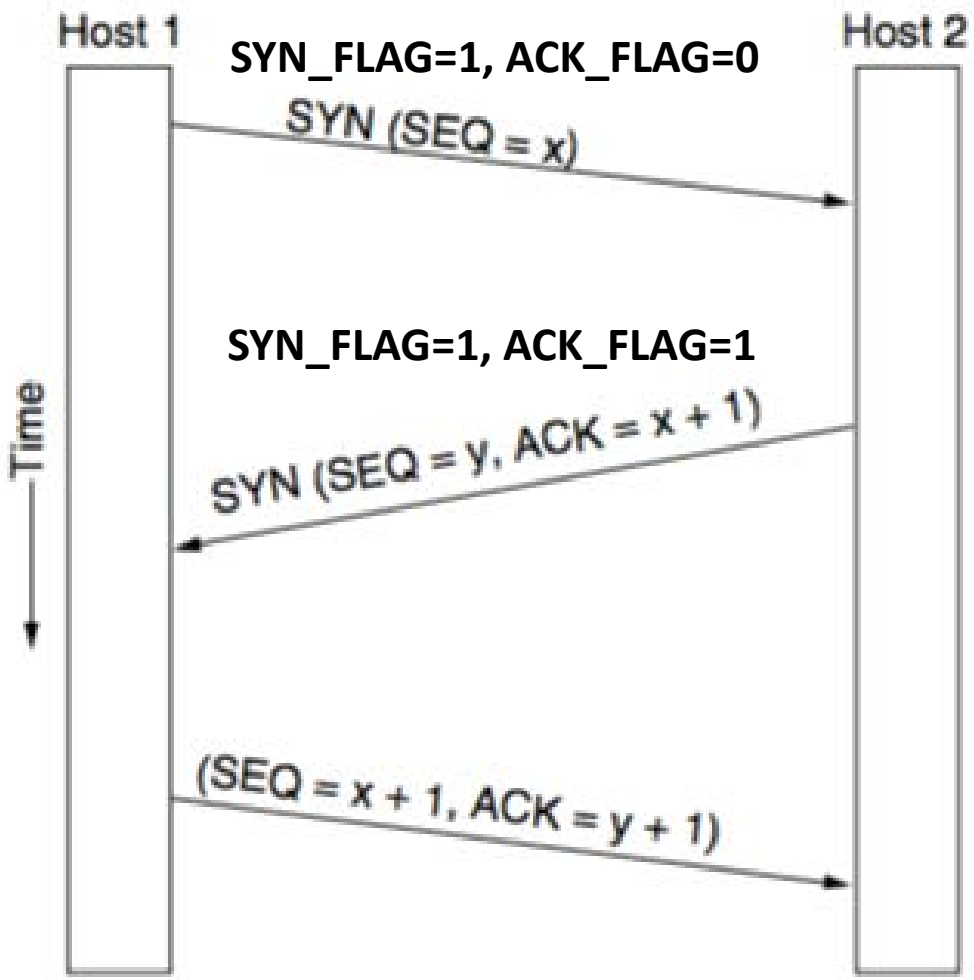
- Segments are constructed dynamically, so retransmissions do not guarantee the retransmission of the same data segment – a retransmission may contain additional data or less data.

- Segments may arrive out-of-order. TCP receiver should handle out-of-order segments in a proper way, so that received buffer space wastage is minimized.

Window Size field in the TCP Segment Header

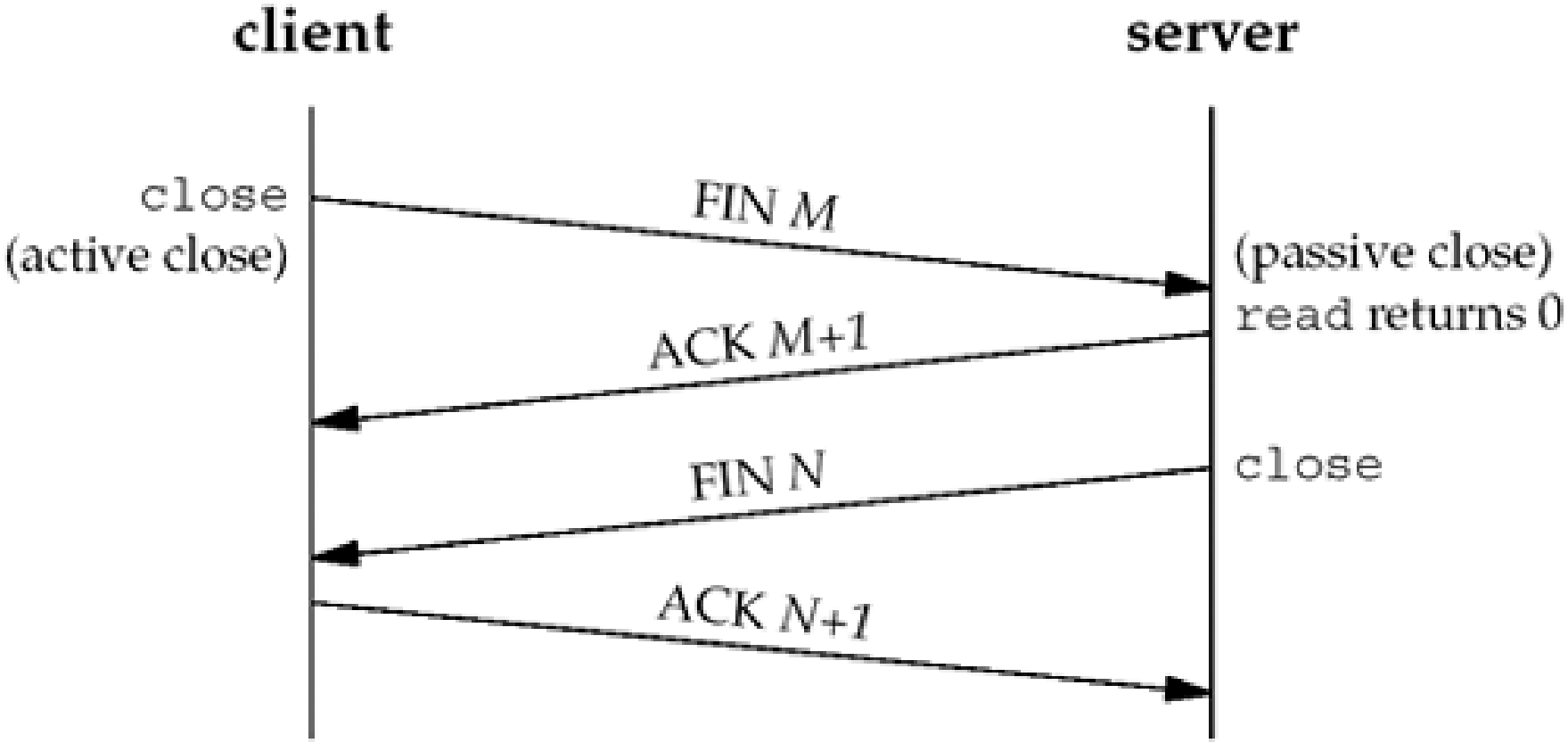
- Flow control in TCP is handled using a variable sized sliding window.
- The *window size* field tells how many bytes the receiver can receive based on the current free size at its buffer space.
- **What is meant by window size 0?**
- TCP Acknowledgement – combination of acknowledgement number and window size

TCP Connection Establishment



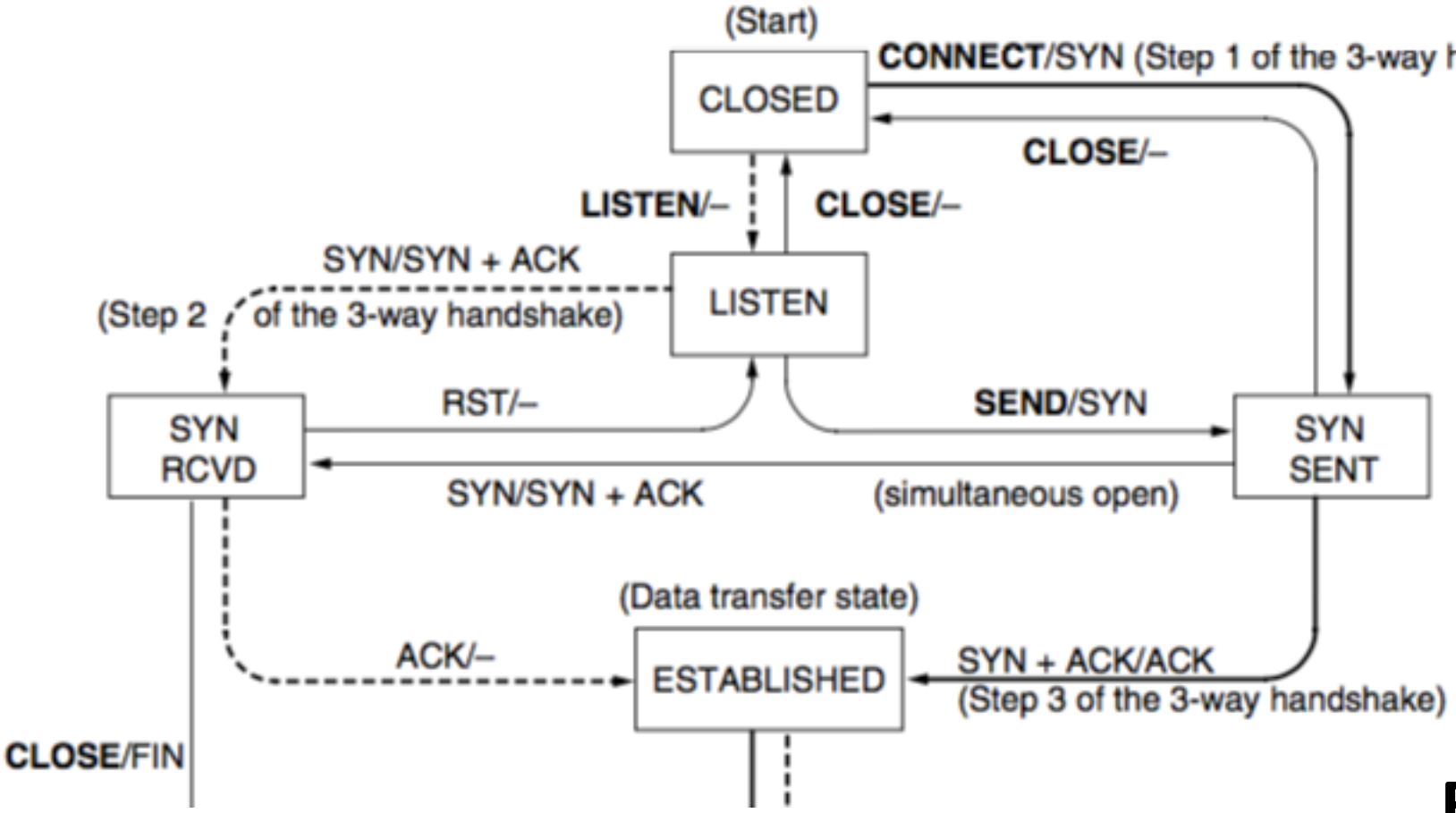
- **How to choose the initial sequence number?**
  - Protect delayed duplicates, do now generate the initial sequence number for every connection from 0
  - Original implementation of TCP used a clock based approach, the clock ticked every 4 microseconds, the value of the clock cycles from 0 to 2^{32}-1. The value of the clock gives the initial sequence number
- **TCP SYN flood attack**
  - Solution: Use cryptographic function to generate sequence numbers ("syn cookies")

TCP Connection Release



Note: it is possible to combine the 2nd and 3rd segments.

TCP State Transition Diagram – Connection Modeling

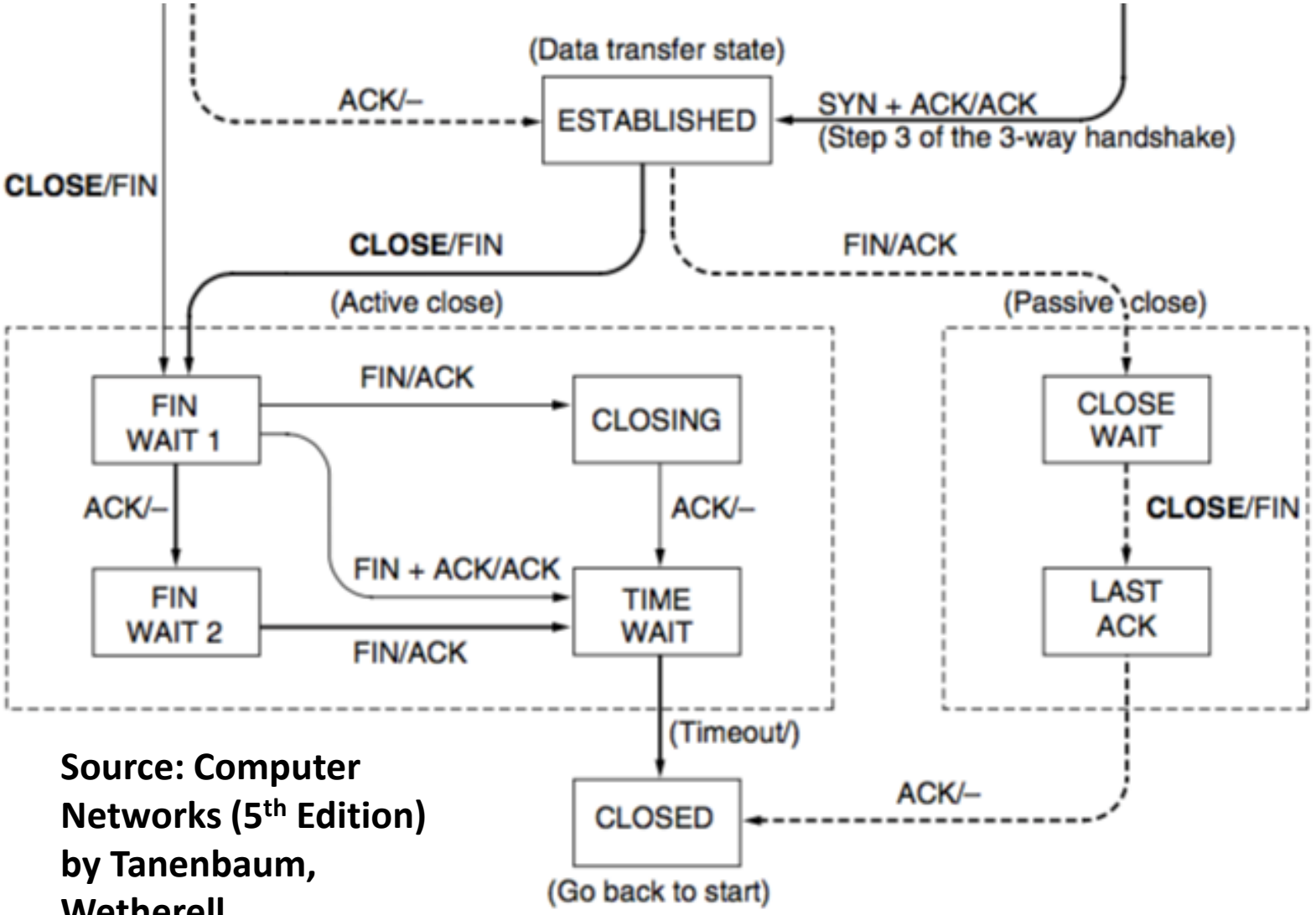


State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Event/Action
Dashed: Server
Solid: Client

TCP State Transition Diagram – Connection Modeling

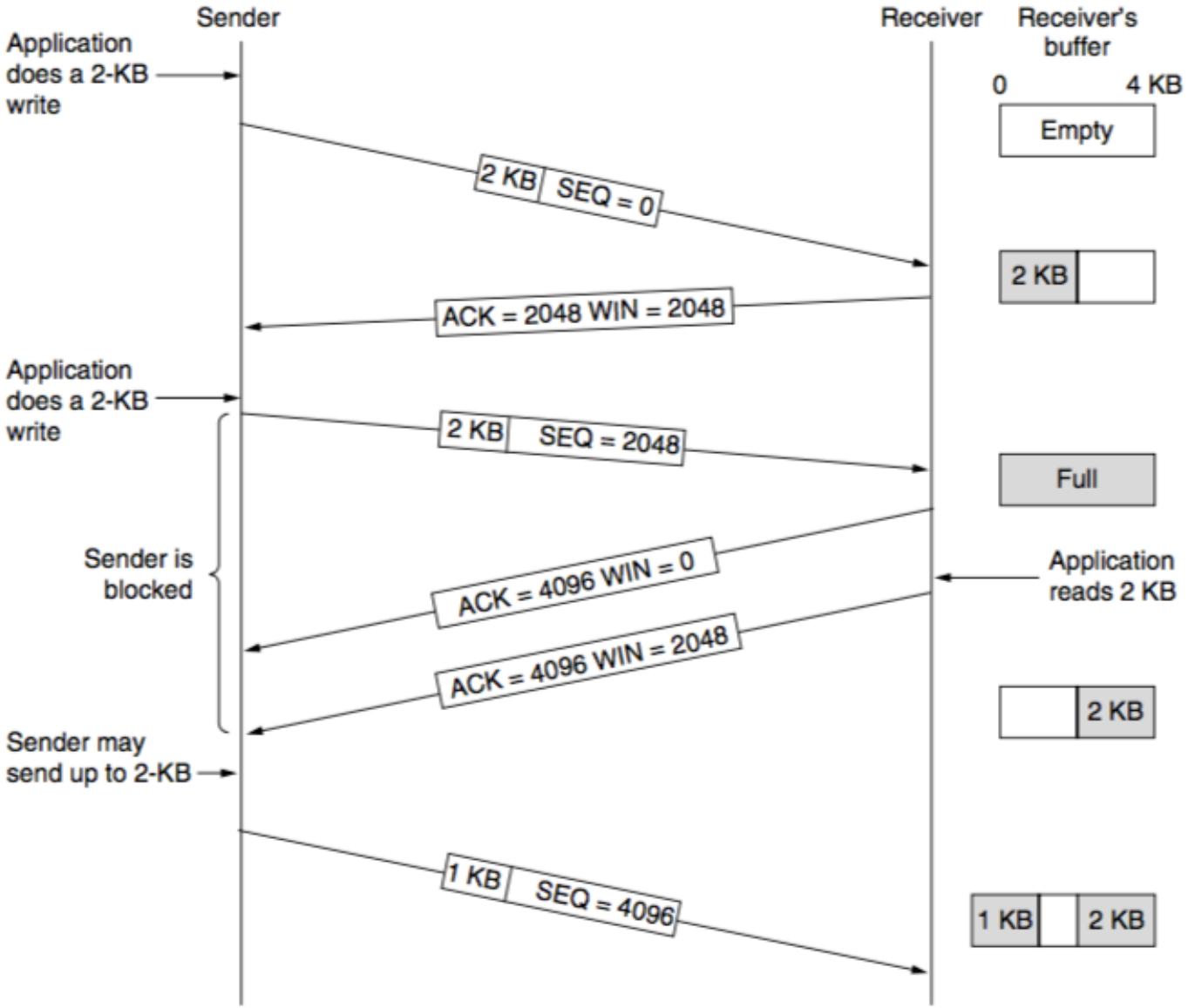


Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

Event/Action
Dashed: Server
Solid: Client

TCP Sliding Window



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Delayed Acknowledgements

- Consider a telnet connection, that reacts on every keystroke
- In the worst case, whenever a character arrives at the sending TCP entity, TCP creates a 21 byte TCP segment, 20 bytes of header and 1 byte of data. For this segment, another ACK and window update is sent when the application reads that 1 byte, and another segment to echo the character on the remote terminal
- This results in a huge wastage of bandwidth (approx. 3 times initial segment)
- **Delayed acknowledgements:** Delay acknowledgement and window updates for up to 500 msec in the hope of receiving few more data packets within that interval
- **However, the sender can still send multiple short data segments**

Nagle's Algorithm

- When data come into the sender in small pieces, just send the first piece and buffer all the rest until the first piece is acknowledged.

- Then send all buffered data in one TCP segment and start buffering again until the next segment is acknowledged.
  - **Only one short packet can be outstanding at any time.**

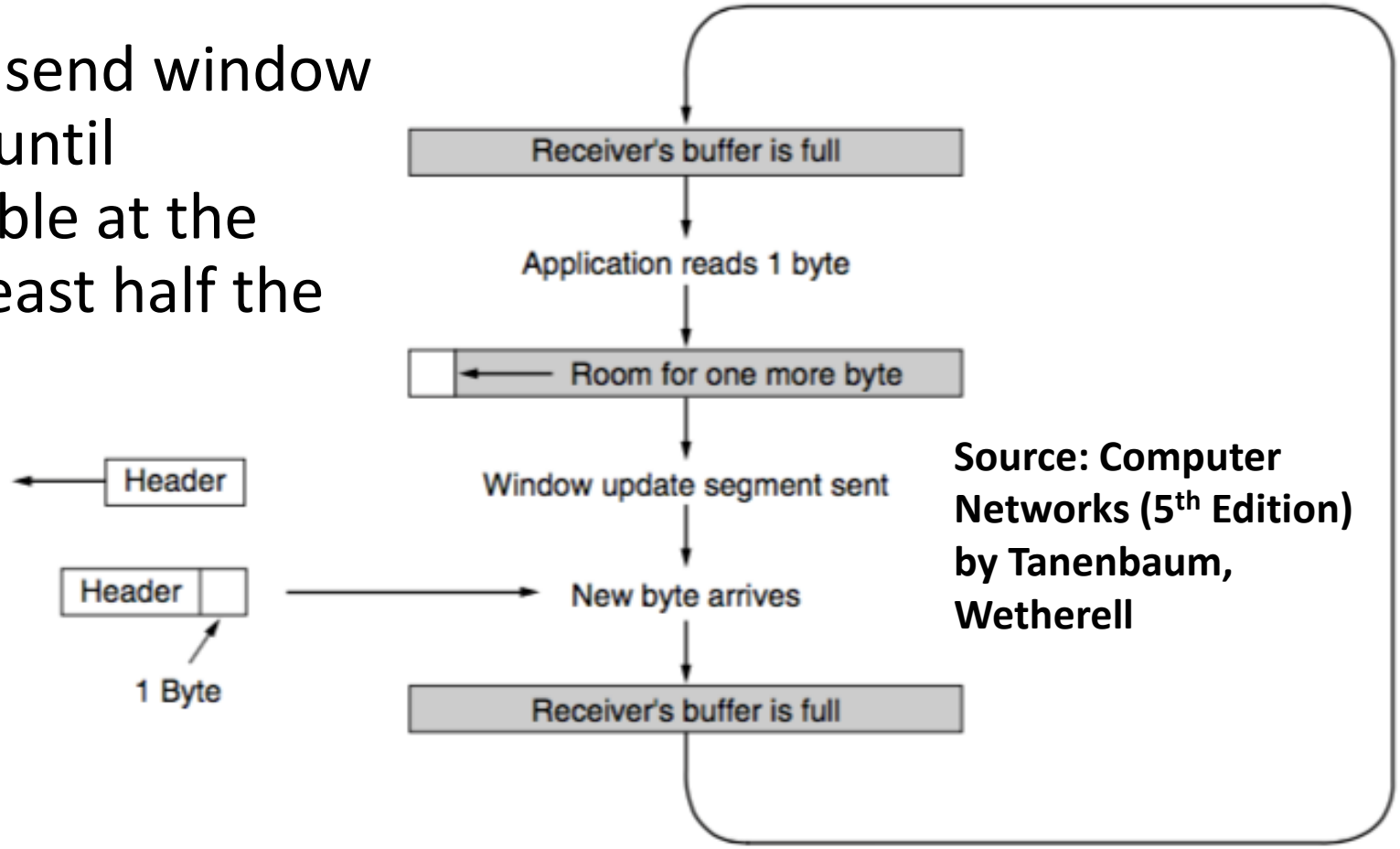
- **Do we want Nagle's Algorithm all the time?**

- **Nagle's Algorithm and Delayed Acknowledgement**
  - Receiver waits for data and sender waits for acknowledgement – results in starvation

Silly Window Syndrome

- Data are passed to the sending TCP entity in large blocks, but an interactive application on the receiver side reads data only 1 byte at a time

- **Clark's solution:** Do not send window update for 1 byte. Wait until sufficient space is available at the receiver buffer (e.g. at least half the window size)



Source: Computer Networks (5th Edition) by Tanenbaum, Wetherell

Handling Short Segments – Sender and Receiver Together

- Nagle's algorithm and Clark's solution to silly window syndrome are **complementary**
- **Nagle's algorithm:** Solve the problem caused by the sending application delivering data to TCP a byte at a time
- **Clark's solution:** Solve the problem of the receiving application fetching the data up from TCP a byte at a time
- Exception: The **PSH flag** is used to inform the sender TCP entity to create a segment immediately without waiting for more data, and the receiver TCP entity to send it to the application immediately on receipt
- **URG flag** (rarely used) is even more aggressive, allows out-of-order data delivered to receiver and to receiving application, even when receiver window is zero

Handling Out of Order in TCP

- TCP buffers out of order segments and forward a duplicate acknowledgement to the sender.
- **Acknowledgement in TCP – Cumulative acknowledgement**
- Receiver has received bytes 0, 1, 2, _, 4, 5, 6, 7
  - TCP sends a cumulative acknowledgement with ACK number 3, acknowledging everything up to byte 2
  - Once 4 is received, a duplicate ACK with ACK number 3 (next expected byte) is forwarded – **triggers congestion control**
  - After timeout, sender retransmits byte 3
  - Once byte 3 is received, it can send another cumulative ACK with ACK number 8 (next expected byte)

TCP Timer Management

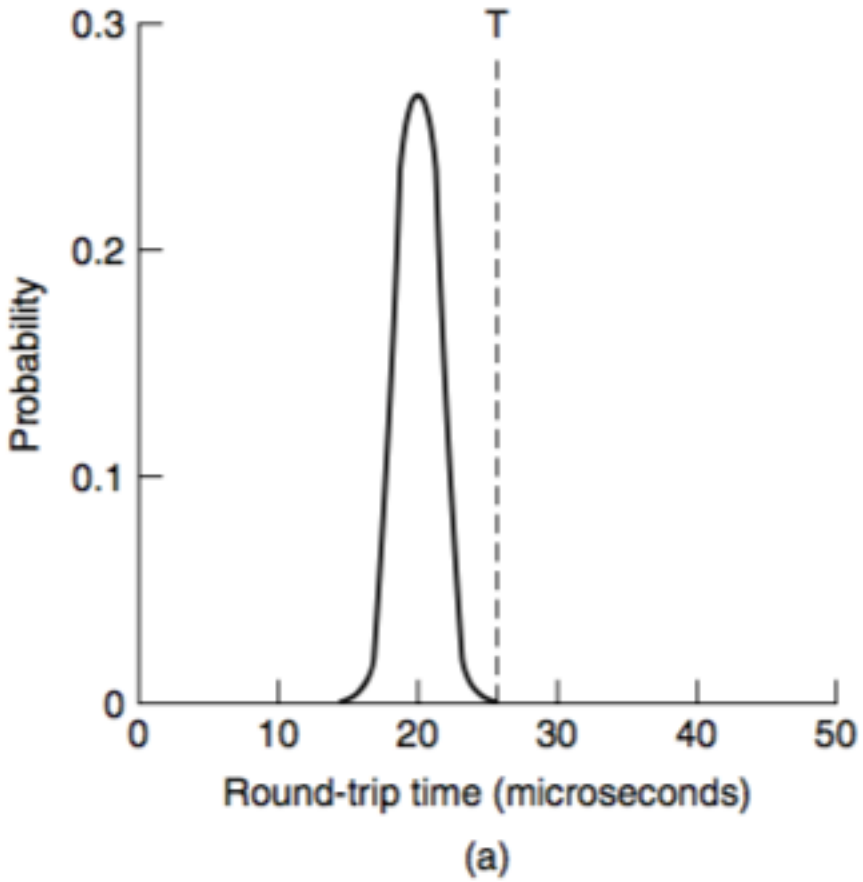
- **TCP Retransmission Timeout (RTO)**: When a segment is sent, a retransmission timer is started
  - **If the segment is acknowledged before the timer expires, the timer is stopped**
  - **If the timer expires before the acknowledgement comes, the segment is retransmitted**

- **What can be an ideal value of RTO ?**

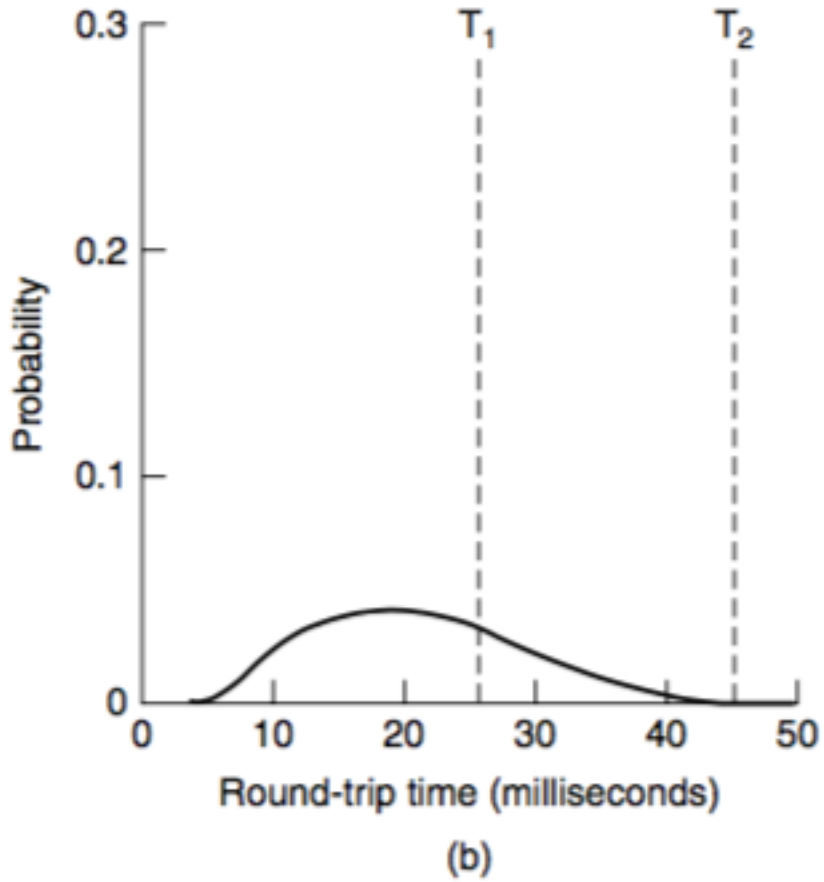
- **Possible solution**: Estimate RTT, and RTO is some positive multiples of RTT

- **RTT estimation is difficult for transport layer – why?**

RTT at Data Link Layer vs RTT at Transport Layer



Data Link Layer



Transport Layer

RTT Estimation at the Transport Layer

- Approach: use a dynamic algorithm that constantly adapts the timeout interval, based on continuous measurements of network performance.
- **Jacobson's algorithm (1988) - used in TCP**
  - For each connection, TCP maintains a variable, ***SRTT (smoothed Round Trip Time)*** – best current estimate of the round trip time to the destination
  - When a segment is sent, a timer is started (both to see how long the acknowledgement takes and also to trigger a retransmission if it takes too long)
  - If the ACK gets back – measure the time (say, *R*)
  - Update SRTT as follows

    $$\boldsymbol{SRTT_{new} = \alpha\, SRTT_{old} + (1 - \alpha)R}$$
  - **(Exponentially Weighted Moving Average – EWMA)**
  - α is a *smoothing factor* that determines how quickly the old values are forgotten. Typically $\alpha = 7/8$
  - Essentially a low-pass filtering of SRTT, hence "smoothing"
  - **Note: *SRTT* is not *RTO*!!**

Problem with EWMA

- Even given a good value of SRTT, choosing a suitable RTO is nontrivial.
- Initial implementation of TCP used RTO = 2 * SRTT
- Experience showed that a constant value was too inflexible, because it failed to response when the **variance went up (RTT fluctuation is high) – happens normally at high load**
- **Consider variance (or something similar to it) of RTT during RTO estimation**

RTO Estimation

- Update RTT variation (*RTTVAR*) as follows.

$$\boldsymbol{RTTVAR_{new} = \beta\, RTTVAR_{old} + (1 - \beta)|SRTT - R|}$$

- Typically β = ¾

- RTO is estimated as follows,

$$\boldsymbol{RTO = max\,(SRTT + 4 \times RTTVAR, 1\ second)}$$

- **Why 4 ?**
  - **Somehow arbitrary**
  - **Jacobson's paper is full of clever tricks – use integer addition, subtraction and shift – computation is lightweight, applicable even for resource-constrained devices**
  - **1 second is a conservative choice, but has become standard**
- See RFC 2988 for more details (including initial values of the variables)

Karn's Algorithm

- How will you get the RTT estimate, when a segment is lost and retransmitted again?

- **Karn's algorithm:**
  - Do not update estimates on any segments that has been retransmitted after timeout
  - However, react to the event of a segment being lost
    - The timeout is doubled each successive retransmission until the segment gets through the first time

Other TCP Timers

- **Persistent TCP Timer:** Avoid deadlock when receiver buffer is announced as zero
  - **After the timer goes off, sender forwards a probe packet to the receiver to get the updated window size**

- **Keepalive Timer:** When a connection has been idle for a long duration, check with other party when the timer goes off, close the connection if no response is received

- **TCP TIME_WAIT:** Wait before closing a connection – twice the packet lifetime

TCP Congestion Control

- Based on implementation of AIMD using a window and with packet loss as the binary signal
- TCP maintains a **Congestion Window (CWnd) –** number of bytes the sender may have in the network at any time
- **Sending Rate = Sender Window / RTT**
- **Sender Window (SWnd) = Min (CWnd, RWnd)**
- RWnd – Receiver advertised window size

1986 Congestion Collapse

- In 1986, the growing popularity of Internet led to the first occurrence of congestion collapse – a prolonged period during which goodput dropped precipitously (more than a factor of 100)

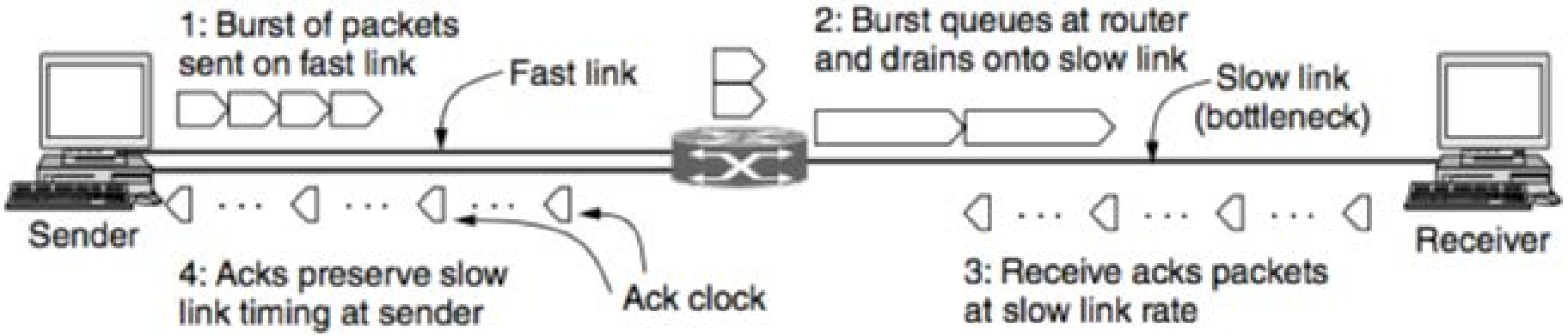
- Early TCP Congestion Control algorithm – Effort by Van Jacobson (1988)

- **Challenge for Jacobson** – Implement congestion control without making much change in the protocol (made it instantly deployable)

- **Packet loss is a suitable signal for congestion – use timeout to detect packet loss. Tune CWnd based on the observation from packet loss**

Adjust CWnd based on AIMD

- One of the most interesting ideas – use ACK for clocking



- ACK returns to the sender at about the rate that packets can be sent over the slowest link in the path.

- Trigger CWnd adjustment based on the rate at which ACK are received.

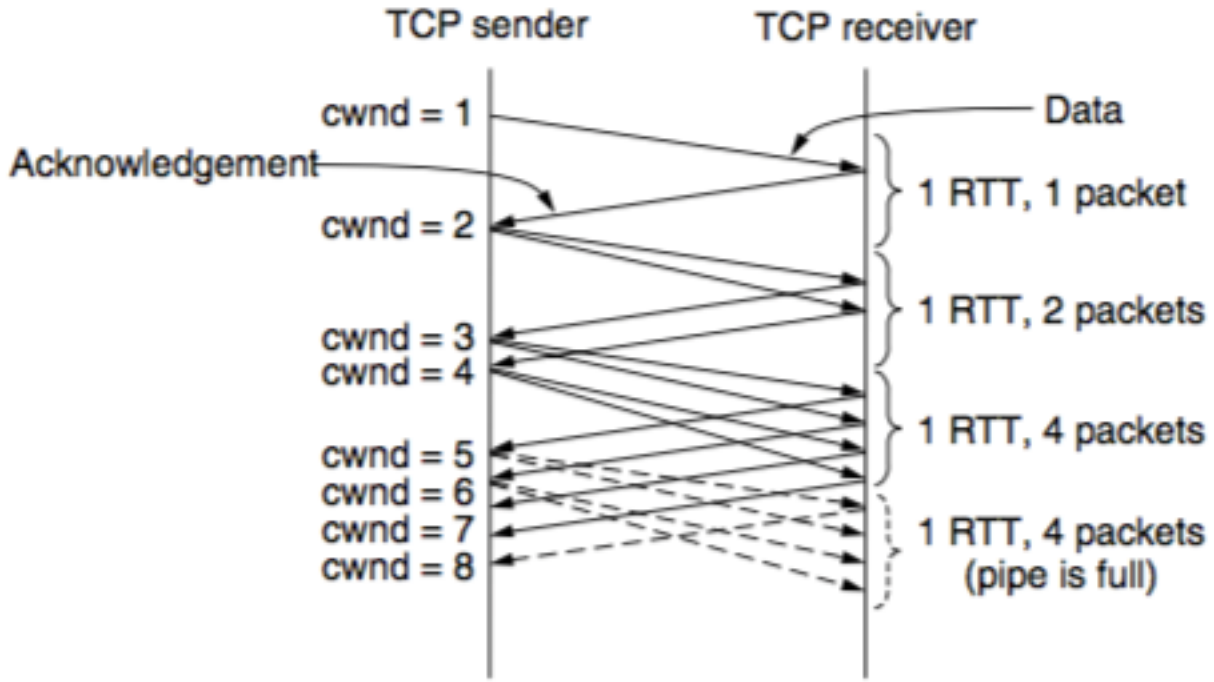
Increase Rate Exponentially at the Beginning – The *Slow Start*

- The terminology *Slow Start* is widespread, but confusing in this context!
- AIMD rule will take a very long time to reach a good operating point on fast networks if the CWnd is started from a small size.
- A 10 Mbps link with 100 ms RTT
  - Appropriate CWnd = BDP = 1 Mbit
  - 1250 byte packets -> 100 packets to reach BDP
  - CWnd starts at 1 packet, and increased 1 packet at every RTT
  - 100 RTTs are required, implying 10 sec before the connection reaches a moderate rate

- **Slow Start - Exponential increase of rate to avoid slow convergence**
  - **Rate is not slow at all !** 😃
  - **CWnd is doubled at every RTT**

TCP Slow Start

- Every ACK segment allows two more segments to be sent
- For each segment that is acknowledged before the retransmission timer goes off, the sender adds one segment's worth of bytes to the congestion window.



Slow Start Threshold

- Slow start causes exponential growth, eventually it will send too many packets into the network too quickly.
- To keep slow start under control, the sender keeps a threshold for the connection called the **slow start threshold (ssthresh).**
- Initially ssthresh is set to BDP (or arbitrarily high), the maximum that a flow can push to the network.
- Whenever a packet loss is detected by a RTO, the ssthresh is set to be half of the current ngestion window

Additive Increase (Congestion Avoidance)

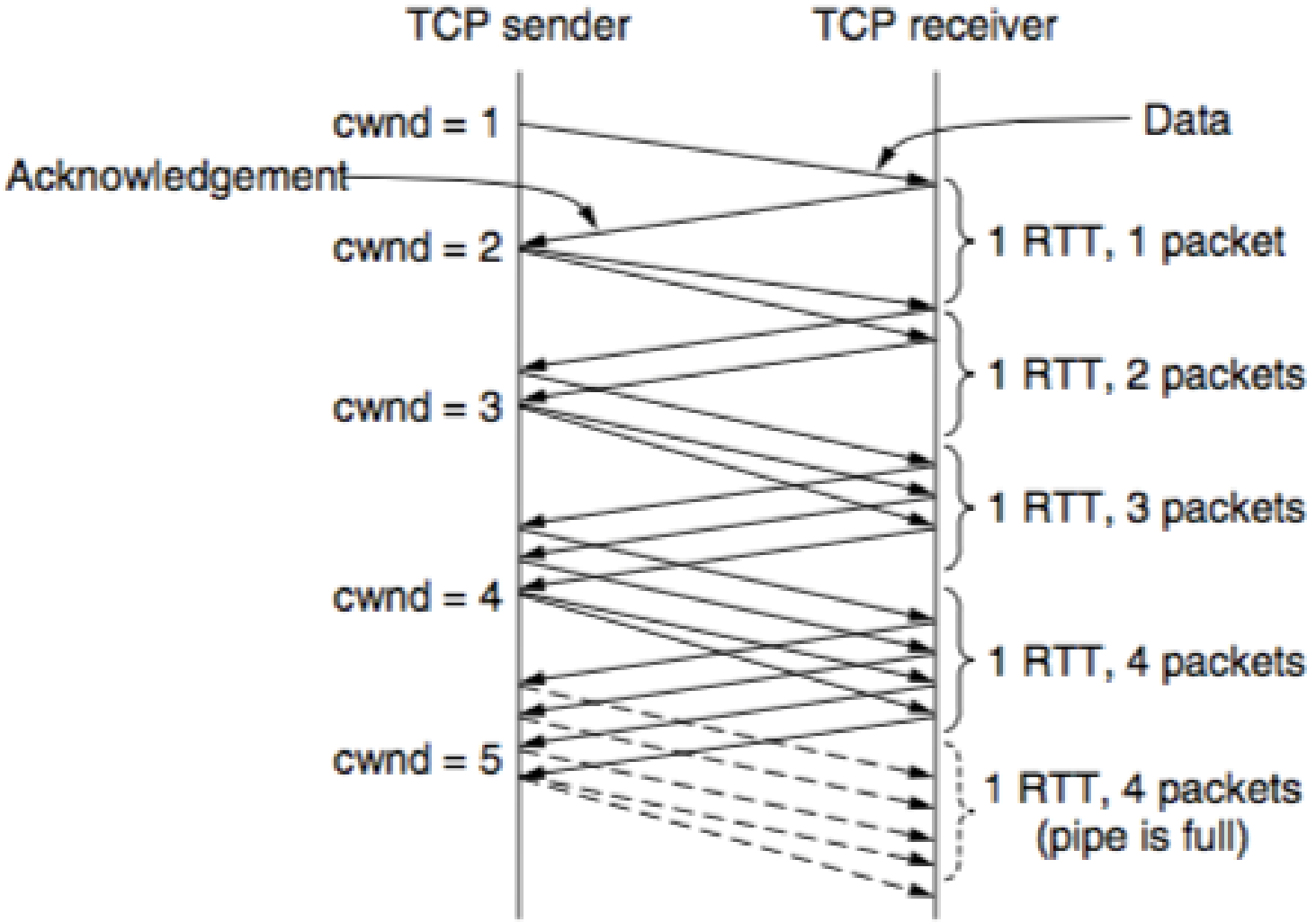
- Whenever ssthresh is crossed, TCP switches from slow start to additive increase.

- Usually implemented with an partial increase for every segment that is acknowledged, rather than an increase of one segment per RTT.

- A common approximation is to increase Cwnd for additive increase as follows (MSS is *Maximum Segment Size*, e.g. 1 kB):

$$CWnd_{new} = CWndo_{ld} + \frac{MSS \times MSS}{CWnd_{old}}$$

Additive Increase – Packet Wise Approximation

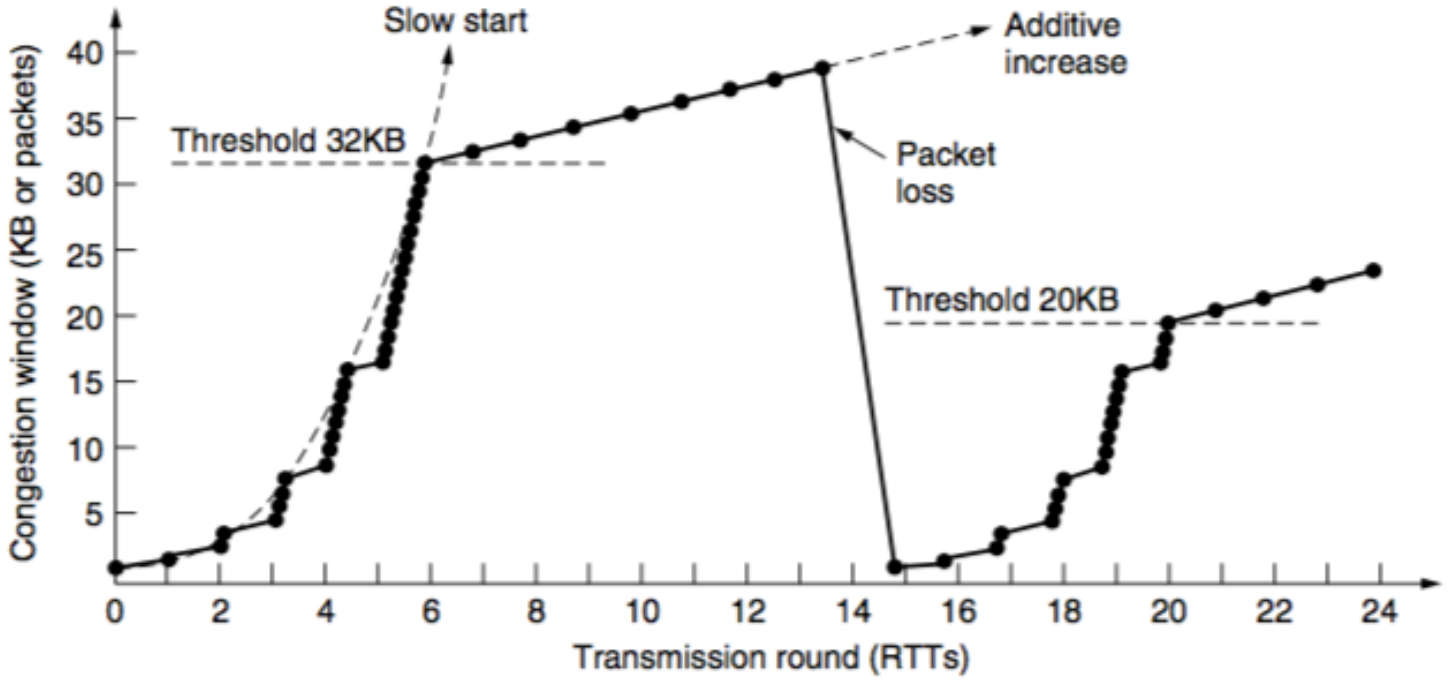


Triggering an Congestion

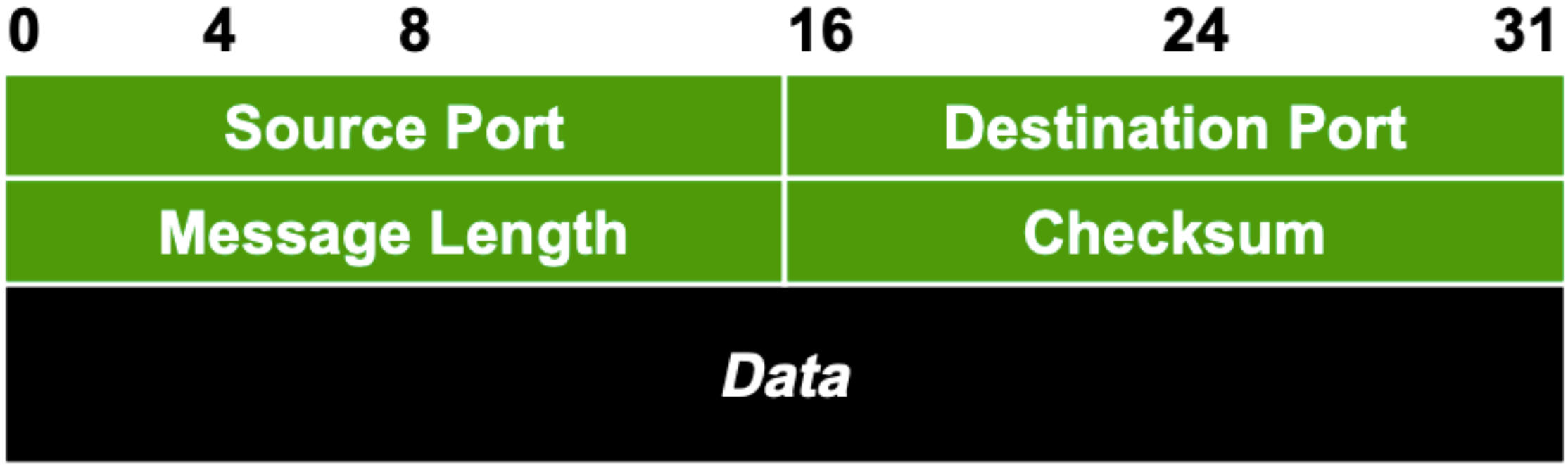
- Two ways to trigger a congestion notification in TCP – (1) RTO, (2) Duplicate ACK
- **RTO**: A sure indication of congestion, however time-consuming
- **Duplicate ACK:** TCP receiver sends a duplicate ACK when it receives out of order segment
  - **A loose way of indicating congestion**
  - **TCP arbitrarily assumes that THREE duplicate ACKs (DUPACKs) imply that a packet has been lost – triggers congestion control mechanism**
  - The identity of the lost packet can be inferred from the DUPACK – **the very next packet in sequence**
  - **Retransmit the lost packet and trigger congestion control**

Fast Retransmission – TCP Tahoe

- Use THREE DUPACK as the sign of congestion
- Once 3 DUPACKs have been received,
  - Retransmit the lost packet (**fast retransmission)** – takes one RTT
  - Set ssthresh as half of the current CWnd
  - Set CWnd to 1 MSS (1 kB in the plot)



UDP Datagram Format



UDP versus TCP

- Choice of UDP versus TCP is based on
  - Functionality
  - Performance

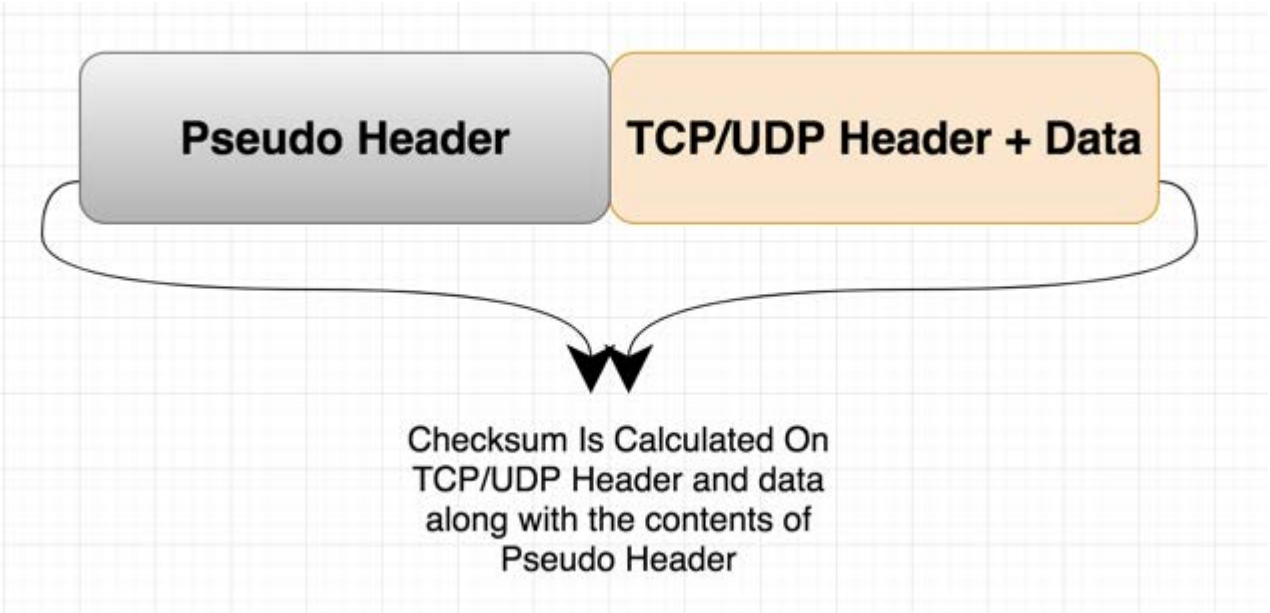
- **Performance**
  - TCP's window-based flow control scheme leads to bursty bulk transfers (not rate based)
  - TCP's "slow start" algorithm can reduce throughput
  - TCP has extra overhead per segment
  - UDP can send small, inefficient datagrams (constant bit-rate traffic)

UDP versus TCP

- **Reliability**
  - TCP provides reliable, in-order transfers
  - UDP provides unreliable service -- application must accept or deal with (a) packet loss due to overflows and errors, (b) out-of-order datagrams

- **Application complexity**
  - Application-level framing can be difficult using TCP because of Nagle algorithm
  - Nagle algorithm controls when TCP segments are sent to use IP datagrams efficiently
  - But, data may be received and read by applications in different units that how it was sent (message boundaries are not preserved in TCP)

TCP and UDP Checksum Calculation



- **Pseudo Header contains**
  - **Source IP**
  - **Destination IP**
  - **Protocol Header (TCP, UDP or ICMP)**
  - **Segment or datagram length**
  - **Reserved 8 bit**

- An additional layer of verification that the packet has reached to the intended destination (IP also has its own checksum)

This is a broad discussion of the Transport Layer … Next we'll move to the discussion of Network layer and the IP Protocol