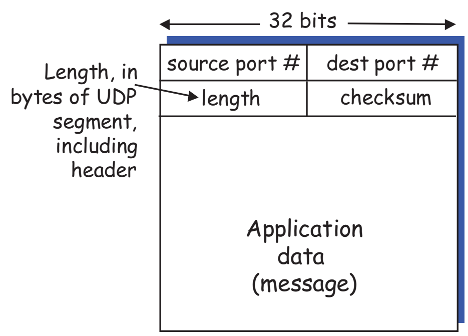
Internet Transport Protocols: UDP and TCP

3. UDP: User Datagram Protocol

Slide 1

- See Section 3.3.
- UDP from previous lectures: connectionless, no flow control, no congestion control, UDP sockets and demultiplexing, used in DNS and multimedia applications
- Segment Format

Slide 2



[JFK/KWR 2/E]

- checksum: sender computes the sum of all 16-bit words (overflows discarded), then stores the 1's complement of the sum.

  Receiver checks the sum of all 16-bit words (including the stored checksum) is $1111\ldots1$.

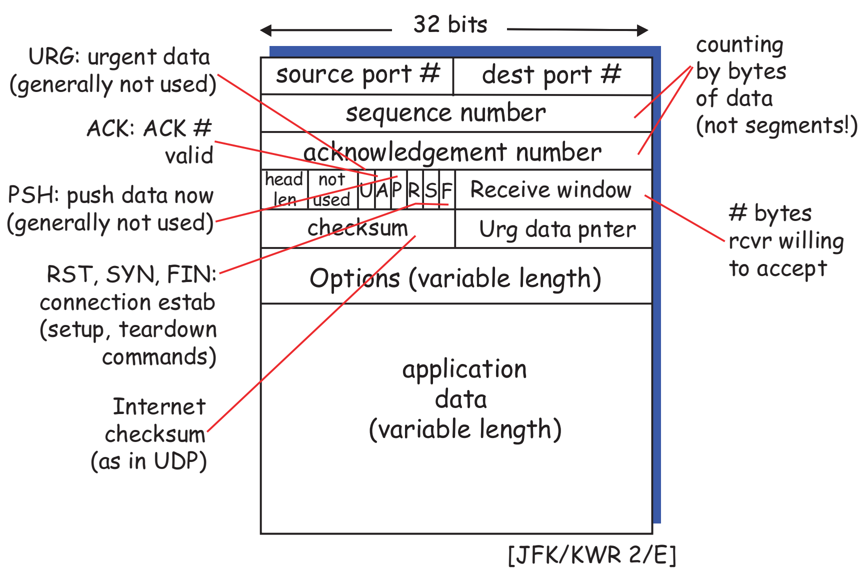
Slide 3

4. TCP: Transport Control Protocol

- See also Section 3.5.
- TCP from previous lectures: provides reliable connection-oriented byte stream bundled with flow control and congestion control
- Evolved over more than 15 years: currently there are many versions of the TCP protocol: Tahoe ('88), Reno ('90), etc.

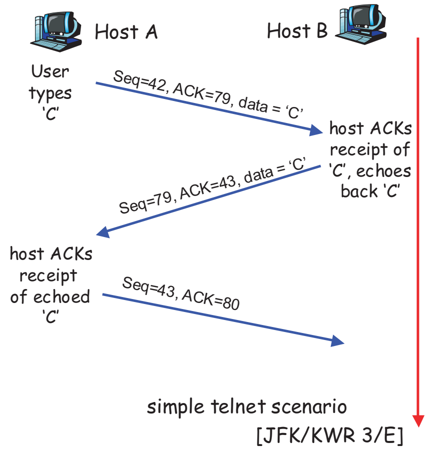
Slide 4

TCP Segment Format and Some Basic Operations



- Seq #s and ACK #s: note the count by bytes, not segments.

Slide 5



- Seq #: number of first byte in segment's data
- ACK #: number of next byte expected from other side (ACKs are cumulative)
- Handling out-of-order segments: up to the implementor.

- Receive window: (used for flow control) indicates the number of bytes the

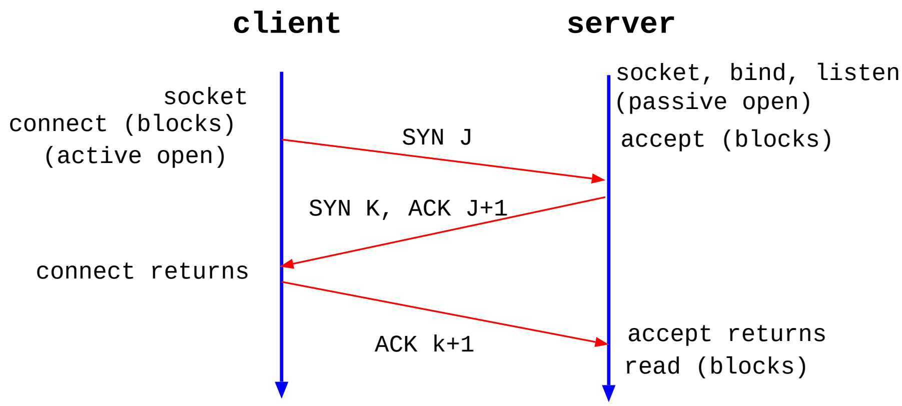
Slide 6

receiver is willing to accept

- Header length (4 bits): length of TCP header in 32-bit words (so, TCP header is 5 words if the options field is empty
- The flag field (6 bits) :
  - ACK bit: $ACK = 1$ indicates the 32-bit number in the ACK field is valid
  - SYN bit: used during connection setup

Recall, the 3-way handshake connection setup:

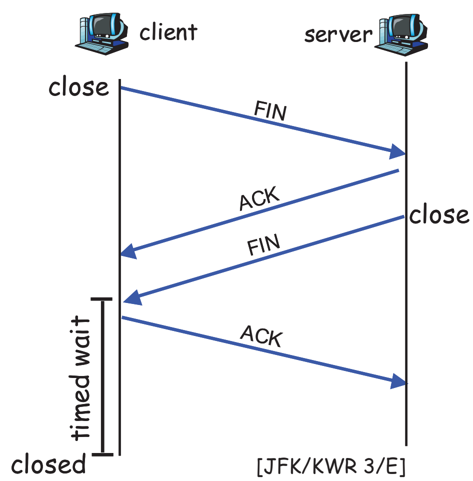
Slide 7



- FIN bit: (finish) used for connection closure
- RST bit: (reset) resets a connection that has become confused, or rejects an attempt to open a connection

So, a simple connection release looks like:

Slide 8



- URG bit: indicates that the sender application has inserted a one byte urgent data in the sender's TCP buffer.
  - ∗ Note that multiple segments can have the URG flag on without actually having the transmitted urgent byte
  - ∗ The 16-bit urgent data pointer indicates the location of the urgent byte

Slide 9

relative to the current sequence number

- PSH bit: the receiver is requested to pass the data to the application layer immediately

- The options field: (optional and has a variable-length) provides a way to add facilities not covered by the above fields: e.g.,
  - sender/receiver negotiate the maximum segment size (MSS) (otherwise, MSS defaults to 536-byte payload), or
  - sender/receiver negotiate a window scaling factor (for use in high speed networks)

TCP Reliable Data Transfer

- TCP uses techniques from both Go-Back-N and Selective Repeat, e.g.:
  - TCP uses pipelining: it allows multiple unACKed segments
  - TCP uses one retransmission timer and cumulative ACKs (as in GBN)

Slide 10

- TCP retransmits only one TCP segment at a time (like SR)

- Additionally, TCP uses some new ideas: e.g.,
  - fast retransmission on three duplicate ACKs
  - doubling the timeout interval
  - delayed ACKs
  - variable window size: for flow control and congestion control

- We'll start by considering a simplified TCP sender (without the new ideas) and then discuss the new modifications.

- The simplified sender uses:
  - SendBase: seq # of the oldest unACKed byte (so, $SendBase - 1$ is the seq # of the last cumulatively ACKed byte)
  - NextSeqNum: seq # of next byte to send

Slide 11

sender_loop {
  event: data received from application
    create TCP segment with NextSeqNum
    if (timer not running) start timer
    pass segment to IP
    NextSeqNum= NextSeqNum + length (data)

  event: timeout
    retransmit not-yet-ACKed segment with smallest seq #
    start timer

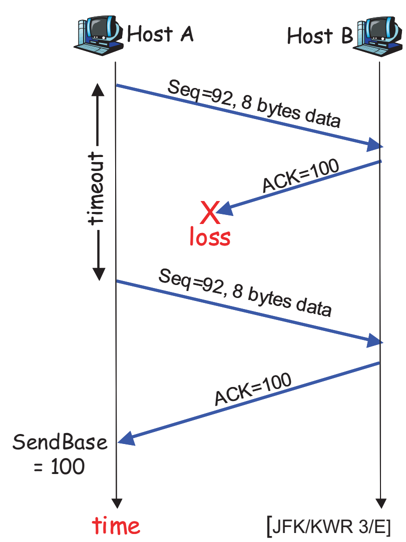
  event: ACK[y] received
    if (y > SendBase) {
    SendBase= y
    if (there are currently not-yet-ACKed segments)
      start timer
    }
}

Note: The retransmission timer is restarted on 3 different events. Shortly,

Slide 12

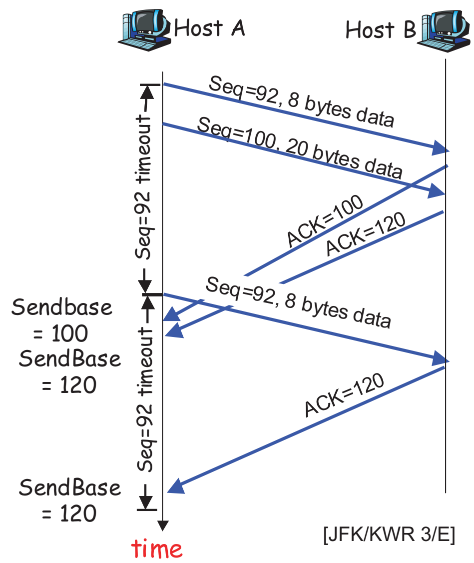
we'll discuss choosing a different timeout interval when the second event (timeout) occurs.

- A Lost ACK scenario



- A Premature Timeout Scenario

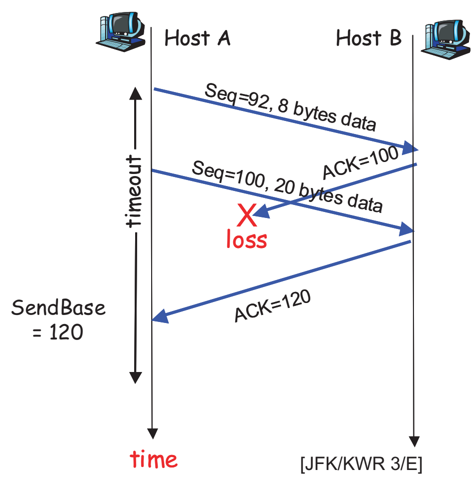
Slide 13



Note: the second segment will not be retransmitted.

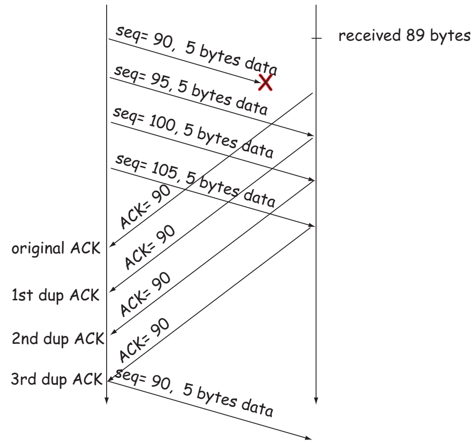
- A Cumulative ACK Scenario

Slide 14



- Fast Retransmit:
  - time-out period is often relatively long
  - a good idea is then to detect lost segments via duplicate ACKs (here we wait for 3 dup. ACKs) and then retransmit an unACK'ed segment

Slide 15



- The sender's algorithm can now be revised as:

Slide 16

```
event: ACK [y] received
        if (y > SendBase) {
           SendBase = y
           if (there are currently not-yet-acknowledged segments)
              start timer
          }
        else {
            increment count of dup ACKs received for y
            if (count of dup ACKs received for y = 3) {
               resend segment with sequence number y
            }
```

a duplicate ACK for already ACKed segment

fast retransmit

[JFK/KWR 2/E]

- Doubling the Timeout Interval
  - TCP uses a certain method to compute a reasonable timeout interval (i.e.,

Slide 17

not too long, and not too short), denoted TimeoutInterval , each time an ACK is received for a segment that has not been retransmitted.

- Shortly, we'll see the details of computing TimeoutInterval
- Doubling the timeout interval refers to:
  - ∗ If a timeout event occurs then TCP assumes that the network is congested; TCP then retransmits the timed out segment, and restarts the timer with 2 TimeoutInterval , if a consecutive timeout event follows then the timer is restarted with 4 TimeoutInterval , and so on.
  - ∗ On the other hand, if the timer is started after the two other events (i.e., ACK received, or application data is received) then the derived TimeoutInterval is used.

- Delayed ACKs : ACK Generation Recommendation [RFC 1122, RFC 2581]

Slide 18

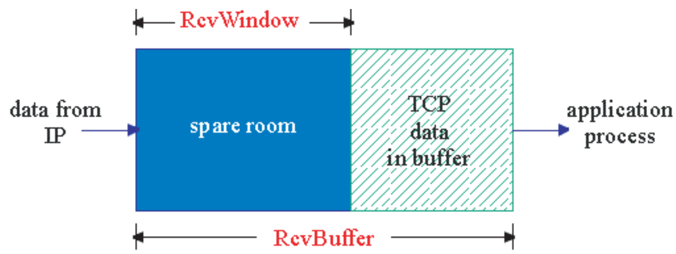
Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

[JFK/KWR 3/E]

Slide 19

TCP Flow Control

- Objective: sender should not overflow receiver's buffer by transmitting too much, too fast. How to?
  - for simplicity, assume TCP receiver discards out-of-order segments.
  - So, the receiver's buffer looks like:



[JFK/KWR 3/E]

the spare room is then specified by

$$RcvWindow = RcvBuffer - [LastByteRcvd - LastByteRead]$$

  - In each segment, the receiver advertises the RcvWindow value.

Slide 20

  - Sender limits its window of unACKed data to the received RcvWindow value
  - That is, sender makes sure that

$$\boxed{LastByteSent - LastByteAcked \leq RcvWindow}$$

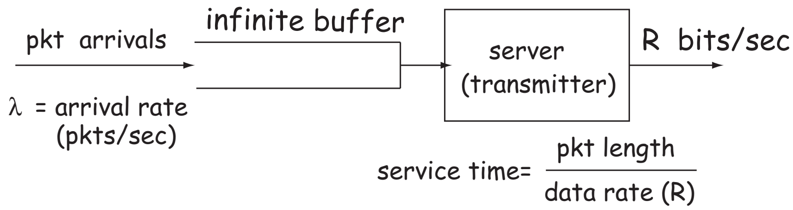
TCP Congestion Control

Causes/Costs of Congestion

- Is congestion bad?
  Sure, routers will drop pkts, senders will timeout, and the network will be flooded with retransmissions.
- However, the causes/costs of congestion are probably not very well appreciated by the above brief remarks, so let's look at two simple scenarios.
- Scenario: one router with an infinite buffer

Slide 21

- "infinite buffer" is unrealistic, however, the analysis gives insight of cases where the buffer is made large enough so no packet is dropped:
- A well-known queueing result states that:



$*$ arrival process: Poisson with rate λ pkt/sec

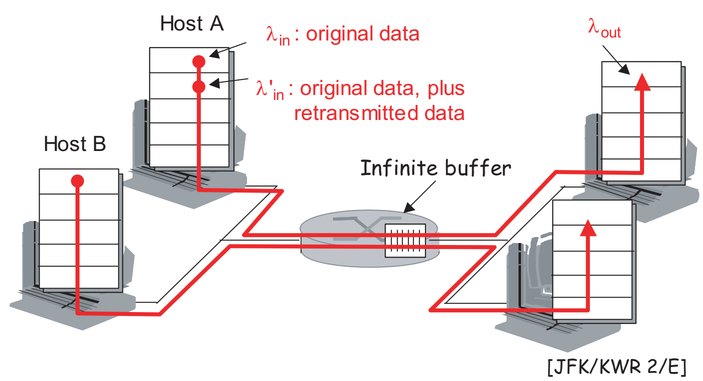
$*$ service time: exponentially distributed with average $\frac{1}{\mu}$ sec. per pkt (that is, μ pkts/sec can be viewed as the departure rate) then

$*$ a steady state solution exists only if $\lambda < \mu$

$*$ the average pkt delay is given by $\frac{1}{\mu - \lambda}$

- So, if two TCP connections share the same (infinite buffer) router, and each transmits equally fast

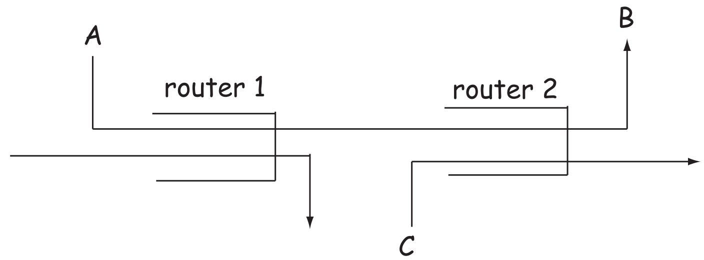
Slide 22



then each connection will enjoy a fair share of $\lambda'_{in} = R/2$ bps, nevertheless, because of delays and retransmissions the throughput is much less than $R/2$.

- Scenario: a 2-hop connection (routers have finite buffers)

Slide 23



- Assuming all 3 connections transmit equally fast, then AB-packets will be seen less than half the time at the output of router 2.
- That is, the AB-connection will not even enjoy the expected fair share of $R/2$ bps.

Approaches to Congestion Control

- Network Assisted Congestion Control:
  - routers provide traffic policing, shaping, and feedback to end systems

Slide 24

  - Examples: IBM SNA, and ATM (Asynchronous Transfer Mode) networks
- End-end Congestion Control:
  - no explicit feedback from the network
  - congestion is inferred by end systems from the observed loss and delay
  - Examples: TCP

TCP Congestion Control Mechanisms

- See also Section 3.7.
- Sender limits transmission rate. How?
  - Let's introduce:

    CongWin: max. number of sent but unACK'ed bytes
  - At any instant, sender makes sure that

$$LastByteSent - LastByteAcked \leq \min\left(CongWin, RcvWindow\right)$$

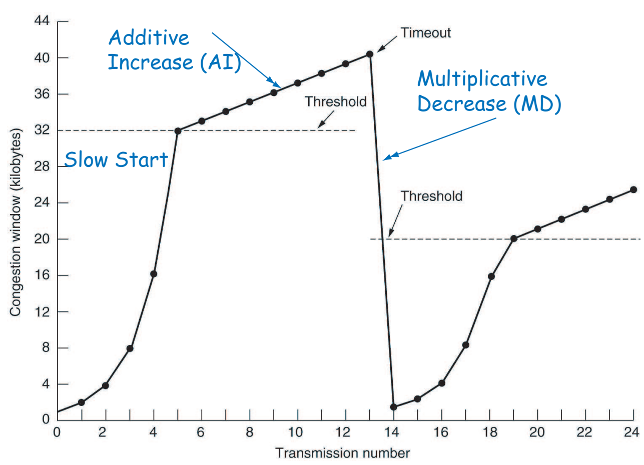
Slide 25

- So, roughly, during congestion (i.e., when $CongWin \leq RcvWindow$) the effective data rate $= CongWin/RTT$ bytes/sec.
- TCP reduces CongWin when it perceives a pkt loss event : i.e., either a timeout event, or a 3 duplicate ACKs.
  Note: TCP reacts to timeout events and 3-duplicate ACKs events differently. Why?

■ Basic Mechanisms for Adjusting CongWin:

- slow start
- additive increase
- multiplicative decrease (in reaction to loss events)

Slide 26



[Tan 4/E]

Note the variable $Threshold$. Initially, $Threshold$ is set to some 'big' value, say 64K Byte (it may change subsequently).

Slide 27

- Slow Start:
  - When connection begins, start with $CongWin = 1$ MSS.
  - Then increment $CongWin$ by 1 every ACK received
    - (this implies **doubling** $CongWin$ every RTT, assuming the source can keep the window full)

  until $CongWin$ reaches $Threshold$ or a loss event occurs.
- Additive Increase (AI):
  - Entered when $CongWin$ reaches $Threshold$.
  - $CongWin$ is increased by $\frac{MSS}{CongWin}$ every ACK received
    - (this implies increasing $CongWin$ linearly by 1 MSS every RTT, assuming the source can keep the window full)

  until a loss event occurs.
  - Viewed as a congestion avoidance phase.

Slide 28

- Multiplicative Decrease (MD) [TCP Reno]:
  - Loss Event is Time Out:
    set $Threshold = CongWin/2$; enter slow start
  - Loss Event is 3-Duplicate ACKs:
    set $Threshold = CongWin/2$; set $CongWin = Threshold$, enter additive increase.

Simplified TCP Throughput Analysis

- As can be seen, analyzing TCP's throughput is not easy.
- Nevertheless, obtaining a rough idea of the average throughput of a long-lived TCP connection gives useful insight.
- So, let's make the following (highly) simplifying assumptions:
  - First, let's ignore the slow start phases (they are typically very short, since the sender grows out of them exponentially fast).

Slide 29

- Second, assume that RTT is approximately constant over a long-lived connection.
- Third, assume that the CongWin size at which a loss event occurs (denoted W) is approximately constant over a long-lived connection.

- Then, the data rate of the connection
  - reaches a maximum of W/RTT bytes/sec then
  - loss occurs, and TCP lowers the rate to $0.5 \times W/RTT$, then
  - TCP increases the rate by MSS/RTT every RTT until it reaches the maximum W/RTT, and
  - the above process repeats.
- So, we may conclude that a rough estimate of the throughput is $0.75 \times W/RTT$.

Slide 30

RTT Estimation and Timeouts

- See Section 3.5.3.
- How should TCP set the timeout interval?
  - longer than RTT
  - not too short (else, we get premature timeouts and unnecessary retransmissions)
  - not too long (else, we get slow reaction to lost packets)
- The problem is to estimate the RTT. So, how can one estimate RTT?
- The concept of *exponential weighted moving average*:
  - suppose a random variable X takes on values:

    $$x_1, x_2, x_3, x_4, \ldots$$

    at discrete time instants $t_1, t_2, t_3, t_4, \ldots$
  - At time t_n, $n \geq 1$, we'd like to compute an estimate s_{n+1} of the actual value x_{n+1}, based on the already observed values $(x_1, x_2, \cdots, x_n)$

Slide 31

- We use:

$$s_{n+1} = (1-\alpha)s_n + \alpha x_n$$

where

* x_n is the nth actual (measured) value
* s_n is the nth predicted value (also called smoothed average) ($s_1 =$ some initial value)
* $\alpha \in [0,1]$ is a user-chosen control parameter

- Example:

	$x_1 = 6$	$x_2 = 4$	$x_3 = 6$	4	13	13	13	$\cdots$
$\alpha = 0.5$	$s_1 = 10$	$s_2 = 8$	$s_3 = 6$	$s_4 = 6$	5	9	11	12 $\cdots$

- How do the choice of s_0 and α influence the computations?
  Well, for one thing, the closer α to 0, the larger the weight placed on past history. So, if X changes slowly then choosing small α allows the estimate to ignore most of the noise in recent measurements.

- Now, let's go back to setting the time out interval.

Slide 32

- For each segment that is ACK'ed before a timeout, TCP records the RTT value in $SampleRTT$, and updates a smoothed average:

$$EstimatedRTT = (1-\alpha) * EstimatedRTT + \alpha \times SampleRTT$$

where $\alpha = 1/8$.

- Older versions of TCP used:

$$TimeoutInterval = 2 * EstimatedRTT$$

(multiplying by 2 accounts for variations in packet processing and queueing delays).

- Later, it has been noted that the RTT varies widely, and an additional smoothed average is needed:
  $DevRTT =$
  $(1-\beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$
  where $\beta = 0.25$ is typical; the resulting new key equation is:

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT.$$