

Flow Control

Roman Dunaytsev

The Bonch-Bruевич Saint-Petersburg
State University of Telecommunications

roman.dunaytsev@spbgut.ru

Lecture № 8

Outline

- 1 Introduction
- 2 ON/OFF
- 3 PAUSE
- 4 Stop-and-Wait
- 5 Sliding-window
- 6 Flow control in TCP
 - SWS
 - BDP
 - Scaling

- 1 Introduction
- 2 ON/OFF
- 3 PAUSE
- 4 Stop-and-Wait
- 5 Sliding-window
- 6 Flow control in TCP
 - SWS
 - BDP
 - Scaling

Introduction

- **Flow control** – a technique for assuring that a transmitting entity does not overload a receiving entity with inappropriate amount of data
 - The receiving entity typically allocates a receive buffer of some maximum size for a transfer
 - When data are received, the receiving entity must do a certain amount of processing before passing the data to the higher layer
 - The sender and the receiver may operate at different speeds
 - In the absence of flow control, the receive buffer may fill up and overflow while the receiver is processing previously received data



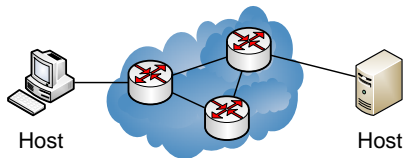
Introduction (cont'd)

- Flow control refers to the **regulating of the rate of data flow** from one device to another so that the receiver has enough time to consume the data in its receive buffer, before it overflows
 - If the receiver does not have enough resources (CPU power, buffer space, etc.) to process data as fast as the sender transmits them, mechanisms to slow down the sender are useful
 - Otherwise, the receiver would have to drop data, causing the sender to retransmit them and to waste further local and network resources
- Flow control as matching the transcription speed of a typist with the dictation rate of an author



Introduction (cont'd)

- Flow control should be distinguished from **congestion control**
- In congestion control, it is not the receiver but intermediate systems that need to be protected against overloading
 - Congestion control is relevant to multihop networks, where end systems (hosts) are connected by a series of intermediate systems (e.g., routers)
- **Flow control** – prevents the sender from overloading the receiver
- **Congestion control** – prevents the sender from overloading the network



Introduction (cont'd)

- At the same time, flow control is tightly linked with congestion control, error control, and quality of service (QoS) provisioning
- If the traffic is overcontrolled, the throughput decreases and the data transfer delay will become excessive
- It is essential for flow control to provide a good compromise between:
 - High throughput
 - High resource utilization
 - Low control overhead
- **Flow control synchronizes different processing speeds of senders and receivers**
 - It allows a receiver running on a lower speed to accept data from a sender using a higher speed, without being overloaded

Introduction (cont'd)

- **2 types of control systems** :
 - Open-loop control systems
 - Closed-loop control systems
- **Open-loop control systems** – do not have a feedback loop and, thus, are not self-correcting
- **Closed-loop control systems** – use a feedback loop
- When the characteristics of a network path and a receiver are known **a priori** and are relatively static, **open-loop flow control** can be used to control the sender transmission, without requiring a feedback loop for adjustment
 - In many cases, estimating the rate a priori can be extremely difficult

- In **open-loop flow control**, the rate is negotiated with the receiver before the flow begins
 - Negotiation with the receiver is, of course, feedback, but it generally happens once, after which the flow control is truly open-loop
 - E.g.: ATM CBR (Constant Bit Rate), VBR (Variable Bit Rate), and UBR (Unspecified Bit Rate) services
- **Closed-loop flow control** relies on feedback information from the receiver to adjust its sending rate
 - Closed-loop flow control is appropriate when the sender has little knowledge about the receiver
 - E.g.: ATM ABR (Available Bit Rate) service

Introduction (cont'd)

- **Feedback control loop** – allows the sender to transmit based on the changing ability for the receiver to accept data
 - Closed-loop flow control allows to use the available resources more efficiently, which in turn reduces congestion and data losses
- **2 key elements of closed-loop flow control :**
 - Signaling system
 - Response system
- **Signaling system** – provides the sender with information about the available resources at the receiver
- **Response system** – determines the sender's reaction to flow control signals

Introduction (cont'd)

- Signaling systems can vary in:
 - Accuracy (number of distinguishable states of receiver resource utilization)
 - Update frequency
 - Signaling path (in-band or out-of-band)
 - Relationship to other mechanisms (e.g., Stop-and-Wait or sliding-window ARQ)
- **In-band signaling** – the sending of data and control information in **the same** frequency band/circuit/channel/connection/type of PDUs, as used for data transfer
- **Out-of-band signaling** – the exchange of control information in a **separate** frequency band/circuit/channel/connection/type of PDUs

Introduction (cont'd)

- Flow control can take place at various levels:
 - **End-to-end** – protects the destination from overflow
 - **Hop level** – protects individual links and buffers from overflow
 - The further subdivision (entry-to-exit, etc.) depends on a given technology/architecture/etc.
- **2 types of flow control** :
 - Hardware flow control
 - Software flow control
- **Hardware flow control** – uses dedicated pins/circuits for signaling
 - Aka RTS/CTS (Request To Send/Clear To Send) flow control
 - This is **out-of-band signaling**, since data and flow control signals do not share the same path
- **Software flow control** – uses control characters or PDUs to return control (feedback) information to the other side

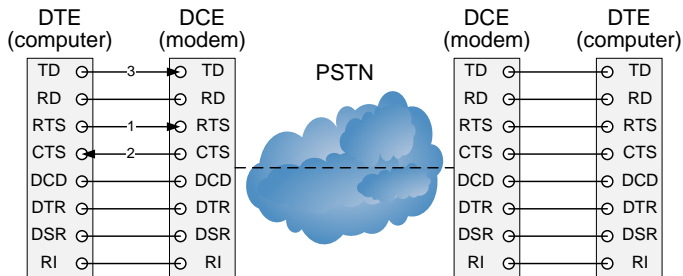
Outline

- 1 Introduction
- 2 ON/OFF**
- 3 PAUSE
- 4 Stop-and-Wait
- 5 Sliding-window
- 6 Flow control in TCP
 - SWS
 - BDP
 - Scaling

- **ON/OFF flow control** – relies on 2 control signals: ON and OFF
 - When the receiver senses that it can no longer accept incoming data, it sends an OFF signal to the sender, which causes the latter to stop transmitting
 - Once the receiver has consumed enough of the data in its receive buffer so that it can receive more, it sends an ON signal to the sender, causing it to resume transmission
- The receiver distinguishes only 2 states:
 - READY to accept data
 - NOT READY to accept data
- Works well if the propagation delay is small, otherwise the sender can overrun the receiver
- Examples of ON/OFF flow control:
 - RTS/CTS (hardware flow control, out-of-band signaling)
 - XON/XOFF (software flow control, in-band signaling)

ON/OFF (cont'd)

- **RTS/CTS** – hardware ON/OFF flow control and can be found in RS-232 between **Data Terminal Equipment (DTE)** and **Data Circuit-terminating Equipment (DCE)**
 - 1 When a DTE wants to send data to the local DCE, it turns RTS on
 - 2 If the DCE is ready to accept data, it answers by turning CTS on
 - 3 The DTE transmits data over the TD (Transmit Data) circuit to the local DCE
 - 4 As soon as the DCE turns the CTS signal off, the DTE must stop transmission



ON/OFF (cont'd)

- **XON/XOFF** – software ON/OFF flow control
 - The 'X' stands for transmitter, so XON and XOFF are control commands used to turn the sender on and off, respectively
- This is **in-band signaling**, since it inserts control commands directly into the data stream
- The sender, upon getting a XON command, transmits data at an arbitrary rate until it receives a XOFF command
- After this, the sender does not transmit any data until a XON command is again obtained

ON/OFF (cont'd)

- XON/XOFF flow control was employed in DDS
 - **Digital Data System (DDS)** – the U.S. digital transmission network that was established in 1974 by AT&T and provided full-duplex data channels at transmission rates between 2.4 and 56 kbit/s
- It used 2 characters of the ASCII (American Standard Code for Information Interchange) character set
 - The DC1 (0x11) character is used for XON and the DC3 (0x13) character is used for XOFF
 - When the sender receives an XOFF character, it must stop its transmission, and it may resume as soon as XON character is received
- This is **in-band signaling**
 - The occurrence of these characters in the payload must be prevented by using proper escaping mechanisms (character stuffing)

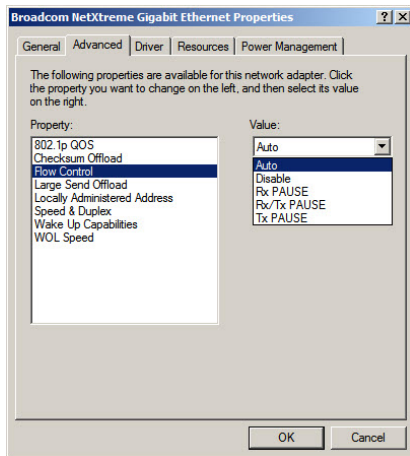
Outline

- 1 Introduction
- 2 ON/OFF
- 3 PAUSE**
- 4 Stop-and-Wait
- 5 Sliding-window
- 6 Flow control in TCP
 - SWS
 - BDP
 - Scaling

- The full-duplex mode defined by the IEEE 802.3x includes an **optional** flow control operation implemented by a special frame called **PAUSE frame**
 - This is **out-of-band signaling**
- It is defined only for use across a single full-duplex link
 - I.e., it is hop-level flow control
 - It cannot be used on a shared (half-duplex) LAN, nor does it operate through switches
- PAUSE may be used to control frame flow between **directly connected**:
 - Workstation/workstation (i.e., a simple, 2-station network)
 - Workstation/switch
 - Switch/switch

PAUSE (cont'd)

- Flow control in Ethernet
 - Rx PAUSE, Rx/Tx PAUSE, Tx PAUSE = PAUSE frame receipt, receipt and transmission, transmission is enabled, respectively

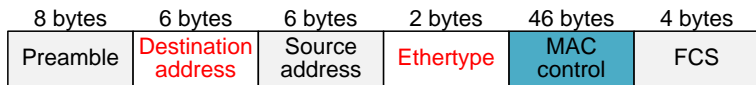


PAUSE (cont'd)

- A device can prevent buffer overflow by sending PAUSE frames to its partner on the full-duplex link
- The reception of the PAUSE frame by the partner will cause it to stop sending data frames for a given period of time
- If an additional PAUSE frame arrives before the current PAUSE time has expired, then its parameter replaces the current PAUSE time
 - Thus, a PAUSE frame with parameter 0 allows traffic to resume immediately
- During the PAUSE state the station is allowed to send only PAUSE frames, allowing 2 stations to pause each other at the same time
- After completion of the suspension period, the partner resumes its normal packet transmission

PAUSE (cont'd)

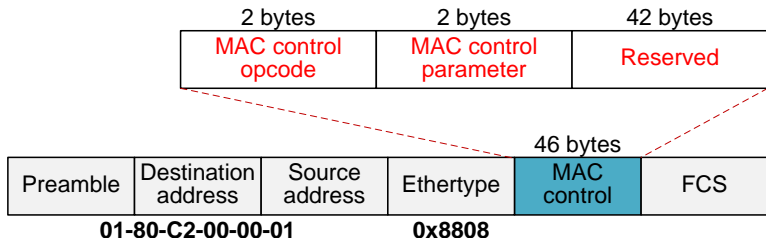
- **Destination address**, 6 bytes
 - Specifies either the unique destination address of the station to be paused or the multicast address **01-80-C2-00-00-01**
 - This multicast address has been reserved by the IEEE 802.3 for use in PAUSE frames
- **Ethertype** (aka **Type**), 2 bytes
 - Contains the reserved value of 0x8808 used for all MAC control frames



- **01**-80-C2-00-00-01 is multicast, since I/G = 1
 - The destination MAC address of FF-FF-FF-FF-FF-FF is broadcast

PAUSE (cont'd)

- **MAC control opcode**, 2 bytes
 - Contains the control operation code (aka **opcode**) for PAUSE – 0x0001
- **MAC control parameter**, 2 bytes
 - Specifies the duration of the PAUSE event (from 0x0000 to 0xFFFF) in units of 512-bit times
 - **Bit time** = the amount of time required to send 1 bit on the network
 - E.g., 10 Mbit/s: 0 - 3355 ms; 100 Mbit/s: 0 - 335.5 ms
- **Reserved**, 42 bytes
 - Padded with '0's to keep the minimum length of 46 bytes



Outline

- 1 Introduction
- 2 ON/OFF
- 3 PAUSE
- 4 Stop-and-Wait**
- 5 Sliding-window
- 6 Flow control in TCP
 - SWS
 - BDP
 - Scaling

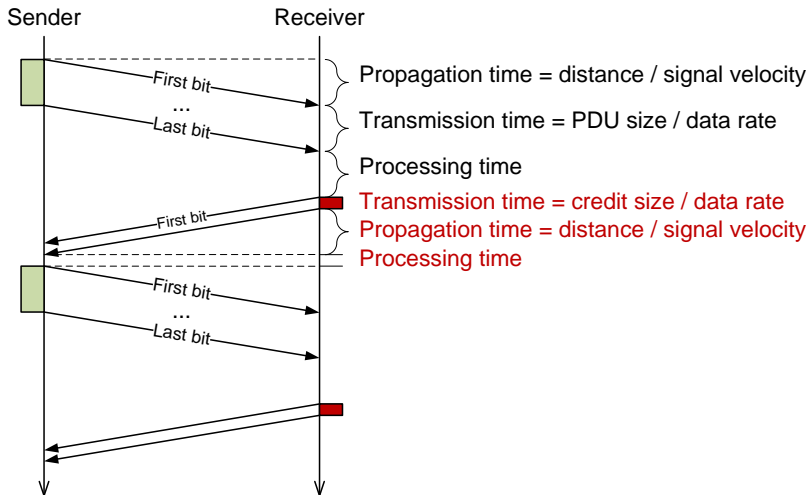
- **Stop-and-Wait flow control** is based on Stop-and-Wait ARQ:
 - The sender transmits a PDU
 - After the receiver gets this PDU, it indicates its readiness to accept another PDU by sending back a **credit** (aka a **permit** or an **ACK**)
 - The sender must wait until it receives the credit before sending the next PDU
 - Thus, only 1 PDU at a time can be in transit
 - The receiver can thus stop the flow of data simply by delaying a credit
- Stop-and-Wait is used in Type 3 LLC service
 - I.e., acknowledged connectionless service

Stop-and-Wait (cont'd)

- **Benefits** of Stop-and-Wait flow control:
 - Simplicity and ease of implementation
- **Shortcomings** of Stop-and-Wait flow control:
 - The throughput and link utilization are generally low
- I.e., even though the receiver can process many PDUs at a time, it can not get them together due to the limit of maximum 1 PDU
 - In this case, the link remains unnecessarily idle
- One way to improve the link efficiency is to allow more than 1 PDU to be transmitted before a credit can be expected from the receiver
 - Aka sliding-window flow control (similar to **continuous ARQ**)

Stop-and-Wait (cont'd)

- **Processing time** – due to the finite processing power of the devices

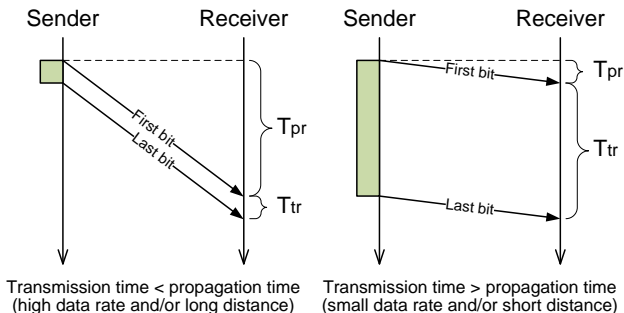


Stop-and-Wait (cont'd)

- **Transmission time** – due to a finite data rate of a link
- **Transmission time = PDU size (in bits) / data rate (in bits/s)**
 - Typically, the transmission time for a credit is considered to be 0
 - This is because a credit is assumed to be very small in size resulting in a negligible value of the transmission time
 - Similar reason could be used for 0 processing time for a credit
- **Propagation time** – due to a finite velocity of signal propagation
 - The speed of light in vacuum is about 300,000 km/s
 - The velocity of signal propagation in optical fiber and coaxial copper cable is about 70% of the speed of light in vacuum ($\sim 200,000$ km/s)
 - The velocity of radio waves is close to the speed of light in vacuum
- **Propagation time = distance (in m) / signal velocity (in m/s)**
 - E.g., a USA coast-to-coast path has the propagation delay of ~ 25 ms

Stop-and-Wait (cont'd)

- For very **high data rates** and/or **long distances** between the sender and receiver (e.g., WAN), Stop-and-Wait flow control provides inefficient link utilization
- For very **small data rates** and/or **short distances** between the sender and receiver (e.g., LAN), Stop-and-Wait flow control provides approximately the same link utilization as sliding-window flow control



Outline

- 1 Introduction
- 2 ON/OFF
- 3 PAUSE
- 4 Stop-and-Wait
- 5 Sliding-window**
- 6 Flow control in TCP
 - SWS
 - BDP
 - Scaling

- **Bit length of a link** – the number of bits present on the link when a stream of bits fully occupies the link
- **Bit length of a link = data rate (in bits/s) * propagation time (in seconds)**
- In situations where the bit length of the link is greater than the PDU length, Stop-and-Wait flow control is highly inefficient
- Efficiency can be significantly improved by allowing multiple PDUs to be sent back-to-back at once

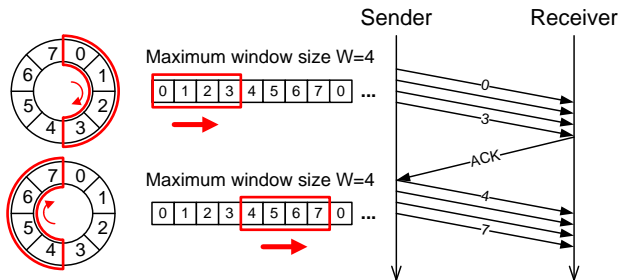
Sliding-Window (cont'd)

- **Sliding-window** is the generalization of Stop-and-Wait to more than 1 PDU
 - I.e., the receiver allows the sender to send up to a certain number of PDUs without getting further credit or ACK
 - This maximum number of PDUs allowed to be transmitted without receiving a credit is said to be **the maximum window size**
- Variants of sliding-window flow control:
 - Credits issued at the end of windows
 - Credits (ACKs) issued after each PDU
 - Credit mechanisms allowing variable-size windows
 - etc.

Sliding-Window (cont'd)

• Example:

- Let the receiver specify the maximum window size of 4 PDUs
- Suppose that PDUs can be numbered from 0 through 7
- PDUs 0, 1, 2, 3 have been sent and successfully received
- After the processing of the received PDUs is complete, the receiver sends an ACK for all these PDUs
- On receiving this ACK, the sender is permitted to send PDUs 4, 5, 6, 7



Sliding-Window (cont'd)

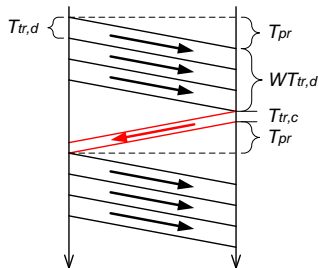
- Some implementations also allow a system to cut off the flow of PDUs from the other side by sending a **Receive Not Ready (RNR)** command, which acknowledges former PDUs but forbids transfer of future PDUs
- As soon as the system can accept new data, it sends a **Receive Ready (RR)** command
 - This RR cancels the effect of the previous RNR
- It also contains the sequence number of the next expected PDUs
- E.g., the **High-level Data Link Control (HDLC)** protocol uses special supervisory frames (S-frames) for this purpose: the RR and RNR frames
 - Since user data are transmitted in another type of PDUs (I-frames), this is **out-of-band signaling**

Sliding-Window (cont'd)

- **Credits issued at the end of windows**

- Data rate in bits/s = R
- Data PDU transmission time = $T_{tr,d}$
- Credit transmission time = $T_{tr,c}$
- Propagation delay = T_{pr}
- Window size in PDUs = W

$$R_{effective} = \frac{W * T_{tr,d}}{W * T_{tr,d} + T_{tr,c} + 2 * T_{pr}} * R$$

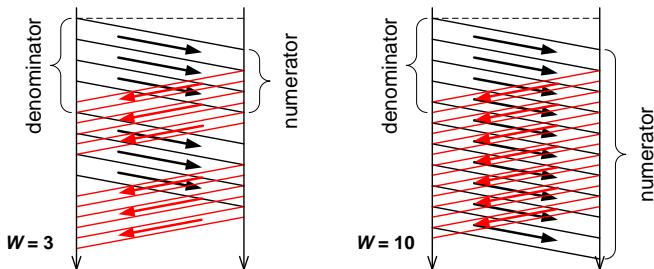


Sliding-Window (cont'd)

- Credits issued after each PDU, advancing the window by 1

- Data rate in bits/s = R
- Data PDU transmission time = $T_{tr,d}$
- Credit transmission time = $T_{tr,c}$
- Propagation delay = T_{pr}
- Window size in PDUs = W

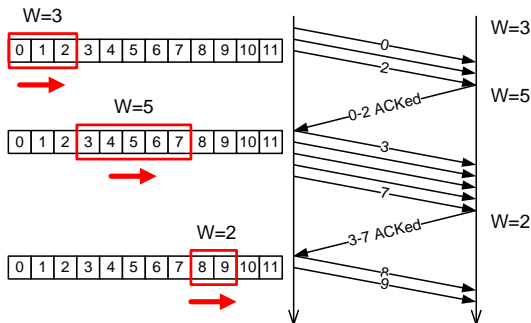
$$R_{effective} = \min \left(\frac{W * T_{tr,d}}{T_{tr,d} + T_{tr,c} + 2 * T_{pr}} * R, R \right)$$



Sliding-Window (cont'd)

• Variable-size windows

- More flexibility is obtained if variable-size windows are used and adjusted according to buffer occupancy
- It can be implemented by including a 'window size' field in credits to specify the number of PDUs or bytes allowed to be sent
- This type of flow control is widely used at the transport layer (e.g., ISO Transport Protocol (TP), IETF TCP)

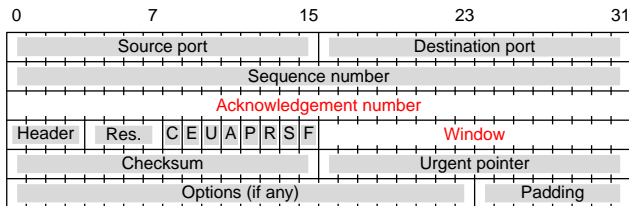


Outline

- 1 Introduction
- 2 ON/OFF
- 3 PAUSE
- 4 Stop-and-Wait
- 5 Sliding-window
- 6 Flow control in TCP**
 - SWS
 - BDP
 - Scaling

Flow Control in TCP

- The receiving TCP returns 2 parameters to the sending TCP:
 - Acknowledgement (ACK) number
 - Window size (aka the receive window, rwnd)
- The interpretation is 'I am ready to receive bytes with sequence numbers ACK, ACK+1, ACK+2, ..., ACK+rwnd-1'
- Since ACKs can be piggybacked onto data segments, TCP uses **in-band signaling**



Flow Control in TCP (cont'd)

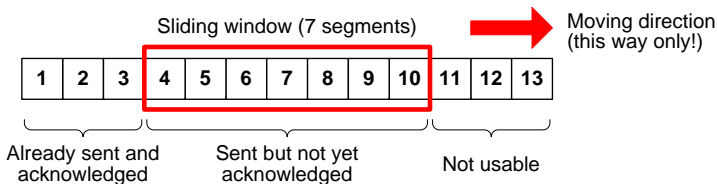
- During connection establishment each host allocates a receive buffer of some size to this connection
- Thus, the rwnd defines the amount of free room in the receive buffer
- Since TCP is not permitted to overflow the allocated buffer, the sender must not transmit more data than it was defined by the receiver
- Because the free room at the receive buffer changes over time, the rwnd value dynamically changes during lifetime of the connection
- The maximum value of the rwnd (without scaling) is $2^{16} - 1 = 65,535$ bytes

Flow Control in TCP (cont'd)

- To utilize the network bandwidth effectively, TCP uses sliding-window flow control to send multiple segments at a time before it stops to wait for an ACK
- In high-speed networks with a large distance between the sender and receiver, this approach is far more efficient than sending 1 segment at a time and waiting for the ACK
- At any time, **window = min (rwnd, cwnd)**
- **rwnd (receive window)** is a receiver-side limit on the amount of outstanding data
- **cwnd (congestion window)** is a sender-side limit on the amount of outstanding data

Flow Control in TCP (cont'd)

- Note that TCP operates in a byte-stream fashion, not in segments
- But for simplicity, we consider TCP operation in terms of 'segments'
- The maximum amount of data that can be placed into the segment is limited by the Maximum Segment Size (MSS)
 - For IPv4, $MSS = MTU - 40$ bytes
 - Typically, the MSS is equal to 1460 bytes
- A **full-sized** segment - a segment that contains the maximum number of data

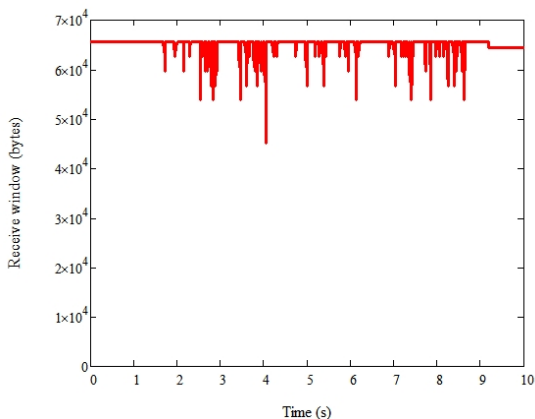


Flow Control in TCP (cont'd)

- If there is no free room in the receive buffer, the receiver sends an ACK with $rwnd=0$
 - But even in this case, the sender will periodically transmit a **probe segment**
 - Such a segment (from host A to host B) generally contains $Seq(A) = ACK(B) - 1$ with 1 (or more) byte(s) of garbage data
 - The receiver drops this segment, but since it has to acknowledge its reception, in the ACK segment it can define a new value of the $rwnd$
 - Eventually the buffer will begin to empty and a new ACK will contain a non-zero value of the $rwnd$
 - Probing zero window prevents the sender from blocking

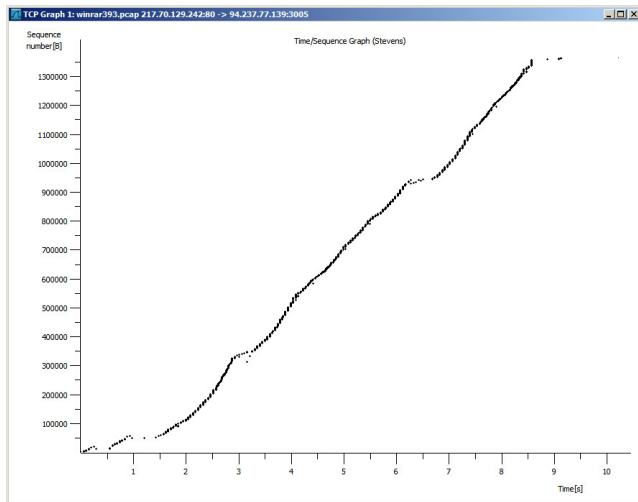
Flow Control in TCP (cont'd)

- Because the free room at the receive buffer changes over time, the `rwnd` value dynamically changes during lifetime of the connection
- **Example 1**



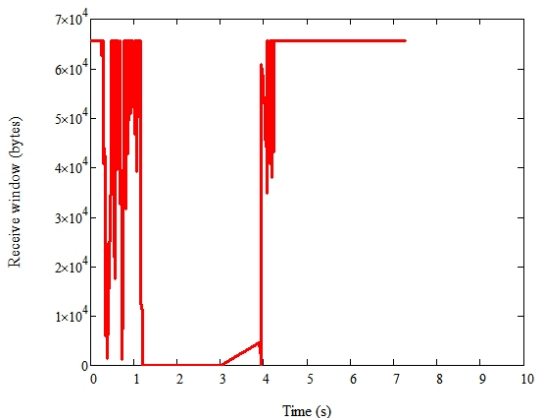
Flow Control in TCP (cont'd)

- **Example 1:** the time-sequence (Stevens) graph



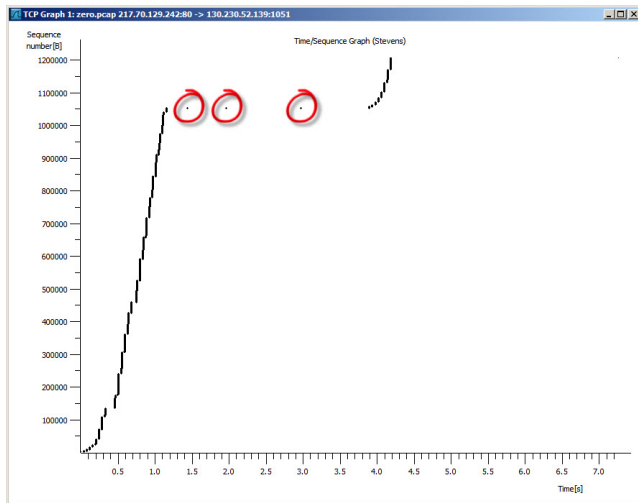
Flow Control in TCP (cont'd)

- When the receive buffer fills completely, the receiver sends an ACK that contains a zero window advertisement ($rwnd=0$)
- **Example 2**



Flow Control in TCP (cont'd)

- **Example 2:** the time-sequence graph with denoted probe segments



Flow Control in TCP (cont'd)

- **Example 2:** the SYN segment
 - No Window Scale option, rwnd = 65,535 bytes

```
1 0.000000 130.230.52.139 217.70.129.242 TCP [SYN] Seq=0 Win=65535 Len=0 MSS=1460 SACK_PERM=1
  Frame 1: 62 bytes on wire (496 bits), 62 bytes captured (496 bits)
  Ethernet II, Src: Notebook_d3:25:19 (00:06:1b:d3:25:19), Dst: All-HSRP-routers_34 (00:00:0c:07:ac:34)
  Internet Protocol, Src: 130.230.52.139 (130.230.52.139), Dst: 217.70.129.242 (217.70.129.242)
  Transmission Control Protocol, Src Port: optima-vnet (1051), Dst Port: http (80), Seq: 0, Len: 0
    Source port: optima-vnet (1051)
    Destination port: http (80)
    [Stream index: 0]
    Sequence number: 0 (relative sequence number)
    Header length: 28 bytes
  Flags: 0x02 (SYN)
    000. .... = Reserved: Not set
    ...0 .... = Nonce: Not set
    .... 0... = Congestion window Reduced (CWR): Not set
    .... 0... = ECN-Echo: Not set
    .... 0... = Urgent: Not set
    .... 0... = Acknowledgement: Not set
    .... 0... = Push: Not set
    .... 0... = Reset: Not set
  .... 0... = Syn: Set
  .... 0... = Fin: Not set
  window size: 65535
  Checksum: 0x4337 [validation disabled]
  Options: (8 bytes)
    Maximum segment size: 1460 bytes
    NOP
    NOP
    TCP SACK Permitted Option: True
  0000  00 00 0c 07 ac 34 00 06 1b d3 25 19 08 00 45 00  ....4...%...E.
  0010  00 30 21 be 40 00 80 06 c6 5f 82 e6 34 8b d9 46  .0!@...4..F
  0020  81 f2 04 1b 00 50 84 70 a4 62 00 00 00 00 70 02  ....P.p..b....p.
  0030  ff ff 43 37 00 00 02 04 05 b4 01 01 04 02  c7....
```


Flow Control in TCP (cont'd)

- At 1.201527 s of the TCP connection, the data flow was interrupted from the receiver side by setting $rwnd = 0$ bytes
- At 3.895820 s of the TCP connection (almost after 2.7 s), the data flow was resumed by announcing $rwnd = 4600$ bytes

| | | | | | |
|------|----------|----------------|----------------|-----|---|
| 1197 | 1.160999 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1198 | 1.161037 | 130.230.52.139 | 217.70.129.242 | TCP | optima-vnet > http [ACK] Seq=584 Ack=1038061 win=12975 Len=0 |
| 1199 | 1.161052 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1200 | 1.199320 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1201 | 1.199417 | 130.230.52.139 | 217.70.129.242 | TCP | optima-vnet > http [ACK] Seq=584 Ack=1040321 win=10715 Len=0 |
| 1202 | 1.199449 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1203 | 1.199512 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1204 | 1.199546 | 130.230.52.139 | 217.70.129.242 | TCP | optima-vnet > http [ACK] Seq=584 Ack=1043241 win=7795 Len=0 |
| 1205 | 1.199597 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1206 | 1.201016 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1207 | 1.201065 | 130.230.52.139 | 217.70.129.242 | TCP | optima-vnet > http [ACK] Seq=584 Ack=1045361 win=5675 Len=0 |
| 1208 | 1.201123 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1209 | 1.201262 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1210 | 1.201298 | 130.230.52.139 | 217.70.129.242 | TCP | optima-vnet > http [ACK] Seq=584 Ack=1048281 win=2755 Len=0 |
| 1211 | 1.201352 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1212 | 1.201492 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1213 | 1.201527 | 130.230.52.139 | 217.70.129.242 | TCP | [TCP ZeroWindow] optima-vnet > http [ACK] Seq=584 Ack=1051036 win=0 Len=0 |
| 1214 | 1.483208 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP Keep-Alive] http > optima-vnet [ACK] Seq=1051035 Ack=584 win=6996 Len=0 |
| 1215 | 1.483288 | 130.230.52.139 | 217.70.129.242 | TCP | [TCP ZeroWindow] optima-vnet > http [ACK] Seq=584 Ack=1051036 win=0 Len=0 |
| 1216 | 2.007222 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP Keep-Alive] http > optima-vnet [ACK] Seq=1051035 Ack=584 win=6996 Len=0 |
| 1217 | 2.007295 | 130.230.52.139 | 217.70.129.242 | TCP | [TCP ZeroWindow] optima-vnet > http [ACK] Seq=584 Ack=1051036 win=0 Len=0 |
| 1218 | 3.015228 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP Keep-Alive] http > optima-vnet [ACK] Seq=1051035 Ack=584 win=6996 Len=0 |
| 1219 | 3.015295 | 130.230.52.139 | 217.70.129.242 | TCP | [TCP ZeroWindow] optima-vnet > http [ACK] Seq=584 Ack=1051036 win=0 Len=0 |
| 1220 | 3.895820 | 130.230.52.139 | 217.70.129.242 | TCP | [TCP window update] optima-vnet > http [ACK] Seq=584 Ack=1051036 win=4600 Len=0 |
| 1221 | 3.936388 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |
| 1222 | 3.936455 | 217.70.129.242 | 130.230.52.139 | TCP | [TCP segment of a reassembled PDU] |

Flow Control in TCP (cont'd)

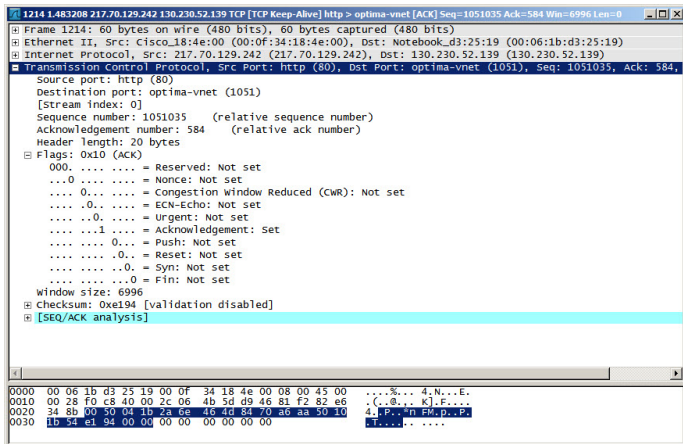
- The receiver's ZeroWindow segment
 - rwnd = 0 bytes
 - ACK = 1051036 (relative)

```
1213 1.201527 130.230.52.139 217.70.129.242 TCP [TCP ZeroWindow] optima-vnet > http [ACK] Seq=584 Ack=1051036 Win=0 Len=0
  Frame 1213: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
  Ethernet II, Src: Notebook_d3:25:19 (00:06:1b:d3:25:19), Dst: All-MSRP-routers_34 (00:00:0c:07:ac:34)
  Internet Protocol, Src: 130.230.52.139 (130.230.52.139), Dst: 217.70.129.242 (217.70.129.242)
  Transmission Control Protocol, Src Port: optima-vnet (1051), Dst Port: http (80), Seq: 584, Ack: 1051036,
    Source port: optima-vnet (1051)
    Destination port: http (80)
    [Stream index: 0]
    Sequence number: 584 (relative sequence number)
    Acknowledgement number: 1051036 (relative ack number)
    Header length: 20 bytes
  Flags: 0x10 (ACK)
    000. .... = Reserved: Not set
    ...0 .... = Nonce: Not set
    .... 0... = Congestion window Reduced (CWR): Not set
    .... .0.. = ECN-Echo: Not set
    .... ..0. = Urgent: Not set
    .... ...1 = Acknowledgement: Set
    .... .... 0... = Push: Not set
    .... .... .0. = Reset: Not set
    .... .... .0. = Syn: Not set
    .... .... ..0 = Fin: Not set
  window size: 0
  Checksum: 0xfcfe7 [validation disabled]
  [SEQ/ACK analysis]

0000  00 00 0c 07 ac 34 00 06 1b d3 25 19 08 00 45 00  ....4..%...E.
0010  00 28 23 9f 40 00 80 06 c4 86 82 e6 34 8b d9 46  .(#@...4..F
0020  81 f2 04 1b 00 50 84 70 a6 aa 2a 6e 46 4e 50 10  .....P.p..*nFNP.
0030  90 0d fc e7 00 00  ....
```

Flow Control in TCP (cont'd)

- The sender's Keep-Alive segment (Seq = ACK - 1 = 1051035)
 - To confirm that the idle connection is still active, the sender sends a probe segment designed to elicit a response from the receiver



```
1214 1.483208 217.70.129.242 130.230.52.139 TCP [TCP Keep-Alive] http > optima-vnet [ACK] Seq=1051035 Ack=584 Win=6996 Len=0
  Frame 1214: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
  Ethernet II, Src: Cisco_18:4e:00 (00:0f:34:18:4e:00), Dst: Notebook_d3:25:19 (00:06:1b:d3:25:19)
  Internet Protocol, Src: 217.70.129.242 (217.70.129.242), Dst: 130.230.52.139 (130.230.52.139)
  Transmission Control Protocol, Src Port: http (80), Dst Port: optima-vnet (1051), Seq: 1051035, Ack: 584,
    source port: http (80)
    Destination port: optima-vnet (1051)
    [Stream index: 0]
    Sequence number: 1051035 (relative sequence number)
    Acknowledgement number: 584 (relative ack number)
    Header length: 20 bytes
    Flags: 0x10 (ACK)
      000. .... = Reserved: Not set
      ...0 ... = Nonce: Not set
      ... 0... = Congestion window Reduced (CWR): Not set
      .... 0.. = ECN-Echo: Not set
      .... ..0. = Urgent: Not set
      .... ...1 ... = Acknowledgement: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
    Window size: 6996
    Checksum: 0xe194 [validation disabled]
    [SEQ/ACK analysis]
```

```
0000  00 06 1b d3 25 19 00 0f 34 18 4e 00 08 00 45 00  ....%... 4.N...E.
0010  00 28 f0 c8 40 00 2c 06 4b 5d d9 46 81 f2 82 e6  :(.@...K).F...
0020  34 8b 00 50 04 1b 2a 6e 46 40 84 70 a6 aa 50 10  4.P...n FM.p.P.
0030  1b 54 e1 94 00 00 00 00 00 00 00 00 00 00 00  T.....
```

- Consider what can happen if a receiving application process reads incoming data 1 byte (or a few in the general case) at a time:
 - ① If the sending process generates data quickly, the sending TCP will transmit a window full of data segments
 - ② Eventually, the sender will receive an ACK that specifies that no additional space remains in the receive buffer ($rwnd=0$)
 - ③ When the receiving process reads 1 byte of data from a full buffer, 1 byte of space becomes available
 - ④ Since space becomes available in its buffer, the receiving TCP generates an ACK that uses the Window field to inform the sender ($rwnd=1$)
 - ⑤ When the sending TCP learns that space is available, it responds by transmitting a segment that contains 1 byte of data
- Although single-byte window advertisements work correctly to keep the receive buffer filled, they result in a series of tiny data segments

- What is wrong with small data segments?
- The sending TCP must compose a segment that contains 1 byte of data, place the segment in an IP packet, and transmit the result
- When the receiving application reads another byte, TCP generates another ACK, which causes the sender to transmit another segment that contains 1 byte of data and so on
- The resulting interaction can reach a steady state in which TCP sends a separate segment for each byte of data

- The transmission of small segments consumes unnecessary network bandwidth because each IP packet carries only 1 byte of data
 - Unacceptable protocol overhead (IP+TCP headers/user data): 40/1 (tiny segments) vs. 40/1460 (full-sized segments)
- Moreover, transferring small segments introduces unnecessary computational overhead at the hosts and intermediate systems along the path
 - Due to processing segments and recomputing their checksums/CRCs
- The above example describes how small segments result when a receiver advertises a small window

- However, a sender can also cause each segment to contain a small amount of data:
 - Firstly, TCP can create and transmit a separate segment each time the application process generates 1 byte of data
 - TCP can also send a small segment if the application process generates data in fixed-sized blocks of N bytes, and the sending TCP extracts data from the buffer in maximum segment sized blocks, M , where $M < N$, so the remaining block in the send buffer can be small
- Known as silly window syndrome, the problem of tiny segments appeared in early TCP implementations
- **Silly window syndrome (SWS)** – each ACK advertises a small amount of space available and each segment carries a small amount of data

- TCP specification (RFC 1122) now includes heuristics that prevent silly window syndrome:
 - One heuristic used on the sending host avoids transmitting small amount of data in each segment
 - Another heuristic used on the receiving host avoids sending small increments in window advertisements that can trigger small data packets
- In practice, TCP software must contain both sender and receiver silly window avoidance code
 - Since TCP connections are full-duplex

- **Receive-side SWS avoidance**
- The receive-side SWS avoidance algorithm prevents small window advertisements in case where a receiving application process extracts data bytes slowly
 - E.g., when the receive buffer fills completely, the receiver sends an ACK that contains a zero window advertisement
 - As the receiving application extracts bytes from the buffer, the receiving TCP computes the newly available space in the buffer
- Instead of sending a window advertisement immediately, **the receiver waits until the available space reaches either 50% of the total buffer size or a maximum-sized segment**

- **Send-side SWS avoidance**
- To avoid sending small segments, a sending TCP must allow the sending application to make multiple calls to write, and must collect the data transferred in each call before transmitting it in a single, large segment
 - This technique is known as **clumping**
- How long should TCP wait before transmitting data?
- On the one hand, if TCP waits too long, the application experiences large delays
 - A fixed delay is not optimal for all application
- On the other hand, if TCP does not wait long enough, segments will be small and throughput will be low

- **Nagle algorithm**

- Named after its inventor, John Nagle
- When a sending application generates additional data to be sent over a connection for which previous data has been transmitted but not yet acknowledged, place the new data in the send buffer as usual, but do not send additional segments until there is sufficient data to fill a maximum-sized segment
 - For IPv4, $MSS = MTU - 40$ bytes
 - For IPv6, $MSS = MTU - 60$ bytes
- If still waiting to send when an ACK arrives, send all data that has accumulated in the send buffer
 - I.e., TCP uses the arrival of an ACK to trigger the transmission of accumulated data

- The Nagle algorithm is **self-clocking**:
 - If an application process generates data 1 byte at a time, TCP will send the first byte immediately
 - Until an ACK arrives or the number of accumulated bytes is less than the MSS, TCP will accumulate additional bytes in its send buffer
- The Nagle algorithm adapts to arbitrary combinations of the network and application speeds:
 - If the application process is fast compared to the network (i.e., a bulk data transfer), successive segments will contain many bytes
 - If the application process is slow compared to the network (e.g., a user typing on a keyboard), small segments will be sent without long delay

- The ideal TCP connection rapidly increases its sliding window and keeps the path between the sender and receiver full of packets at any time
- Keeping the path full of packets requires both rwnd and cwnd to reach the capacity of the transmission path ('pipe') at once
 - Since $\text{window} = \min(\text{rwnd}, \text{cwnd})$
- **Capacity = bandwidth (in bits/s) * Round-Trip Time (in sec.)**
 - Aka **the bandwidth-delay product (BDP)** and is measured in bits/bytes/packets/etc.
- The BDP is twice as large as the bit length of the transmission path, since instead of the one-way propagation time we use the two-way propagation delay – **the Round-Trip Time (RTT)**

BDP (cont'd)

- For long, modern, high-speed links, the BDP can be much greater than the maximum TCP receive window size of 65,535 bytes
- The maximum throughput of a TCP connection is limited to rwnd/RTT
- **The upper bound on maximum link utilization** is the maximum window size divided by the BDP
- **Link utilization = [rwnd / bandwidth-delay product] * 100%**

BDP (cont'd)

- Consider a 1000 km fiber link has a 5 ms one-way delay
 - The velocity of signal propagation in optical fiber is about 200,000 km/s
- The RTT (i.e., the two-way propagation delay) = $2 * 5 \text{ ms} = 10 \text{ ms}$
- When operating at 10 Gbits/s, the BDP = $100 * 10^6$ bits or $12.5 * 10^6$ bytes
- The upper bound on the link utilization is

$$\frac{\text{rwnd}}{\text{BDP}} * 100\% = \frac{65,535}{12.5 * 10^6} * 100\% = 0.52\%$$

- To improve efficiency, the receive window size should be increased

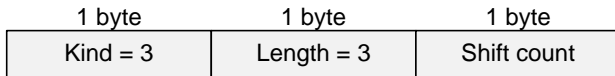
- RFC 1323 'TCP Extensions for High Performance'
 - It defines a new extension for TCP – **TCP Window Scale option** – that permits significantly larger (up to 1 GB) windows to be advertised and utilized
- Both sides must send Window Scale options in their SYN segments to enable window scaling in either direction
 - A Window Scale option in a segment without SYN=1 should be ignored
- This negotiation has 2 purposes:
 - To indicate that the TCP entity is prepared to do both send and receive window scaling
 - To communicate a **scale factor** to be applied to its receive window

Scaling (cont'd)

- **The shift count** indicates how many bits to the left to shift the value in the Window field, to arrive at the actual window size
- For instance:
 - A shift count of 0 multiplies the stated window size by 1 ($2^0 = 1$), so no scaling performed
 - A shift count of 5 multiplies the stated window size by 32 ($2^5 = 32$)
- The shift count is limited to 14, which allows the rwnd of

$$65,535 * 2^{14} = 1,073,725,440 \text{ bytes} \approx 1 \text{ GB}$$

- The 3-byte TCP Window Scale option:



Scaling (cont'd)

- The maximum rwnd size (by default):
 - Microsoft Windows XP: 65,535 bytes
 - Microsoft Windows 7: $65,535 * 2^8 = 16,776,960$ bytes
- TCP optimizers and tweakers:
 - SG TCP Optimizer (freeware): www.speedguide.net/downloads.php
 - Ashampoo WinOptimizer (commercial)
 - Auslogics BootSpeed (commercial)