# Congestion Control

Roman Dunaytsev

The Bonch-Bruevich Saint-Petersburg
State University of Telecommunications

roman.dunaytsev@spbgut.ru

Lecture № 9

# Outline

# Outline

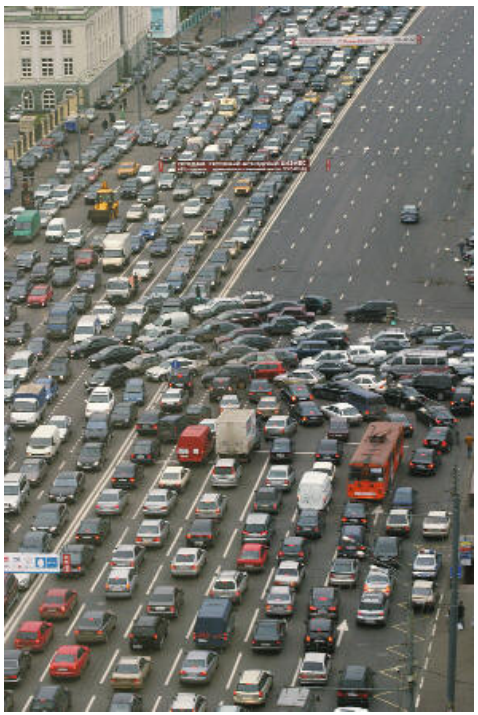
# Introduction

- **Congestion** – the state of a network in which the offered load exceeds the network capacity for a certain period of time
- Under congestion conditions, network performance becomes worse:
  - Packets get lost
  - Delays and delay variation (aka **delay jitter** or **jitter**) increase
  - Local and network resources are wasted
  - **Quality of Experience (QoE)** degrades
- Delay jitter is an important QoS parameter for real-time applications such as voice over IP (VoIP), video on demand (VoD), etc.

# Introduction (cont'd)

- **Congestion occurs when resource demands exceed the capacity**
- Network congestion as **traffic jams** due to:

- Fast rerouting in case of a failure



- Bottleneck

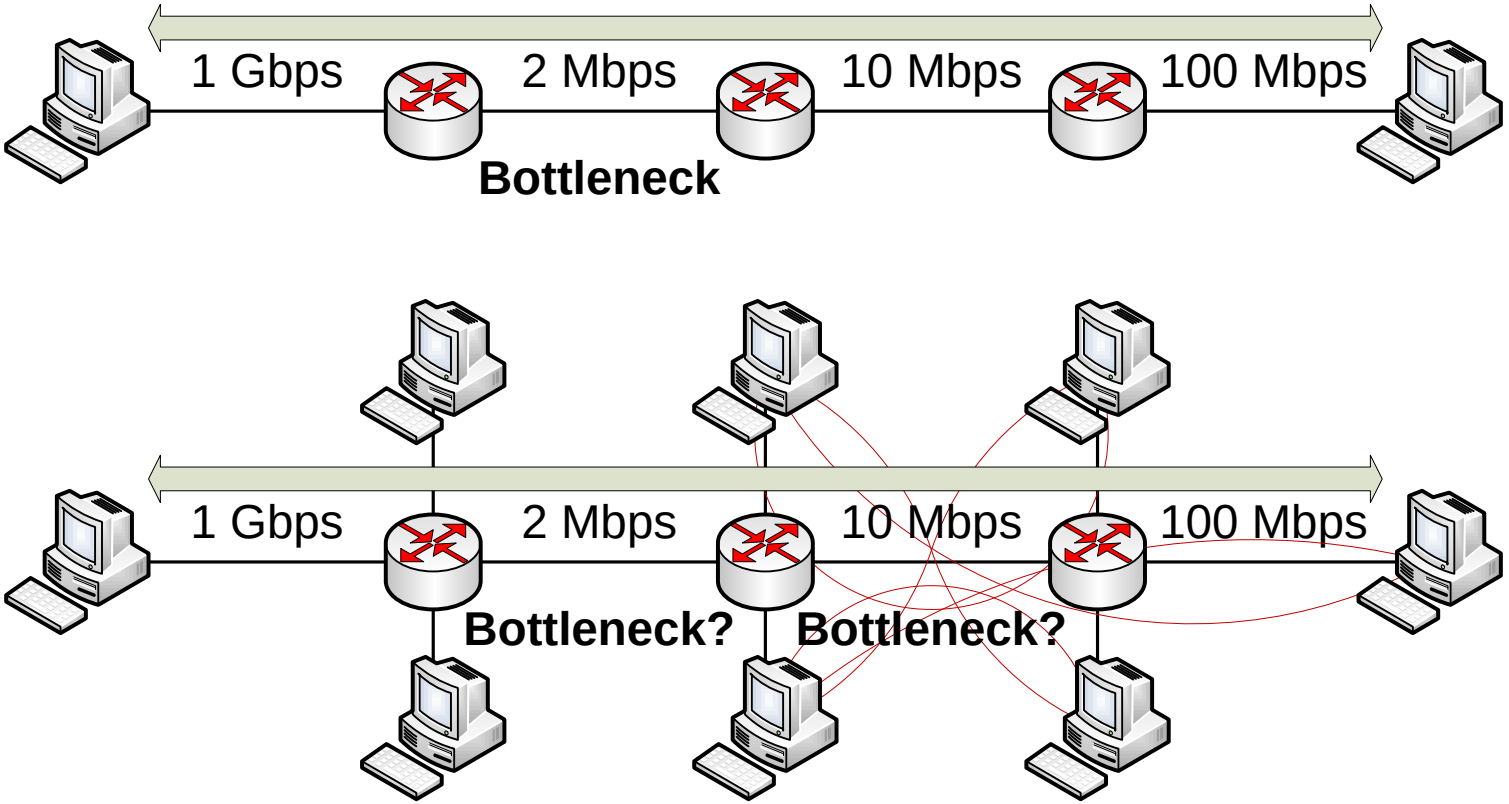
- Cross-traffic

# Introduction (cont'd)

- In engineering, **bottleneck** – a phenomenon by which the performance or capacity of an entire system is severely limited by a single component
- In computer networks, a bottleneck limits **the maximum achievable throughput**

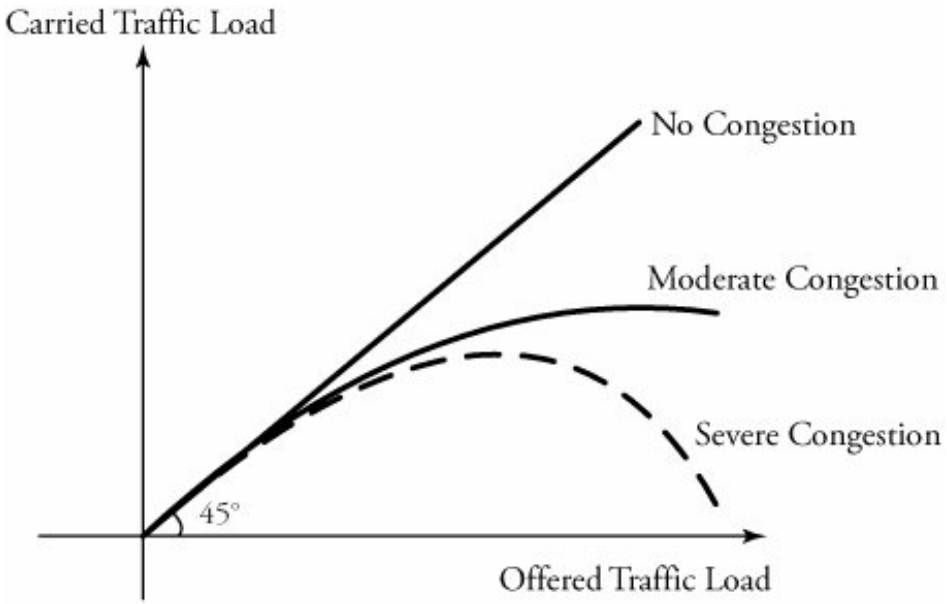
# Introduction (cont'd)

- Identifying actual and potential bottlenecks is a nontrivial task



# Introduction (cont'd)

- **Congestion collapse** – the situation in which an increase in the offered load results in a decrease in the number of packets delivered successfully by the network
- Congestion collapse arises when capacity is wasted sending packets through the network that are dropped before they reach their final destination



# Introduction (cont'd)

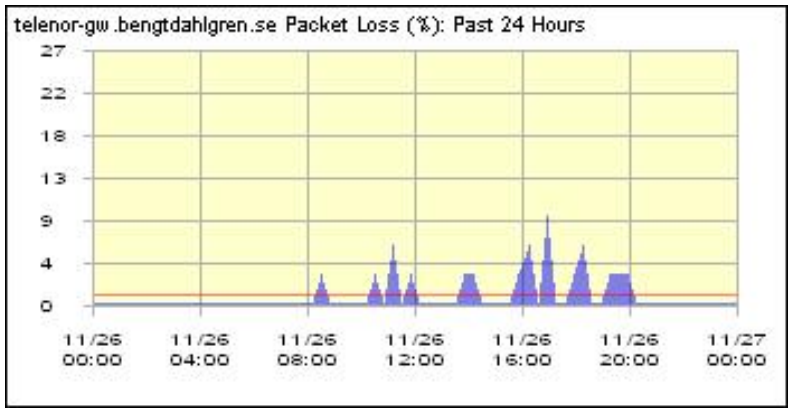
- In the early days of the Internet, TCP did not have effective congestion control algorithms, and the result was networkwide congestion collapse on several occasions in the mid-1980s
  - Congestion collapse begins with a steady increase in the load on the network
  - As hosts send more packets, more packets are queued in the buffers of the routers
  - Increased delays and lost data induced by congestion lead to timeouts, which trigger retransmissions
  - A large number of retransmissions overload the network even further
  - As a result, the network throughput drops to a small fraction of its normal capacity and this condition is stable
- See RFC 896 'Congestion Control in IP/TCP Internetworks'

# Introduction (cont'd)

- As users come and go, so do the data flows they initiate
- Network performance is largely governed by these **fluctuations**
- The Internet Traffic Report monitors the flow of data around the world: www.internettrafficreport.com
  - E.g., router telenor-gw.bengtdahlgren.se (Gothenburg, Sweden)

- Response time


- Packet loss



# Outline

# Overprovisioning

- **2 approaches to avoid congestion**:
  - Overprovisioning
  - Congestion control
- *G. Finnie, 'ISP Traffic Management Technologies: The State of the Art,' Report ISBN 978-1-100-11839-0, January 2009*
  - On behalf of the Canadian Radio Television and Telecommunications Commission
- **Nowadays, the common choice of Internet Service Providers (ISPs) is overprovisioning**
- The overprovisioning ratio varies widely and depends on:
  - The underlying network topology and technology
  - The volume of traffic and anticipated variation in traffic loads
  - The number of users and the kind of users
  - The mix of applications

# Overprovisioning (cont'd)

- The reasons for overprovisioning are of a purely financial nature:
  - Bandwidth has become cheap
  - It is more difficult to control a network that does not have enough bandwidth than an overprovisioned one
  - With an overprovisioned network, an ISP is prepared for the future
- However, adding extra bandwidth cannot solve the problem alone
- In fact, overprovisioning works in combination with TCP congestion control mechanisms

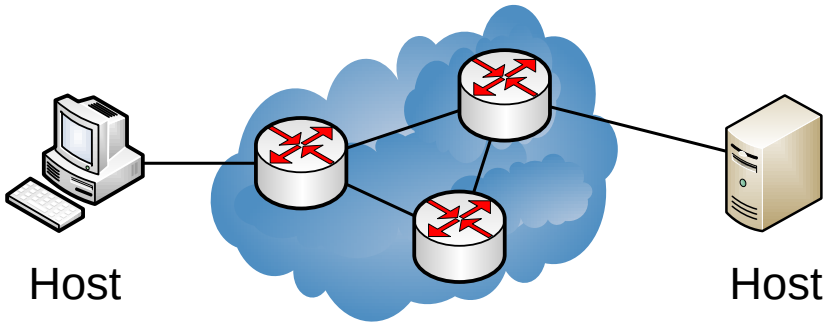
# Outline

# Congestion Control

- The goal of congestion control is to prevent applications from either overloading or underutilizing the available network resources
  - I.e., to achieve the highest possible throughput while maintaining a low packet loss rate and a small delay
- Congestion must be avoided because it leads to queue growth, while queue growth leads to delay and loss
  - Thereby, the term '**congestion avoidance**' is sometimes used instead
- Storing packets in a queue adds significant delays, depending on the length of the queue
  - Thus, queues should be kept short
- Moreover, packet loss can occur no matter how large the buffer is

# Congestion Control (cont'd)

- Congestion control should be distinguished from flow control
- **Flow control** – prevents the sender from overloading the receiver
- **Congestion control** – prevents the sender from overloading the network
  - Congestion control is relevant to multihop networks, where end systems (hosts) are connected by a series of intermediate systems (e.g., routers)
- Congestion control is tightly linked with flow control, error control, and QoS provisioning



# Congestion Control (cont'd)

- **2 methods to control the sending rate**:
  - Rate-based control
  - Window-based control
- **Rate-based control** – the sender is aware of a specific data rate, and the receiver or a intermediate system informs the sender of a new rate that it must not exceed
- **Benefits** of rate-based congestion control:
  - Streaming media applications work best when they are able to sustain a constant or smoothly varying data rate over a relatively long period of time
- **Shortcomings** of rate-based congestion control:
  - Complexity

# Congestion Control (cont'd)

- **Window-based control** – the sender keeps track of the window – – a certain amount of data that it is allowed to send before new feedback arrives
  - With each PDU sent, the window is decreased until it reaches 0
  - The receiver accepts and counts incoming PDUs and informs the sender that it is allowed to increase the window by a certain amount
  - Since the sender's behavior is very strictly dictated by the presence or absence of incoming feedback, window-based control is said to be **self-clocking**
- **Benefits** of window-based congestion control:
  - Network stability, since a new packet is not put into the network until an old packet leaves
- **Shortcomings** of window-based congestion control:
  - Burstiness because packets are put into the network in bursts

# Congestion Control (cont'd)

- **2 types of control systems**:
  - Open-loop control systems
  - Closed-loop control systems
- **Open-loop congestion control** – implies use of **a priori** knowledge about the network
  - E.g., the available bandwidth along a certain route
- A network that is solely based on open-loop control would use **resource reservation**
- Thus, a new flow would only enter if the **Connection Admission Control (CAC)** entity allows it to do so
  - This is how congestion has always been dealt with in the PSTN: when a user wants to call somebody but the network is overloaded, the call is simply rejected

# Congestion Control (cont'd)

- **Closed-loop congestion control** – the sender tunes its data rate based on the **feedback** from the receiver

- **2 types of feedback**:
  - Implicit
  - Explicit

- **Implicit feedback** is based on end-to-end behavior analysis without any explicit help from the network

- **Explicit feedback** is based on messages generated by the network and sent to the end system to indicate a congestion condition

# Congestion Control (cont'd)

- **Implicit congestion control** – a scheme under which **end systems** first detect a possible congestion condition by means other than explicit congestion messages, and then take appropriate action to reduce their rate
  - An example of implicit congestion control is using packet loss (or excessively delayed packets) as a means of detecting congestion in TCP
- **Benefits** of implicit congestion control:
  - Simplicity, since it does not require explicit feedback from the network
- **Shortcomings** of implicit congestion control:
  - Interpreting packet loss as a sign of congestion only works well if this is truly the main reason for packets to be dropped
  - Solely relying on packet drops means that sources always need to increase their rates until queues are full, that is, we have a form of congestion control that first causes congestion and then reacts

# Congestion Control (cont'd)

- What **bad** can happen to a packet as it travels from source to destination?

1. It can be excessively delayed
   - Due to queuing or retransmissions at the link layer
2. It can be dropped
   - Due to buffer overflow, rejected by admission control, mistrouted, equipment malfunctions, checksum fails, etc.
3. It can be duplicated
   - Due to equipment malfunctions

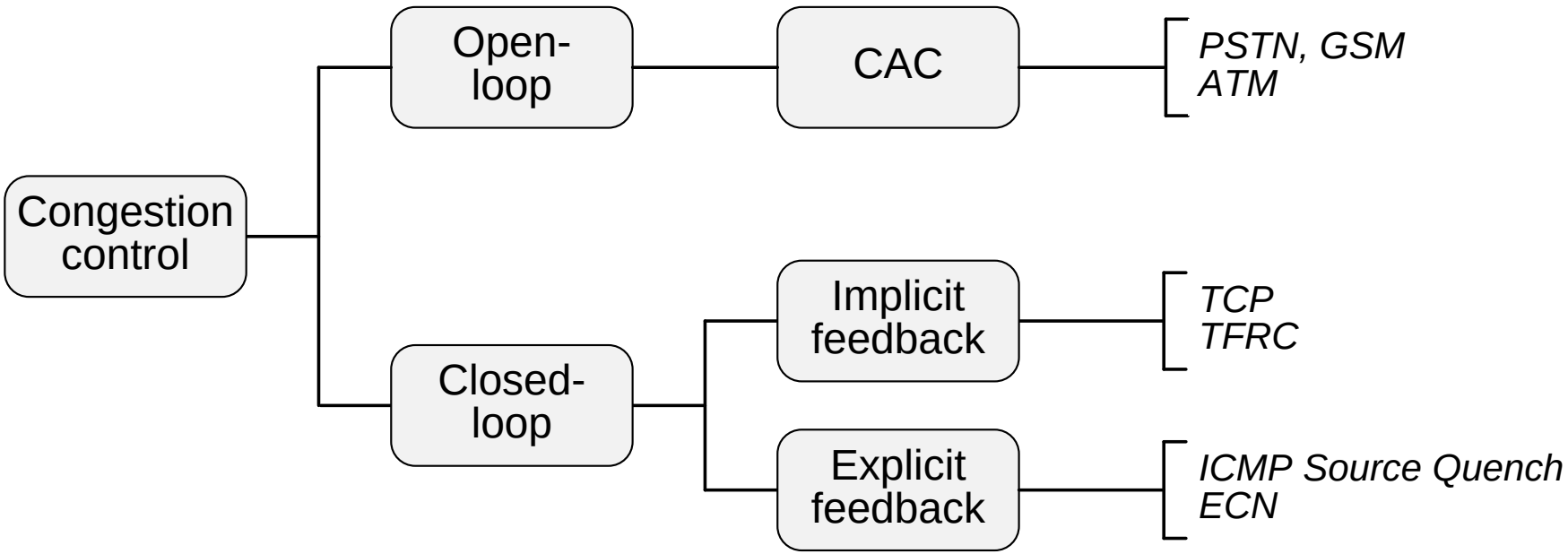
- Each of these things can happen due to a variety of reasons, not only as a result of congestion

# Congestion Control (cont'd)

- **Explicit congestion control** – a scheme under which end systems rely on information from **the network** about impending and current congestion
  - An example of explicit congestion control is sending Source Quench messages in ICMP
- **Benefits** of explicit congestion control:
  - Accuracy
- **Shortcomings** of explicit congestion control:
  - Complexity

# Congestion Control (cont'd)

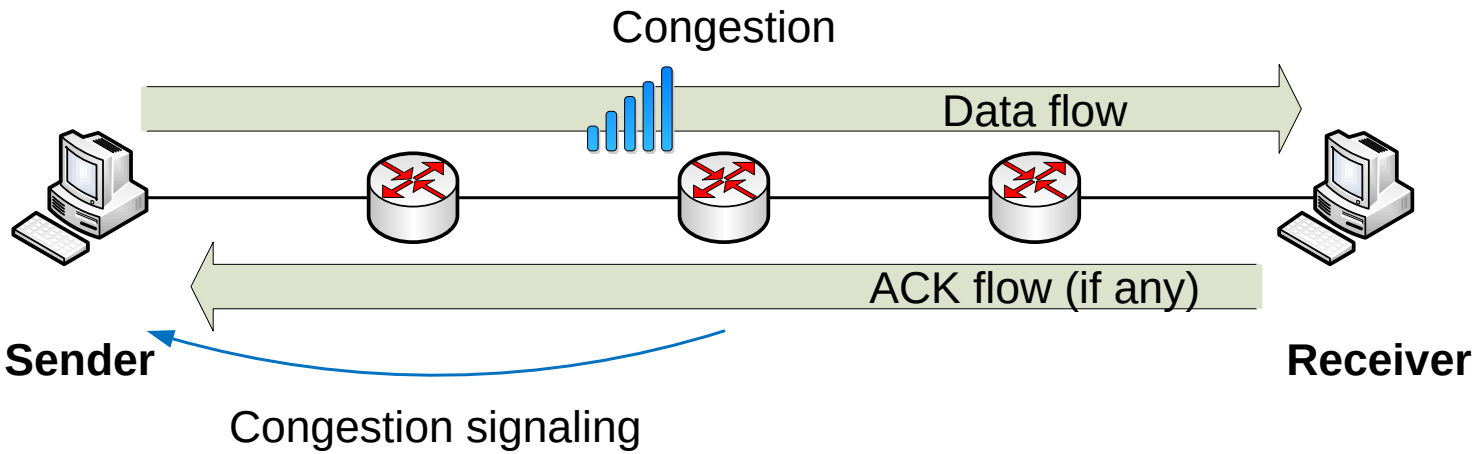
- Taxonomy of congestion control approaches



# Outline

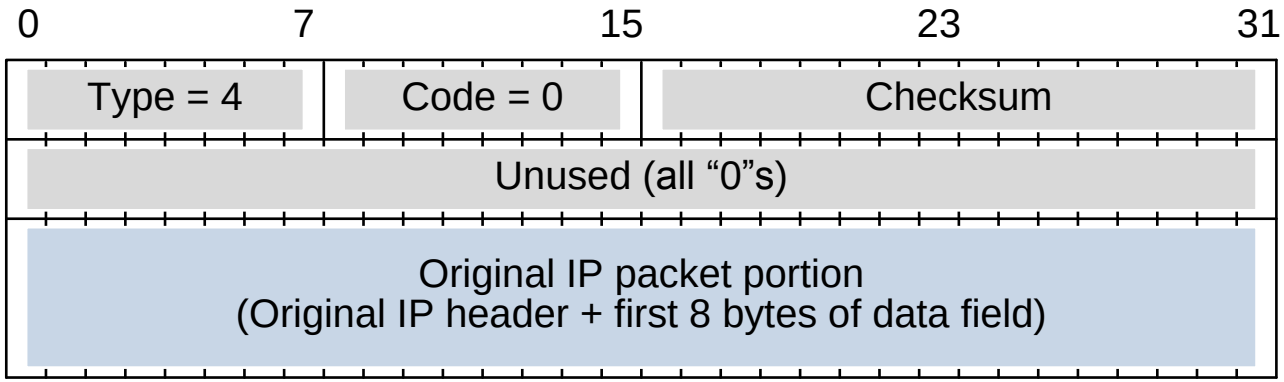
# ICMP Source Quench

- The first congestion control scheme for the Internet was proposed by John Nagle in 1984 and is known as **Source Quench**
- In IPv4, a device that is forced to drop packets due to congestion provides feedback to the senders that overwhelmed it by sending them ICMPv4 Source Quench messages
- When a sender receives one of these messages it knows that it needs to decrease its sending rate



# ICMP Source Quench (cont'd)

- **The Internet Control Message Protocol (ICMP)** was first specified in RFC 792
  - ICMP is the protocol that handles error and other control messages
  - It is used by both end systems and intermediate systems
  - ICMP messages are encapsulated into IP packets
- ICMPv4 Source Quench message format:



# ICMP Source Quench (cont'd)

- There are no rules about when and how a device generates Source Quench messages
  - Typically, 1 message is generated for each dropped packet
  - Devices may decide whether to quench all senders or only certain ones
- There is also no method for the device to signal a sender it has 'quenched' that it is no longer congested and to resume its prior sending rate
  - Usually, a sender will reduce its rate until it no longer receives the messages any more, and then may try to slowly increase the rate again
- The mechanism to respond to Source Quench may be in the transport layer or in the application layer
  - The recommended procedure for TCP is to trigger slow start as if a retransmission timeout had occurred

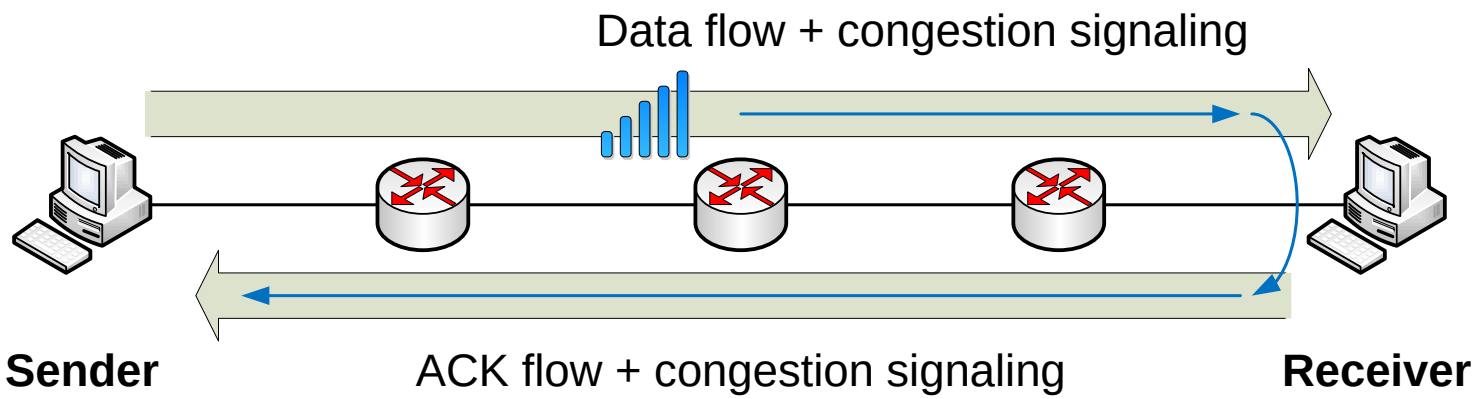
# ICMP Source Quench (cont'd)

- **Benefits** of ICMP Source Quench:
  - It is the fastest method to notify senders that they should reduce their rate
  - The feedback is explicit and, thus, more or less precise
- **Shortcomings** of ICMP Source Quench:
  - The consumption of bandwidth on the reverse path
  - The use of router resources (memory and CPU power) while the router is overloaded
  - The lack of information communicated in Source Quench messages makes them a rather crude tool for managing congestion
- As specified in RFC 1812, **generating source quench messages is not recommended behavior for routers anymore**

# Outline

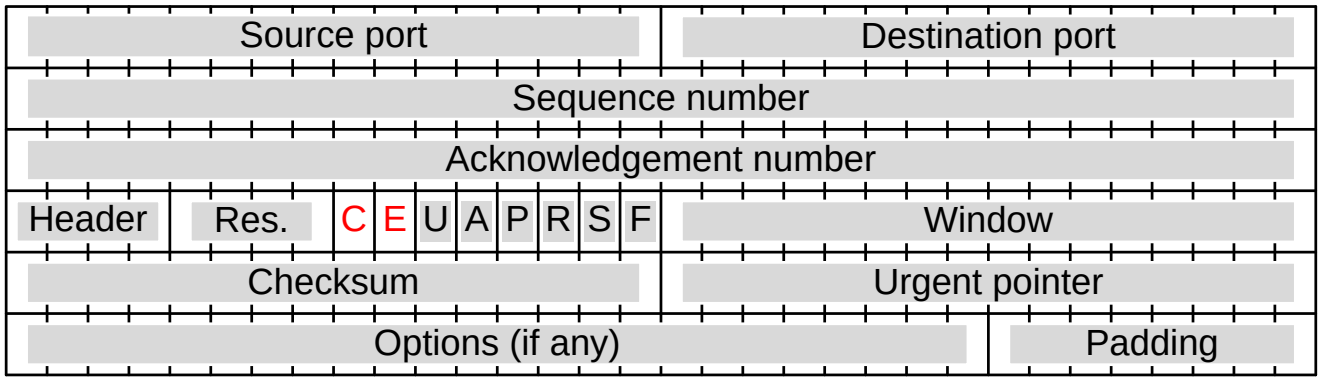
# ECN

- **Explicit Congestion Notification (ECN)** is defined in RFC 3168
- ECN provides a method for an intermediate router to notify the hosts of impending network congestion
- In the Internet, the use of **active queue management (AQM)** allows detection of congestion before the router buffer overflows
- The ECN scheme enhances the AQM behavior by forcing the router to **mark** the packets instead of dropping them



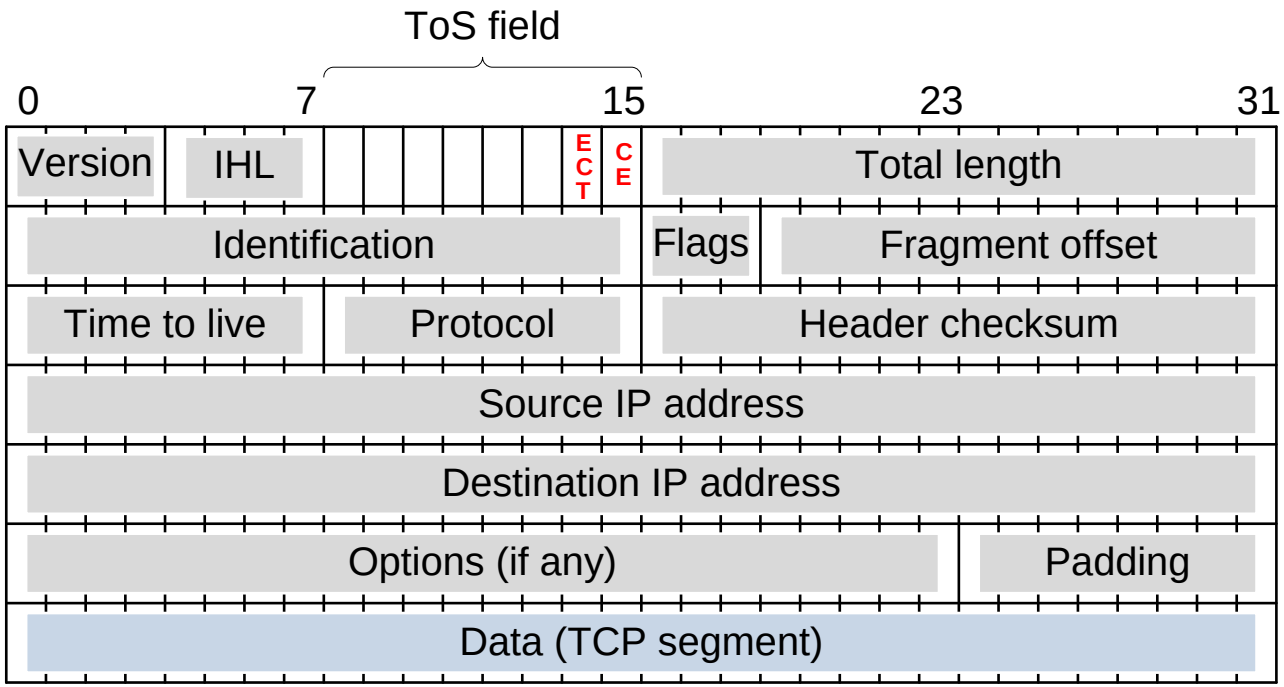
# ECN (cont'd)

- ECN support in TCP uses 2 bits in the TCP header that were previously defined as the Reserved field
- **CWR (Congestion Window Reduced)**
  - Used by the sending host to indicate that it has received a TCP segment with the ECE flag set to 1
- **ECE (ECN-Echo)**
  - Used during connection setup to indicate that a host is ECN-capable
  - Used later during data transfer to indicate that a TCP segment was received with the ECN field in the IP header set to 11



# ECN (cont'd)

- ECN support in IP uses 2 bits of the Type of Service (ToS) field:
- **ECN-Capable Transport (ECT)**
  - It is set by the source and indicates whether the transport connection supports ECN
- **Congestion Experienced (CE)**
  - It is set by a router and indicates that congestion has been encountered



# ECN (cont'd)

- **ECN works as follows**:
- During connection setup, both TCP entities exchange information about their willingness to use ECN
  - The host that performs an active open sets the ECE and CWR flags in the TCP header of the SYN segment
  - The host that does a passive open sets only the ECE flag in the TCP header of the SYN/ACK segment
- During data transfer, the ECN-capable hosts send packets with the ECT and CE flags in the IP header set to either 10 or 01
- If ECN is not in use and congestion is detected at an intermediate router, packets are dropped as usual

# ECN (cont'd)

- If congestion is detected at ECN-capable routers, then packets are not dropped unless the congestion is very severe
  - Instead, a router that has congestion imminent sets the ECT and CE flags in the IP header to 11
- When the receiving host receives this packet, it sets the ECE flag in the TCP header and continues setting this flag in subsequent ACKs
- When the sending host receives the ACK with the ECE flag, it acts as if a single packet has been dropped
  - It triggers the congestion avoidance algorithm and halves the cwnd size
- Then the sending host sets the CWR flag in the TCP header of the next segment in order to acknowledge the reception of the congestion notification
- On receipt of the segment with the CWR flag set on, the receiving host stops setting the ECE flag in subsequent ACKs

# ECN (cont'd)

- **Benefits** of ECN:
  - ECN prevents TCP connections from unnecessary packet drops or retransmission timeouts
- **Shortcomings** of ECN:
  - ECN requires AQM to be in place
  - The potential improvement in TCP performance can only be achieved when both source and destination hosts support ECN and ECN-capable routers are deployed along the path
  - A misbehaving receiver could easily inform a sender that everything was all right by just ignoring the CE bit; this would lead the sender to unfairly increase its cwnd further when it should actually be reduced

# ECN (cont'd)

- Many modern operating systems support ECN
- However, it is usually disabled by default
- As a result, less than 1% of today's Internet traffic is ECN-capable
- Internet2 NetFlow: Weekly Reports (April 26, 2010)

| Type | Octets | | Packets | |
|---|---|---|---|---|
| ECN-Capable | 0.26% | 4.717T | 0.15% | 3.549G |

- *D. Murray, T. Koziniec, 'The state of enterprise network traffic in 2012,' 18th Asia-Pacific Conference on Communications, pp. 179-184, October 2012*
  - 0% of TCP connections requested ECN during the three-way handshake
  - 0.07% of TCP packets were marked as ECN-capable
  - 0.0007% of TCP packets were marked with CE in the IP header

# Outline

# Congestion Control in TCP

- Most Internet applications use TCP for error-free data delivery and proper sequencing
- As a result, **TCP controls a large fraction of bytes and packets traversed over the Internet**

- *H. Schulze, K. Mochalski, 'Internet Study 2008/2009,' ipoque GmbH*

| Protocol Class | Southern Africa | South America | Eastern Europe | Northern Africa | Germany | Southern Europe | Middle East | South-western Europe |
|---|---|---|---|---|---|---|---|---|
| P2P | 65,77% | 65,21% | 69,95% | 42,51% | 52,79% | 55,12% | 44,77% | 54,46% |
| Web | 20,93% | 18,17% | 16,23% | 32,65% | 25,78% | 25,11% | 34,49% | 23,29% |
| Streaming | 5,83% | 7,81% | 7,34% | 8,72% | 7,17% | 9,55% | 4,64% | 10,14% |
| VoIP | 1,21% | 0,84% | 0,03% | 1,12% | 0,86% | 0,67% | 0,79% | 1,67% |
| IM | 0,04% | 0,06% | 0,00% | 0,02% | 0,16% | 0,03% | 0,50% | 0,08% |
| Tunnel | 0,16% | 0,10% | - | - | - | 0,09% | 2,74% | - |
| Standard | 1,31% | 0,49% | - | 0,89% | 4,89% | 0,52% | 1,83% | 1,23% |
| Gaming | - | 0,04% | - | - | 0,52% | 0,05% | 0,15% | - |
| Unknown | 4,76% | 7,29% | 6,45% | 14,09% | 7,84% | 8,86% | 10,09% | 9,13% |

- Internet2 NetFlow: Weekly Reports (April 26, 2010)

| Protocols | | Octets | | Packets |
|---|---|---|---|---|
| ICMP[1] | 0.10% | 1.745T | 0.42% | 10.20G |
| IGMP[2] | 0.00% | 52.09M | 0.00% | 1.403M |
| IP-ENCAP[4] | 0.01% | 242.2G | 0.01% | 208.3M |
| TCP[6] | 87.37% | 1.595P | 83.37% | 2.019T |
| UDP[17] | 11.97% | 218.4T | 15.52% | 375.8G |
| IPv6[41] | 0.07% | 1.196T | 0.11% | 2.696G |
| GRE[47] | 0.20% | 3.685T | 0.21% | 5.114G |
| ESP[50] | 0.28% | 5.131T | 0.34% | 8.255G |
| AX.25[93] | 0.00% | 19.80k | 0.00% | 300.0 |
| PIM[103] | 0.00% | 3.347G | 0.00% | 43.19M |
| IPMP[169] | 0.00% | 0.000 | 0.00% | 0.000 |
| Other | 0.01% | 207.1G | 0.02% | 452.9M |
| Total | 100.00% | 1.825P | 100.00% | 2.422T |

# Congestion Control in TCP (cont'd)

- The standard TCP congestion control is **reactive** window-based congestion control
  - It uses either packet losses or excessively delayed packets to trigger congestion alleviation actions
- In steady state, TCP congestion control works as follows:
  1. The sender increases the window size linearly until a packet loss occurs
  2. Once a packet loss is detected, the sender halves the window size
- Therefore, this rate control strategy is called **additive increase/multiplicative decrease (AIMD)**
- **The current stability of the Internet largely depends on TCP congestion control**

# Congestion Control in TCP (cont'd)

| **TCP function** | **Implementation** |
|---|---|
| Multiplexing/demultiplexing | Port numbers |
| Ordered data transfer and data segmentation | Connection establishment and termination<br>MSS option<br>Path MTU discovery |
| Error control | Checksum computation<br>Sequence numbers<br>Protection against wrapped sequences<br>Cumulative and selective ACKs<br>Retransmission timer and retransmissions |
| Flow control | Receive window<br>Silly window syndrome avoidance<br>Nagle algorithm<br>Window scale option |
| Congestion control | Karn's algorithm<br>Initial window<br>Slow start<br>Congestion avoidance<br>Fast retransmit and fast recovery<br>ECN-support |

# Karn's Algorithm

- TCP uses a retransmission timer to ensure data delivery in the absence of any feedback from the receiver
  1. When TCP sends a segment containing data, it retains a copy of the data in the retransmission queue and starts the retransmission timer
  2. If the data are not acknowledged before the retransmission timer expires, the data are retransmitted
  3. When the segment is acknowledged, TCP removes the data from the retransmission queue
- How to estimate this **Retransmission TimeOut (RTO)** value?
  - If the timeout is set too small, it will cause unnecessary retransmissions
  - If the timeout is set too big, it will lead to lengthy idle periods and late retransmissions

# Karn's Algorithm (cont'd)

- Because of the changing network conditions, RTO must be adjusted dynamically
- The algorithm used to compute and manage the retransmission timer is described in RFC 2988
- It is based on taking RTT samples
- **Round-Trip Time (RTT)** = the time from when a segment is sent until it is acknowledged
- Until the first RTT measurement has been made, the initial RTO value is set to 3 seconds

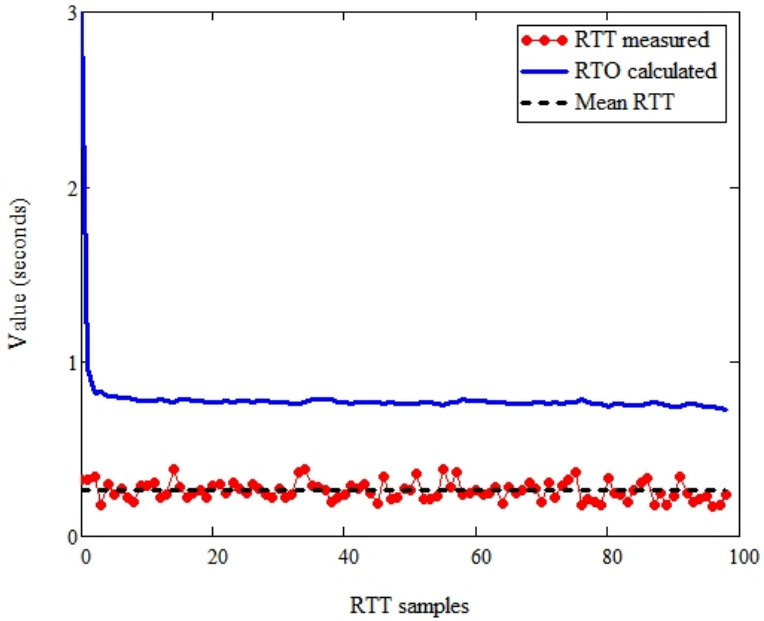
# Karn's Algorithm (cont'd)

- RTO can be approximated as a sum of the mean RTT and the additional term, which reflects the RTT variance
  - RTO = SRTT + max (clock granularity, 4∗RTTVAR)
  - SRTT = Smoothed Round-Trip Time
  - RTTVAR = Round-Trip Time Variation
- SRTT and RTTVAR are computed using **Exponentially Weighted Moving Average (EWMA)**
  - TCP puts more weight on the recent history and less weight on the last measurement
- Commonly, TCP implementations use a coarse-grained timer, having granularity of 500 ms, to measure the RTT and trigger the RTO, which imposes a large minimum value on the RTO

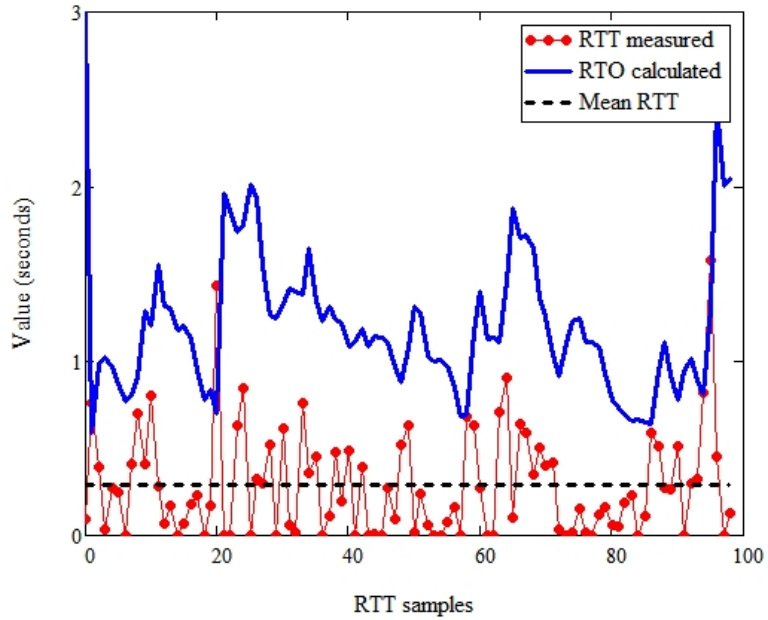
# Karn's Algorithm (cont'd)

- If the RTT has low variance, then the computed RTO value will be only slightly greater than the mean RTT
- If the RTT has high variance, then the computed RTO value will be much greater than the mean RTT

- Low RTT variance



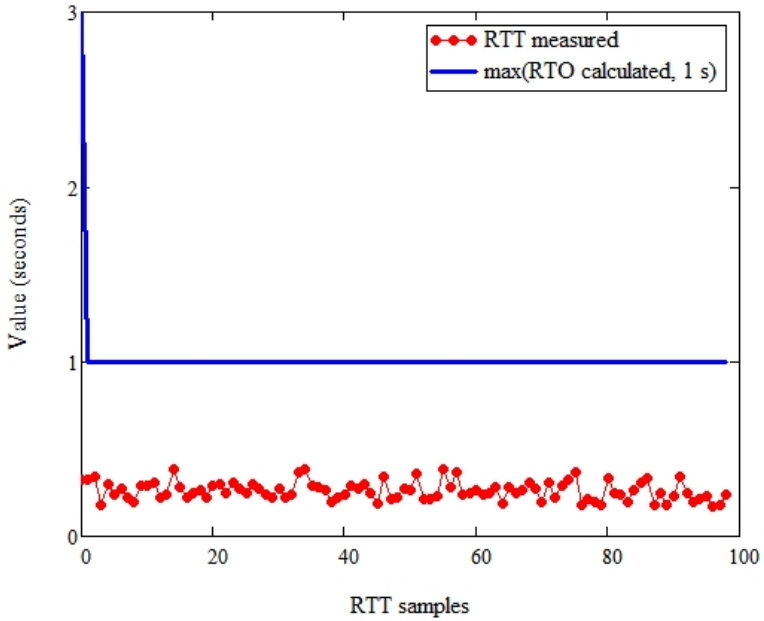

- High RTT variance



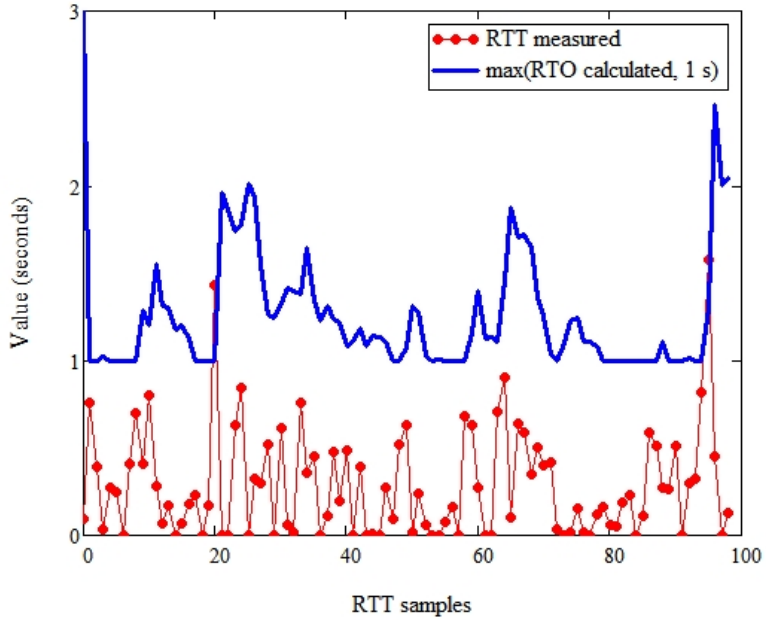
# Karn's Algorithm (cont'd)

- Whenever RTO is calculated, if it is less than 1 second then it should be rounded up to 1 second

- Low RTT variance

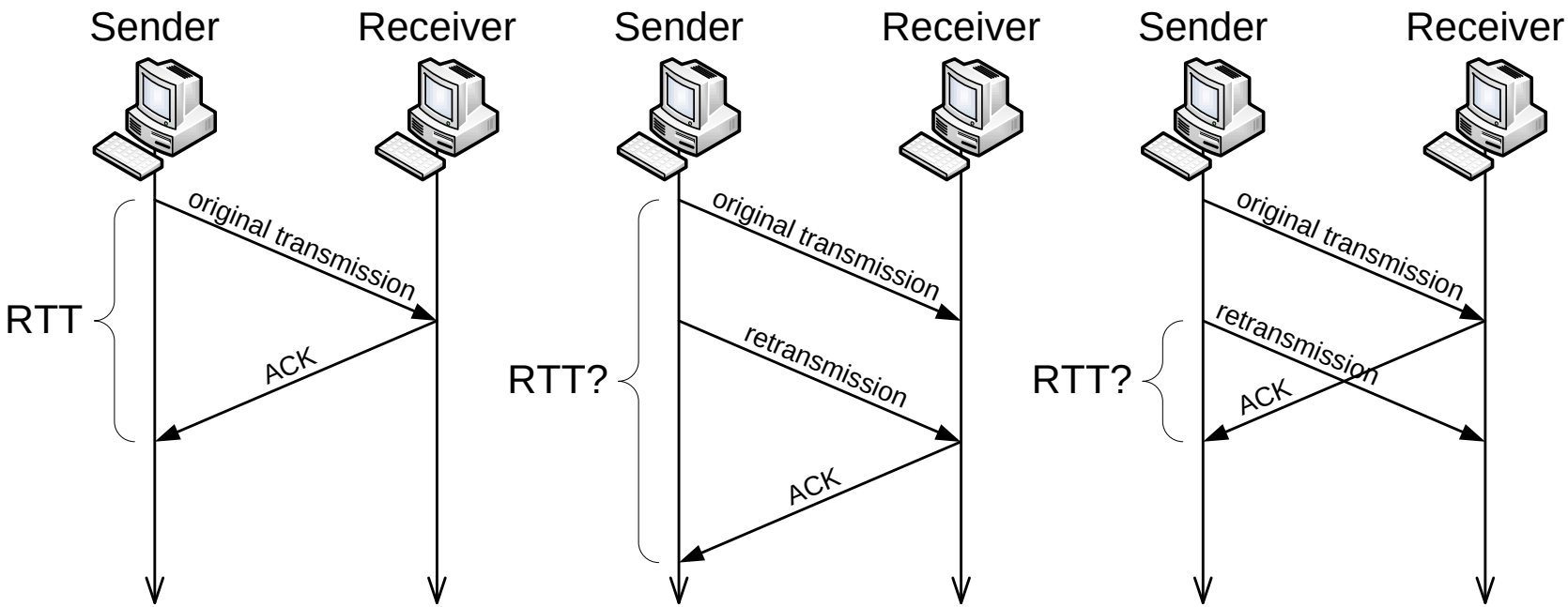



- High RTT variance



# Karn's Algorithm (cont'd)

- **The retransmission ambiguity problem** – a TCP sender's inability to distinguish whether the first ACK that arrives after a retransmission was sent in response to the original transmission or the retransmission
  - This problem occurs after a timeout-based retransmit and after a fast retransmit



# Karn's Algorithm (cont'd)

- How to ensure that ambiguous RTTs will not corrupt the calculation of RTO?
- The solution for taking RTT samples is based on the use of a technique called **Karn's algorithm**, after its inventor, Phil Karn:
  - Do not take into account the RTT sampled from segments that have been retransmitted
  - On successive retransmissions, set each timeout to twice the previous one
- This doubling (aka **the exponential backoff**) is repeated for each unsuccessful retransmission up to 6 times, after that RTO remains constant and equal to 64∗RTO
- RTO, 2∗RTO, 4∗RTO, 8∗RTO, 16∗RTO, 32∗RTO, 64∗RTO, 64∗RTO, 64∗RTO, ...

# Initial Window

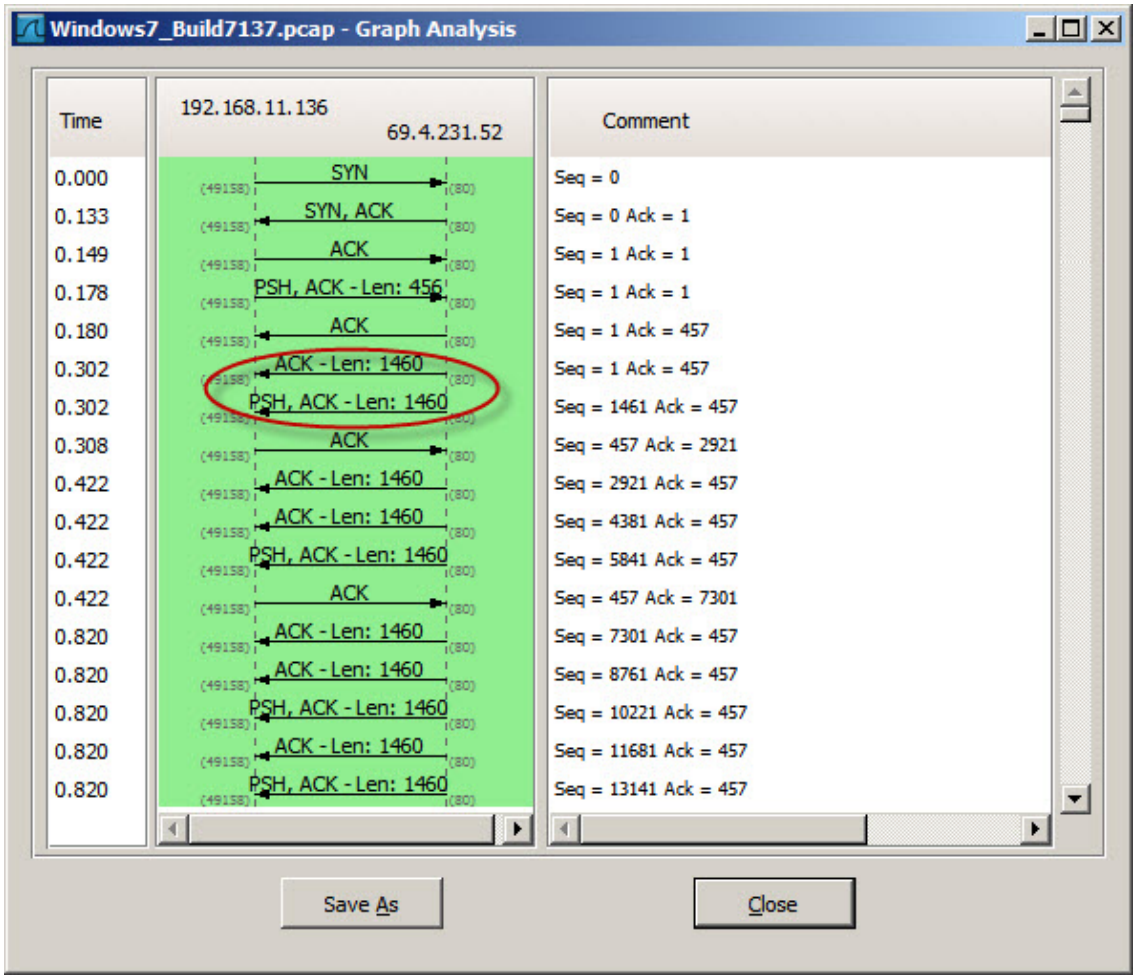
- The algorithms used by TCP to control the amount of data being injected into the network:
  - Slow start
  - Congestion avoidance
  - Fast retransmit and fast recovery (typically, used together)
- To implement these algorithms, 2 variables are added to the TCP per-connection state:
  - Slow start threshold (ssthresh)
  - Congestion window (cwnd)
- **ssthresh** is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission
- **cwnd** is a sender-side limit on the amount of data the sender can transmit into the network before receiving an ACK
  - At any time, **window = min (rwnd, cwnd)**

# Initial Window (cont'd)

- When a new TCP connection is established, the cwnd is set to **the initial window (IW)**
- In RFC 3390, the **maximum** IW size is defined as
  - **IW = min (4∗MSS, max (2∗MSS, 4380 bytes))**
- The valid range is from 1 to 4 full-sized segments
  - E.g., for MSS = 1460 bytes, the maximum IW size is 3 segments
- The majority of Web servers use an IW of 2 segments
- Default TCP settings:
  - Microsoft Windows XP: 1460 bytes
  - Microsoft Windows Server 2003: 4380 bytes
  - Microsoft Windows Server 2008: 2920 bytes
  - Microsoft Windows 7: 2920 bytes

# Initial Window (cont'd)

- Microsoft Windows 7 Ultimate
  - IW = 2920 bytes or 2 segments containing 1460 bytes of data

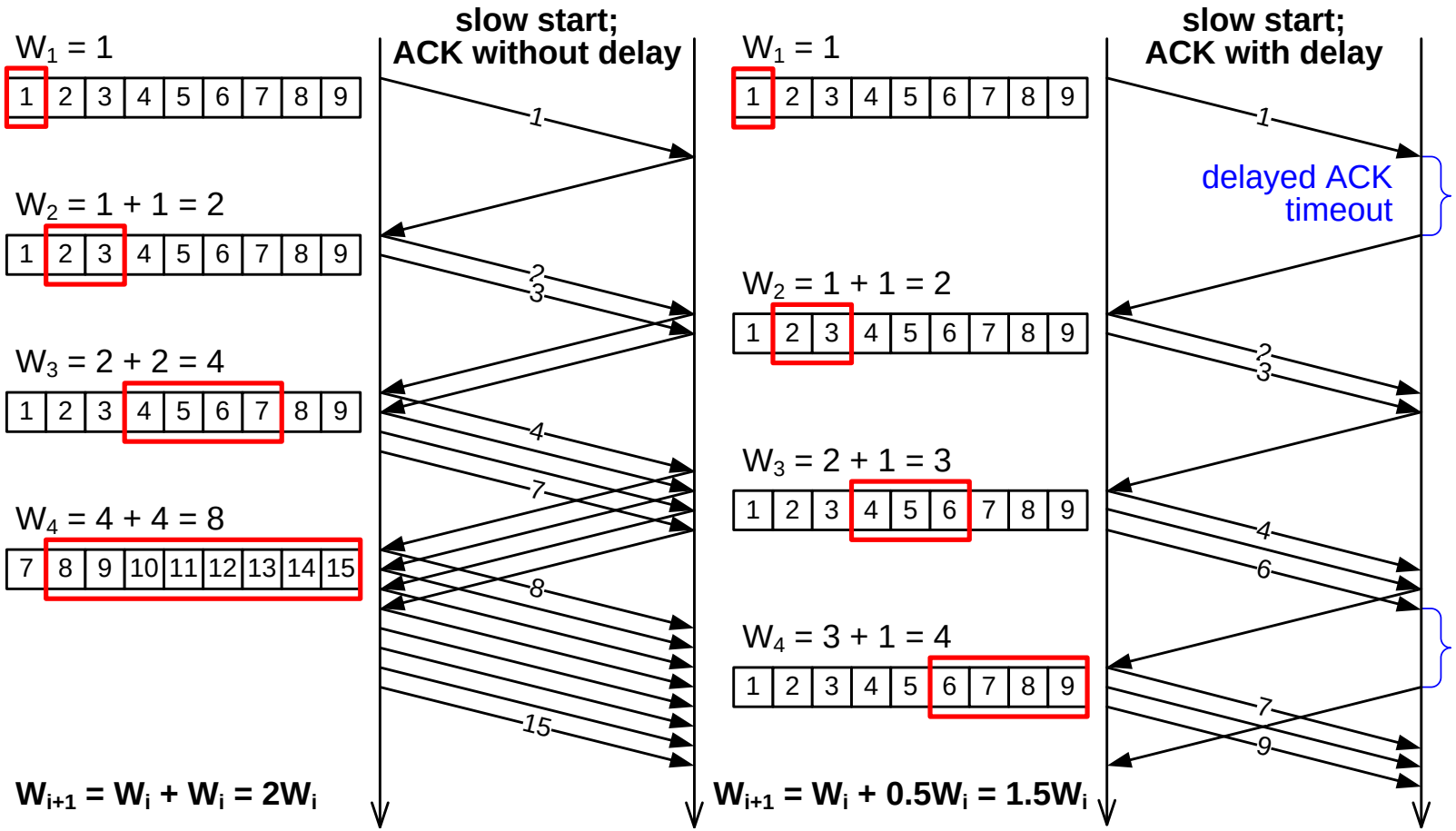


# Slow Start

- To probe the network path and to determine how much bandwidth is available, TCP uses an algorithm called **slow start**
- When a new TCP connection is established, the cwnd is set to the IW size and the sender starts transmitting data
- For every ACK received that acknowledges new data, the cwnd is incremented by the number of bytes in the sender's MSS
  - I.e., by 1 full-sized segment
- This results in an **exponential growth** in the number of segments that can be sent per RTT

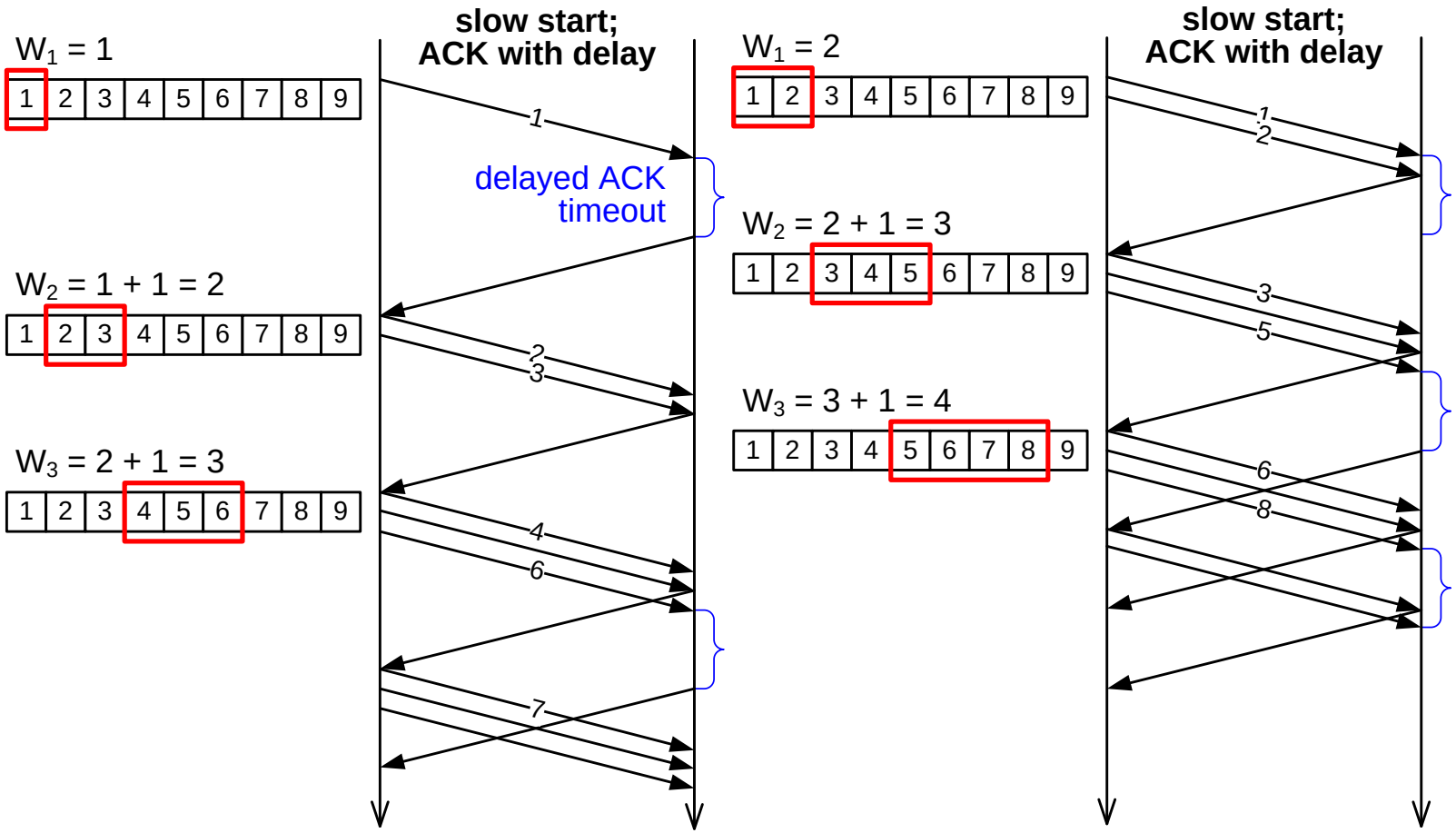
# Slow Start (cont'd)

- When IW $= 1$ segment, a receiver employing delayed ACKs is forced to wait for a timeout before generating an ACK



# Slow Start (cont'd)

- When IW $\geq$ 2 segments, the receiver will generate an ACK after the second data segment arrives, which eliminates the wait on the timeout



# Congestion Avoidance

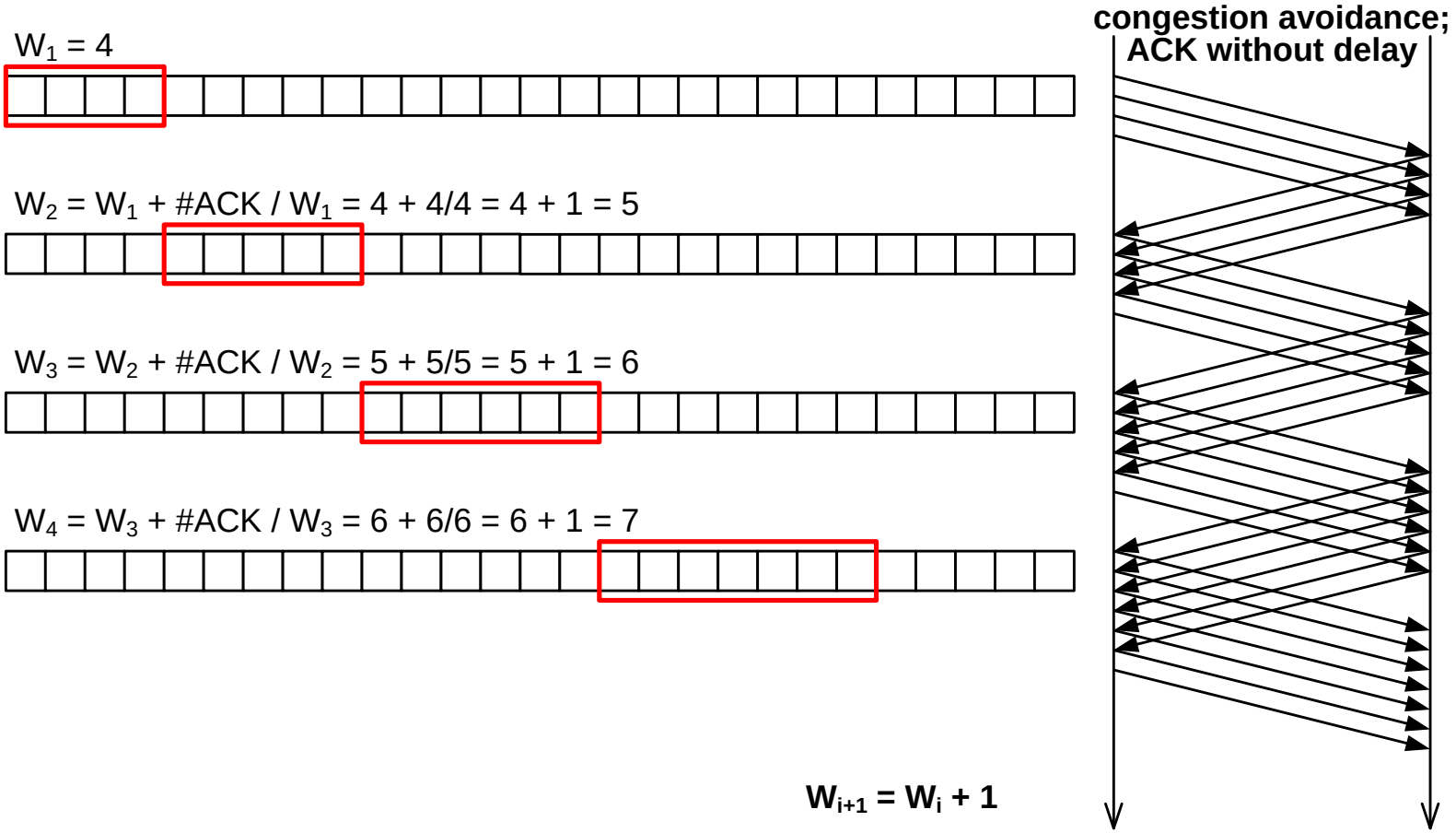
- During slow start, the cwnd increases exponentially over time until a packet loss occurs or the cwnd reaches the ssthresh
- In case of a packet loss, TCP interprets it as the evidence that the network is experiencing congestion and reduces the size of the cwnd
  - In turn, this slows down the sending rate and helps to alleviate the congestion problem
- In case when the cwnd exceeds the value of the ssthresh, TCP enters **congestion avoidance**
- As a rule, the initial value of the ssthresh is set to 65,535 bytes

# Congestion Avoidance (cont'd)

- During congestion avoidance, the cwnd is incremented by **MSS∗MSS/cwnd** bytes for every non-duplicate ACK
- This leads to a **linear growth** of the cwnd, compared to slow start's exponential growth
  - When the receiver acknowledges every received segment, the cwnd increases by 1 full-sized segment for each successfully transmitted window
  - When the receiver employs delayed ACKs, the cwnd increases by 1 full-sized segment for every other successfully transmitted window
- Congestion avoidance continues until congestion is detected

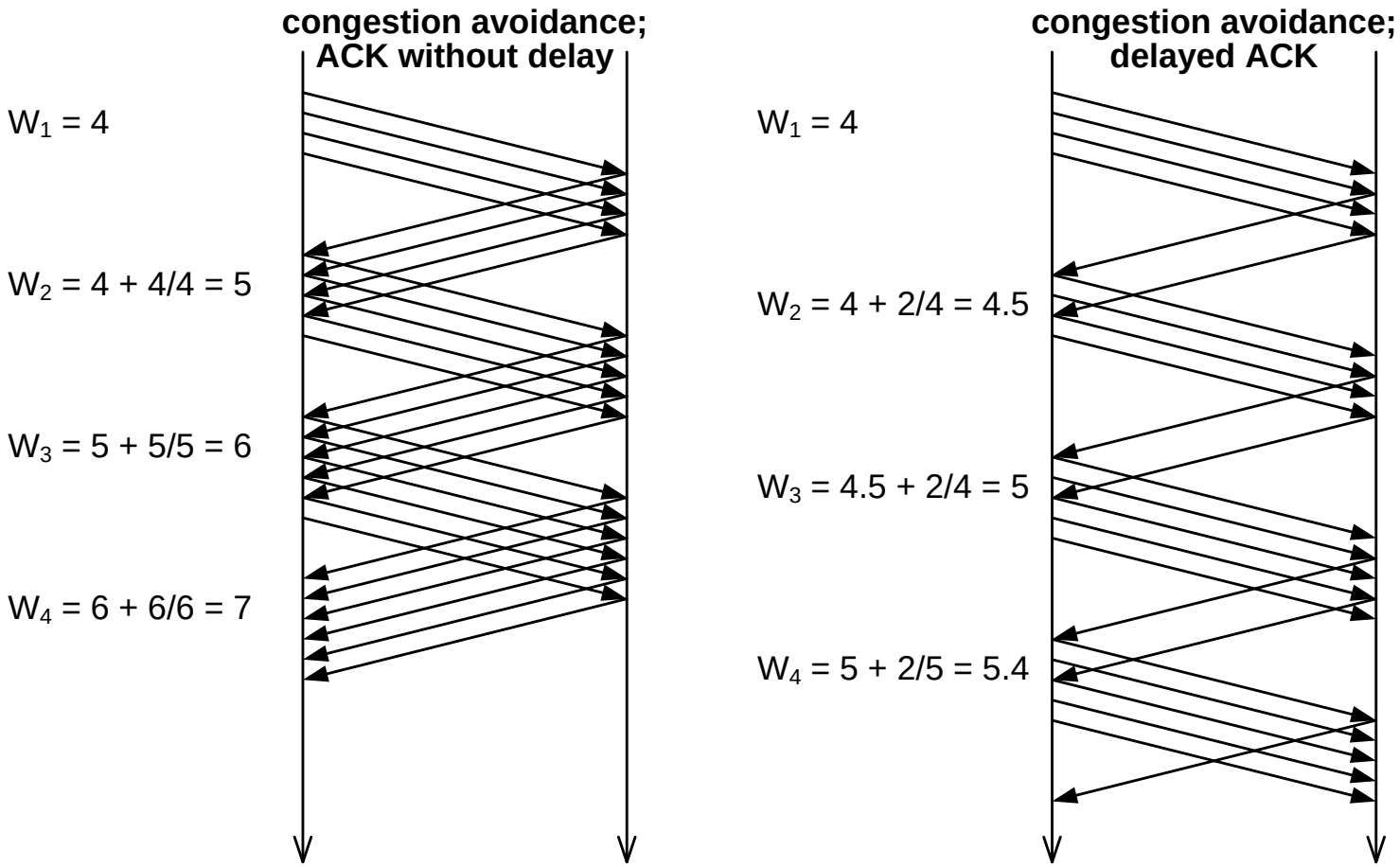
# Congestion Avoidance (cont'd)

- The congestion avoidance phase continues as long as new ACKs arrive before their corresponding timeouts



# Congestion Avoidance (cont'd)

- Since TCP increases the cwnd based on the number of arriving ACKs, reducing the number of ACKs slows the cwnd growth rate

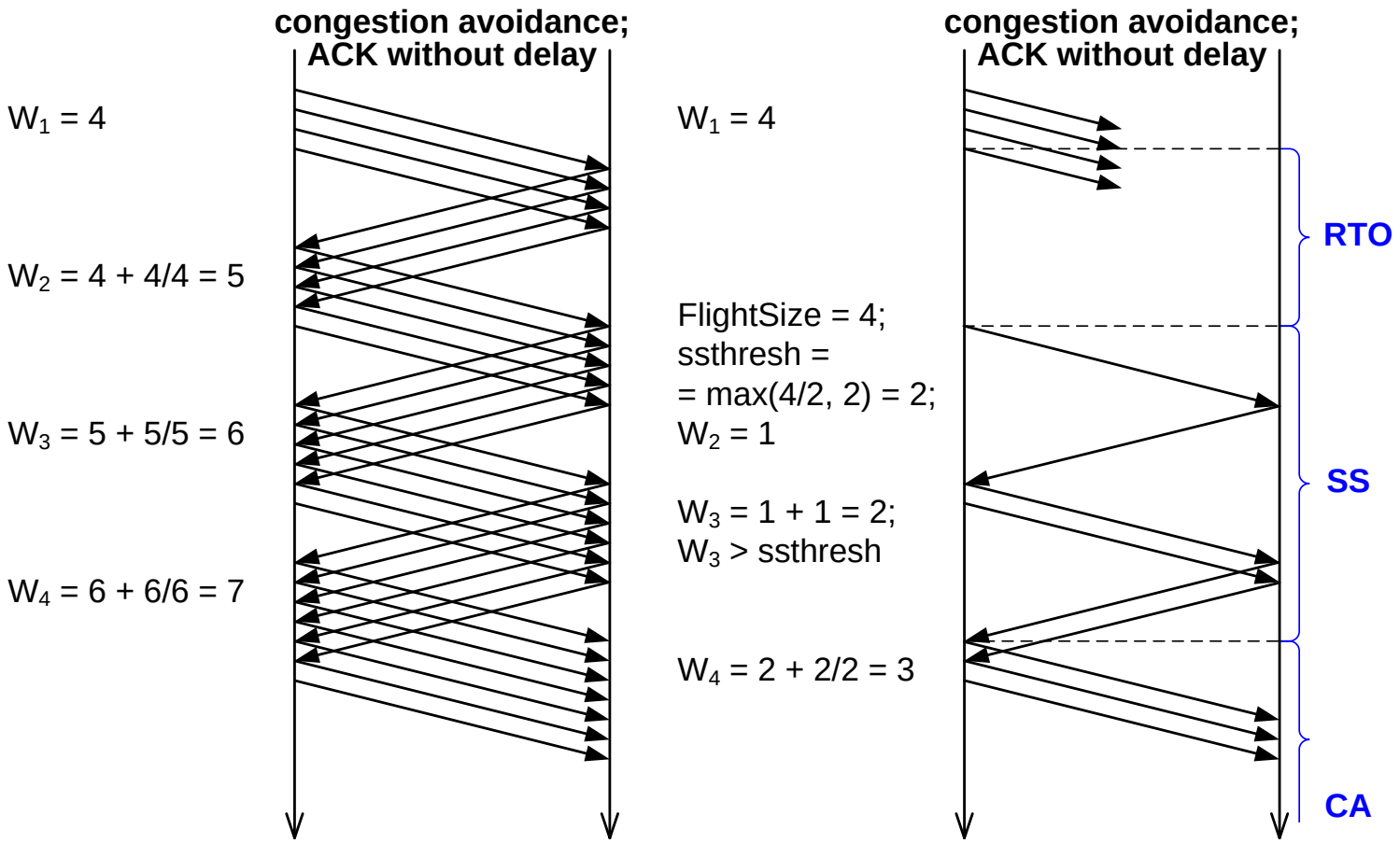


# Congestion Avoidance (cont'd)

- **TCP detects a segment loss in 2 different ways**:
  - By a timeout event
  - By the arrival of duplicate ACKs
- The sender starts the retransmission timer when it sends a window of segments and assumes that the data were lost if no ACK has been received within the specified period
- Then the sender sets the ssthresh to be 1/2 the amount of data in flight (aka **the flight size**) or 2 full-sized segments, whichever is larger
  - ssthresh = max (FlightSize/2, 2∗MSS)
- The sender also sets the cwnd to the size of 1 full-sized segment, retransmits the lost segment, and enters slow start
- When the ssthresh is exceeded, the sender enters the congestion avoidance phase again

# Congestion Avoidance (cont'd)

- After retransmitting the lost segment, the sender uses the slow start algorithm to increase the cwnd from 1 full-sized segment to the new value of the ssthresh



# Fast Retransmit and Fast Recovery

- The other way TCP can detect a segment loss is by the arrival of duplicate ACKs, which is known as **fast retransmit**
- When a duplicate ACK is received, the sender does not know if this is because a data segment was lost or because it was delayed and received out-of-order at the receiver
- Then the sender waits for a small number of duplicate ACKs to be received
- In order to provide timely detection of lost data, the fast retransmit algorithm is triggered when the sender receives 3 duplicate ACKs
  - I.e., 4 identical ACKs in a row
- Thus, after receiving the third duplicate ACK, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire

# Fast Retransmit and Fast Recovery (cont'd)

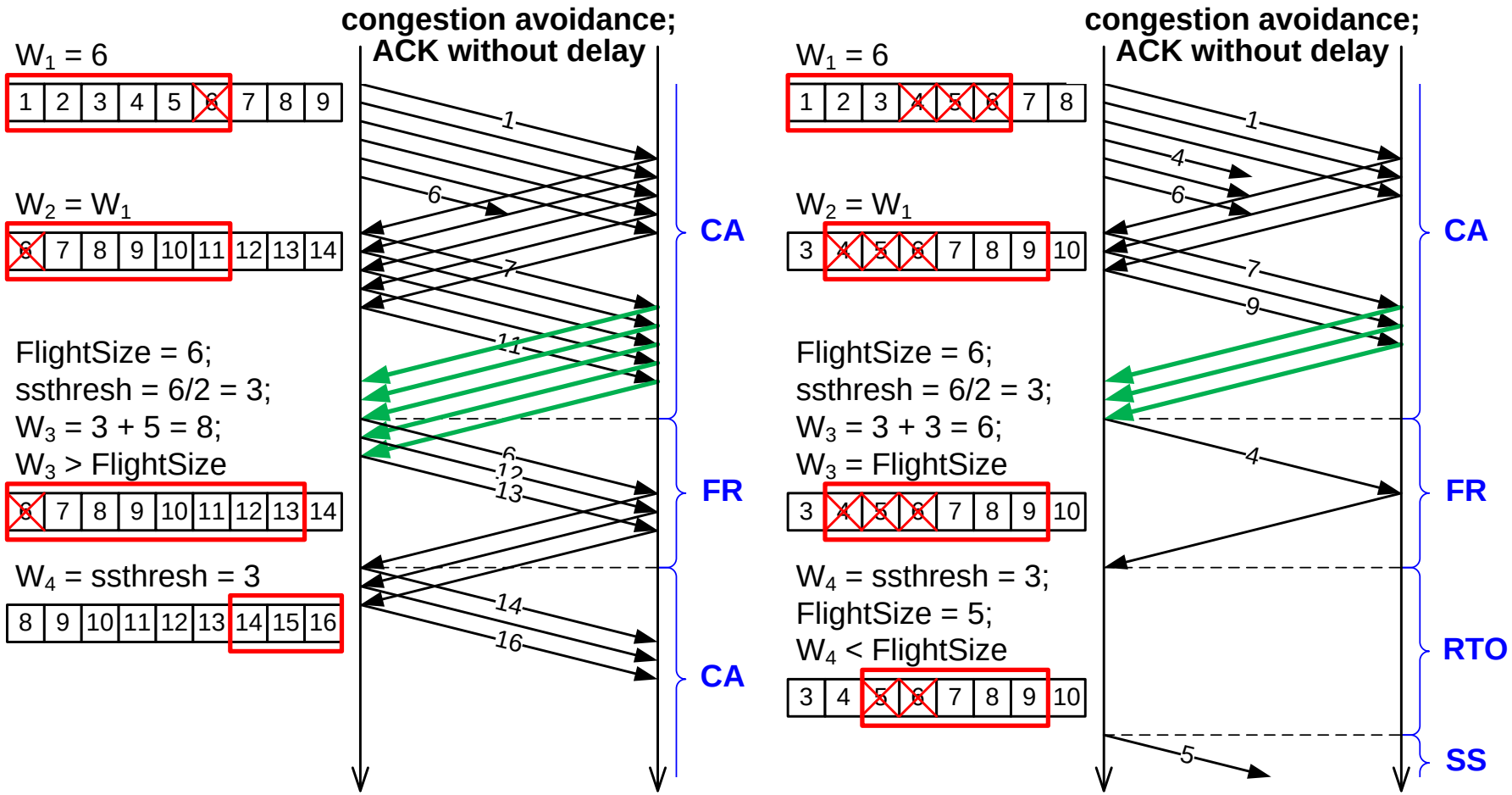
- Fast retransmit allows to avoid timeout and the subsequent slow start phase
- The reason for not performing slow start in this case is that the receipt of the duplicate ACKs tells TCP more than just a segment has been lost:
  - Since the receiver can only generate a duplicate ACK when another segment is received, this implies that subsequent segments have left the network and are in the receiver's buffer
  - Thus, there is still data flowing between the hosts, and TCP does not need to reduce the data rate abruptly by going into slow start
- The fast retransmit algorithm first appeared in **TCP Tahoe** and was followed by slow start
- The fast recovery algorithm appeared in **TCP Reno**

# Fast Retransmit and Fast Recovery (cont'd)

- To speed up the recovery of the sending rate after congestion in the network has been detected and eliminated, a new algorithm, called **fast recovery**, has been specified in RFC 2001
- The fast retransmit and fast recovery algorithms are usually implemented together:
  1. When the third duplicate ACK is received, the ssthresh is set as **max (FlightSize/2, 2∗MSS)**
  2. The lost segment is retransmitted and the cwnd is set to **ssthresh + 3∗MSS**
  3. For each additional duplicate ACK received, the cwnd is incremented by 1 full-sized segment
  4. A new segment is transmitted, if allowed by the updated value of the cwnd and the value of the rwnd advertised in the duplicate ACKs
  5. When the next ACK arrives that acknowledges new data, the cwnd is set to the ssthresh and the sender enters congestion avoidance

# Fast Retransmit and Fast Recovery (cont'd)

- TCP Reno has serious performance problems when multiple segments in the same window are lost



# Fast Retransmit and Fast Recovery (cont'd)

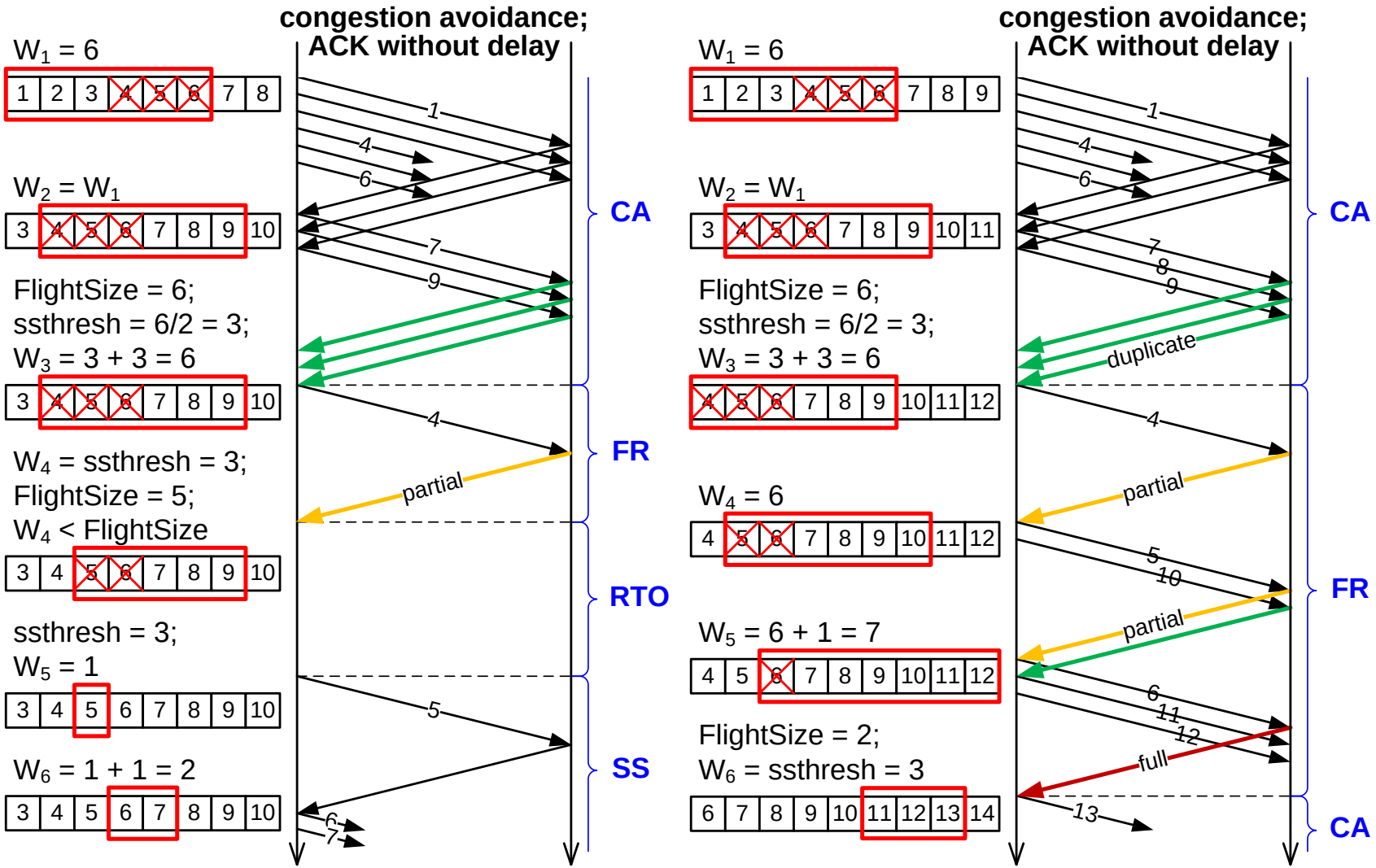
- For a long time, the reference TCP implementation has been TCP Reno first deployed in the 4.3BSD-Reno
- Later it was found that TCP Reno has serious performance problems when multiple segments in the same window are lost
- Since only 1 of the lost segments can be recovered by each invocation of the fast retransmit algorithm, the rest are often recovered using slow start after a usually lengthy retransmission timeout

# Fast Retransmit and Fast Recovery (cont'd)

- **TCP NewReno** is a modification of the basic TCP Reno implementation and incorporates slow start, congestion avoidance, and fast retransmit with **modified fast recovery**
- In TCP Reno (RFC 2581), the reception of a partial ACK takes the sender out of fast recovery
  - **Partial ACK** – an ACK that acknowledges some but not all of the segments sent before fast retransmit
  - **Full ACK** – an ACK that acknowledges all of the segments sent before fast retransmit
- In TCP NewReno (RFC 3782), the sender stays in fast recovery until either all of the segments outstanding by the time fast recovery was entered are acknowledged or the retransmission timer expires

# Fast Retransmit and Fast Recovery (cont'd)

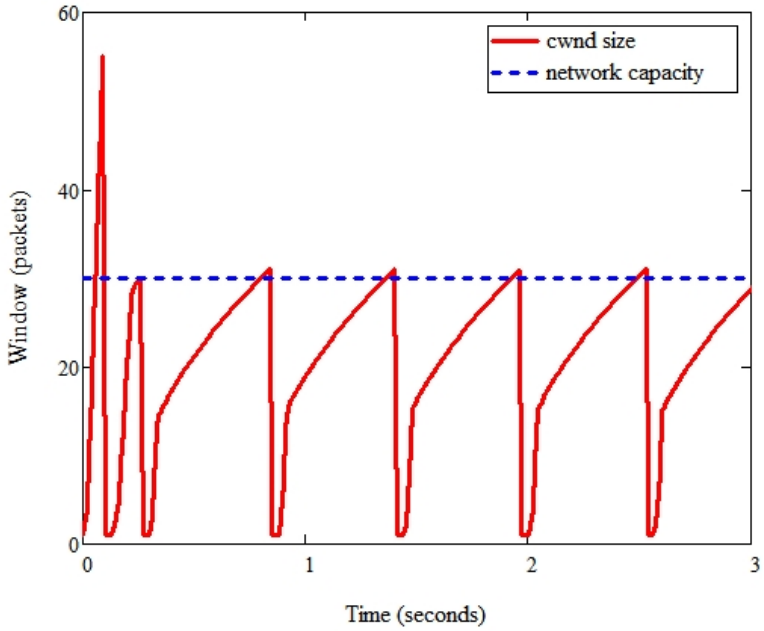
- TCP Reno vs. TCP NewReno (see RFC 3782 for details)



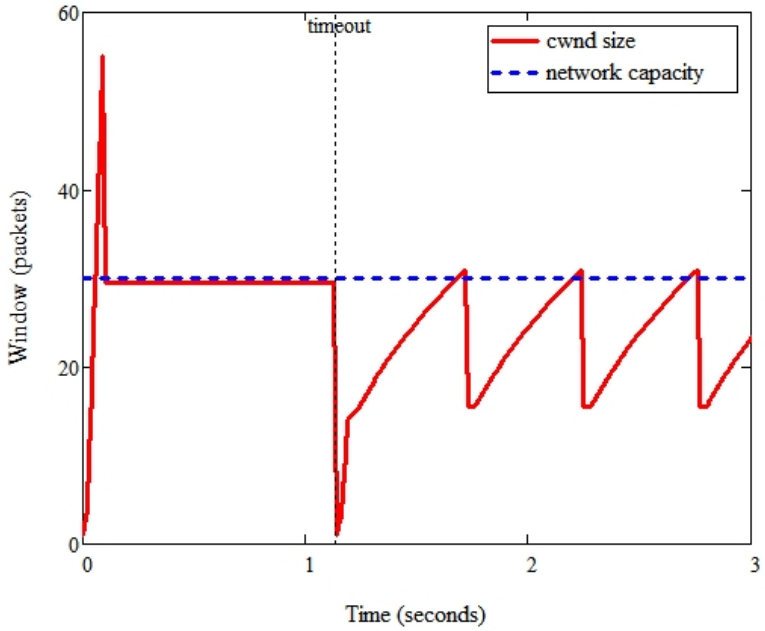
# Fast Retransmit and Fast Recovery (cont'd)

- Every time a segment loss is encountered (regardless of the way it is detected), TCP Tahoe drops the cwnd to 1 full-sized segment and enters the slow start phase

- TCP Tahoe



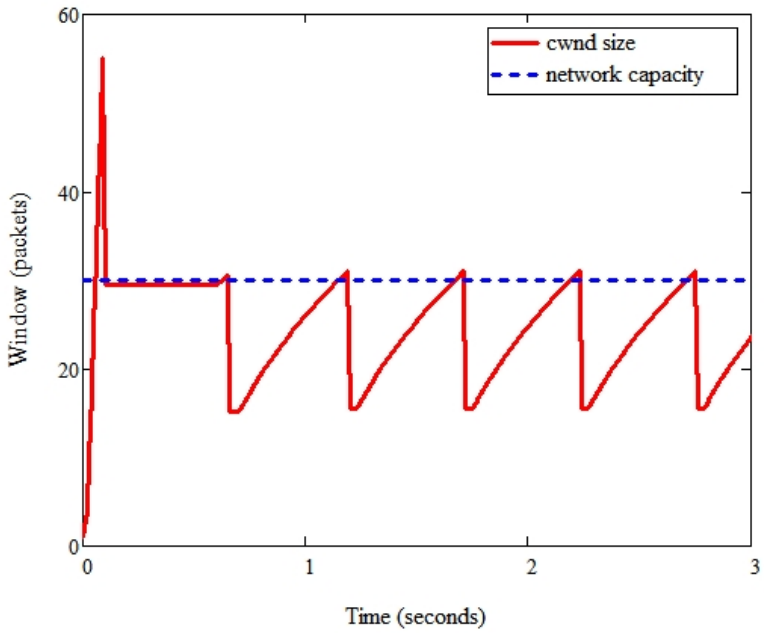

- TCP Reno



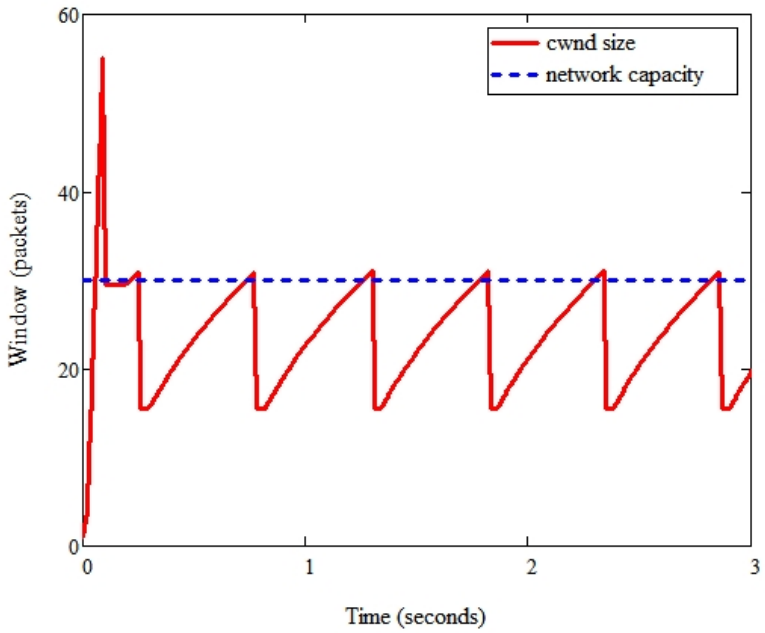
# Fast Retransmit and Fast Recovery (cont'd)

- TCP Reno, TCP NewReno, and TCP SACK provide similar performance in the presence of uncorrelated packet losses (when losses are predominantly single-packet losses)

- TCP NewReno




- TCP SACK



# Fast Retransmit and Fast Recovery (cont'd)

- Nowadays, TCP NewReno and TCP SACK are the most widely used TCP implementations

| **Algorithm** | **TCP implementation** | | | |
|---|---|---|---|---|
| | **Tahoe** | **Reno** | **NewReno** | **SACK** |
| Acknowledgements | Cumulative only | Cumulative only | Cumulative only | Cumulative and selective |
| Karn's algorithm | Yes | Yes | Yes | Yes |
| Initial window | Yes | Yes | Yes | Yes |
| Slow start | Yes | Yes | Yes | Yes |
| Congestion avoidance | Yes | Yes | Yes | Yes |
| Fast retransmit | Yes | Yes | Yes | Enhanced |
| Fast recovery | | Yes | Modified | Enhanced |

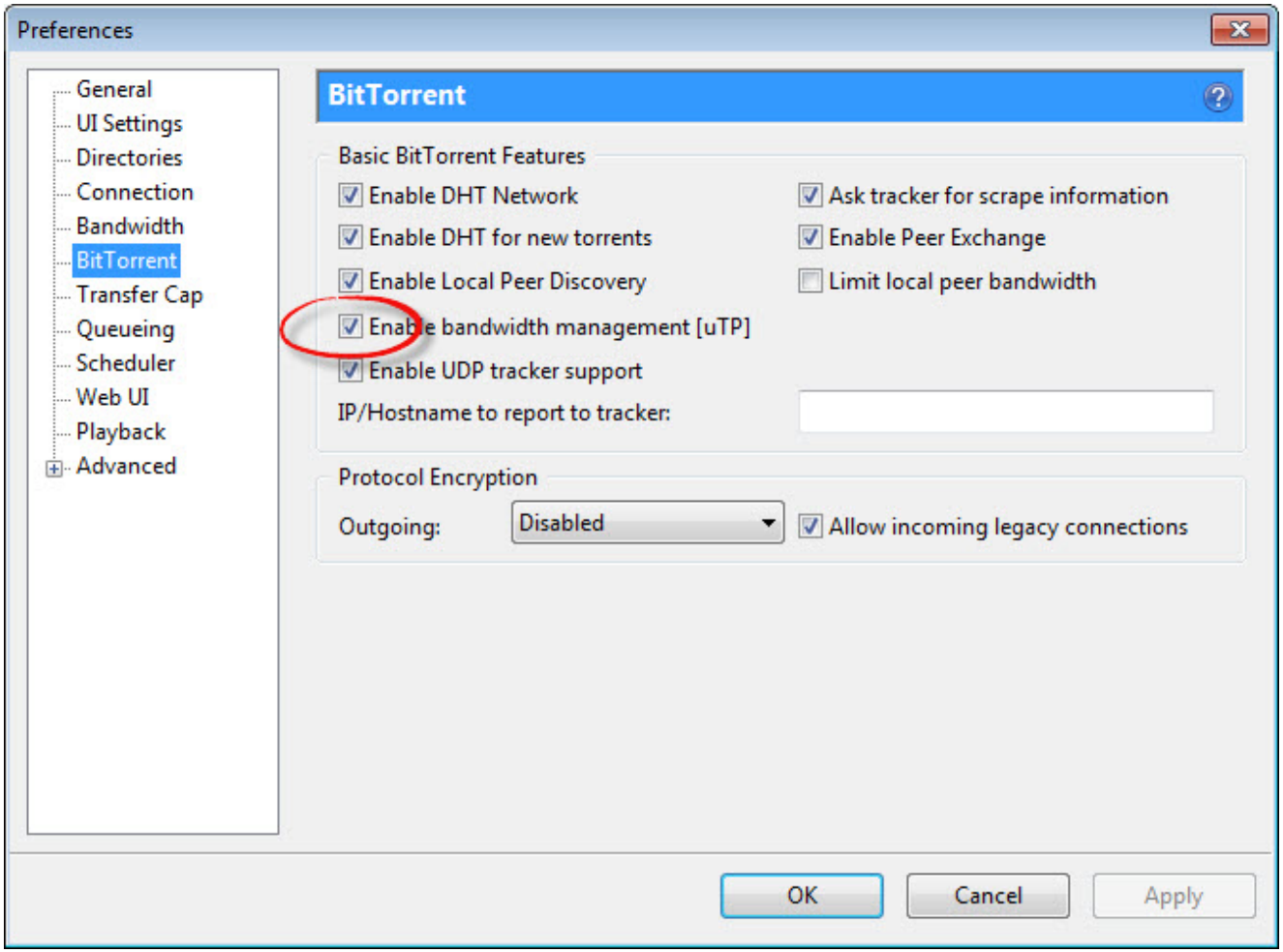
# Outline

# LEDBAT

- **Low Extra Delay Background Transport (LEDBAT)** – a way to transfer data on the Internet quickly without clogging the network
- LEDBAT works as follows (RFC 6817):
  1. Performs one-way delay measurements to estimate queueing delay
  2. When the estimated queueing delay is less than a predetermined target, infers that the network is not yet congested and increases its sending rate to utilize any spare capacity in the network
  3. When the estimated queueing delay becomes greater than the predetermined target, decreases its sending rate as a response to potential congestion in the network
- LEDBAT can be used with
  - TCP
  - Datagram Congestion Control Protocol (DCCP)
  - Appropriate extensions built on top of UDP

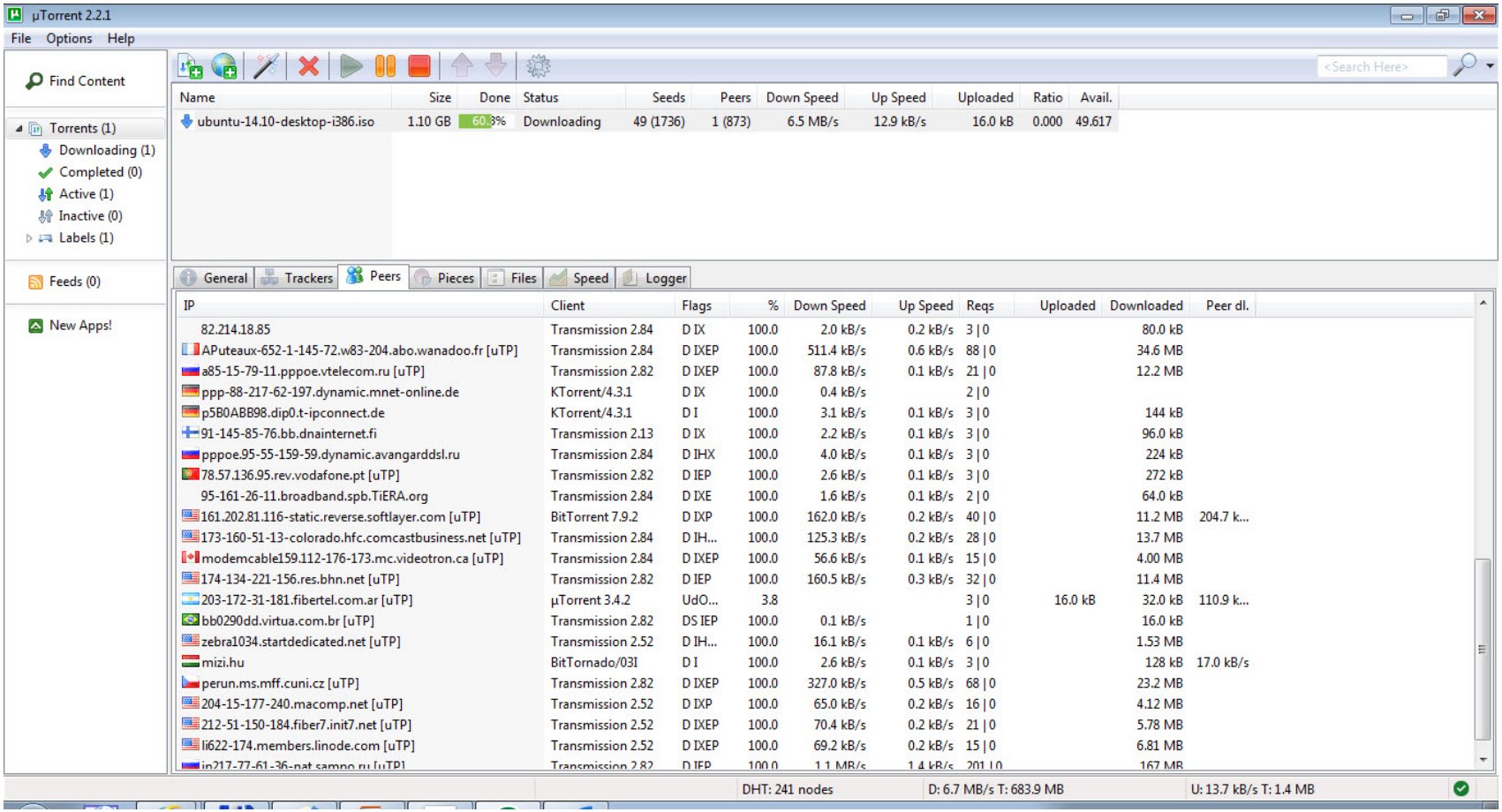
# LEDBAT (cont'd)

- Congestion control in BitTorrent:
  - **uTorrent Transport Protocol (uTP)**
  - TCP



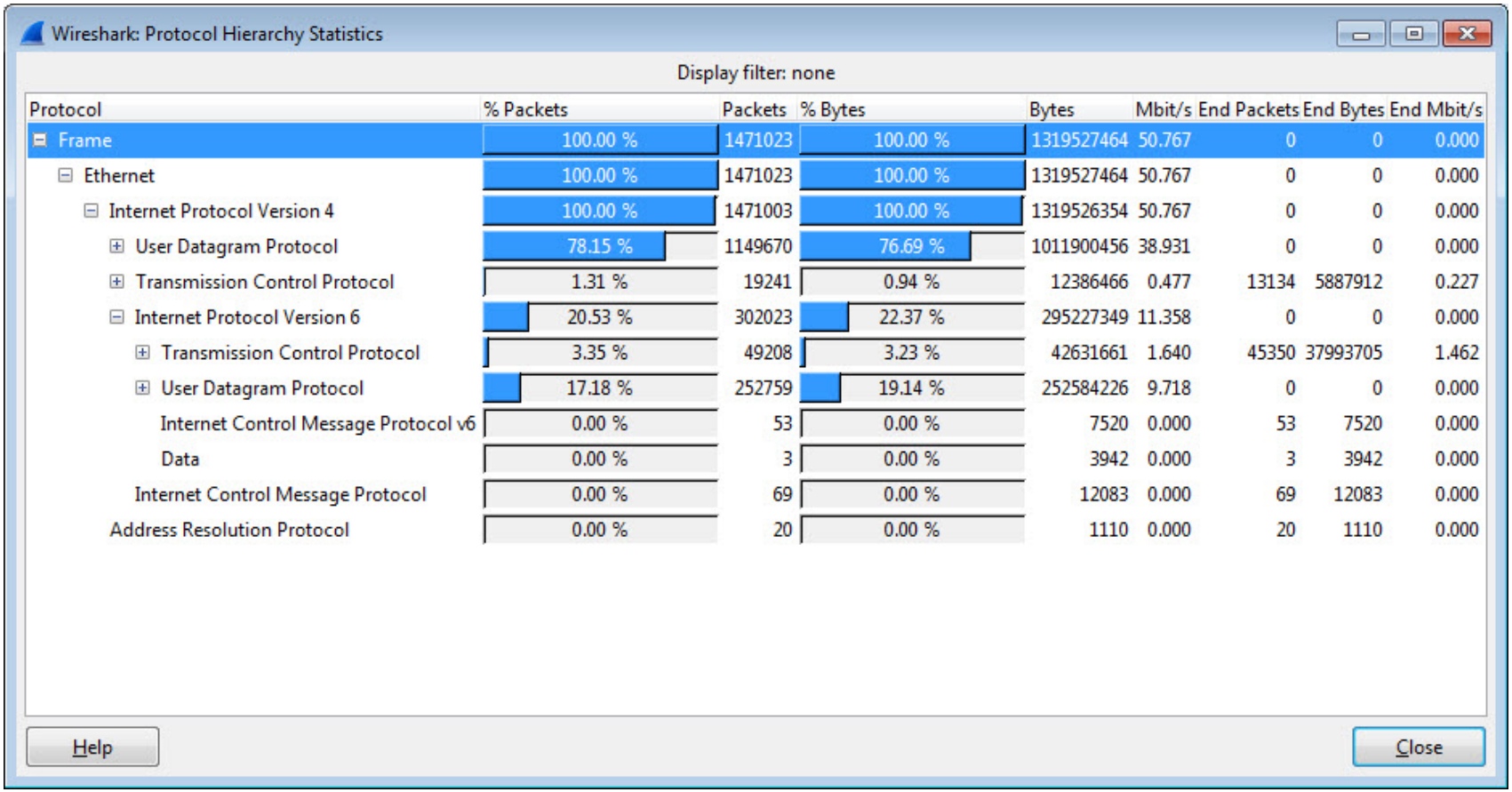
# LEDBAT (cont'd)

- The motivation for uTP is for BitTorrent clients to not disrupt internet connections, while still utilizing the unused bandwidth fully



# LEDBAT (cont'd)

- Today, **uTP controls a large fraction of BitTorrent's bytes and packets traversed over the Internet**




| Protocol | % Packets | Packets | % Bytes | Bytes | Mbit/s | End Packets | End Bytes | End Mbit/s |
|---|---|---|---|---|---|---|---|---|
| Frame | 100.00 % | 1471023 | 100.00 % | 1319527464 | 50.767 | 0 | 0 | 0.000 |
| Ethernet | 100.00 % | 1471023 | 100.00 % | 1319527464 | 50.767 | 0 | 0 | 0.000 |
| Internet Protocol Version 4 | 100.00 % | 1471003 | 100.00 % | 1319526354 | 50.767 | 0 | 0 | 0.000 |
| User Datagram Protocol | 78.15 % | 1149670 | 76.69 % | 1011900456 | 38.931 | 0 | 0 | 0.000 |
| Transmission Control Protocol | 1.31 % | 19241 | 0.94 % | 12386466 | 0.477 | 13134 | 5887912 | 0.227 |
| Internet Protocol Version 6 | 20.53 % | 302023 | 22.37 % | 295227349 | 11.358 | 0 | 0 | 0.000 |
| Transmission Control Protocol | 3.35 % | 49208 | 3.23 % | 42631661 | 1.640 | 45350 | 37993705 | 1.462 |
| User Datagram Protocol | 17.18 % | 252759 | 19.14 % | 252584226 | 9.718 | 0 | 0 | 0.000 |
| Internet Control Message Protocol v6 | 0.00 % | 53 | 0.00 % | 7520 | 0.000 | 53 | 7520 | 0.000 |
| Data | 0.00 % | 3 | 0.00 % | 3942 | 0.000 | 3 | 3942 | 0.000 |
| Internet Control Message Protocol | 0.00 % | 69 | 0.00 % | 12083 | 0.000 | 69 | 12083 | 0.000 |
| Address Resolution Protocol | 0.00 % | 20 | 0.00 % | 1110 | 0.000 | 20 | 1110 | 0.000 |