Leopold–Franzens–University
Innsbruck

Institute of Computer Science

Distributed and Parallel Systems Group

Responding to Spurious Loss Events in TCP/IP

Master Thesis

Supervisor: Dr. Michael Welzl

Author: Raffler Thomas Bakk.techn

May 27, 2007

Eidesstattliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Contents

[1] LaTeX

List of Figures

Preface

Acknowledgements

This thesis would not have been possible without the help of the following people:

Supervisor: Dr. Michael Welzl

English enhancement: Kashif Munir

NS2 code for the Eifel Algorithm and the Delay spike generation: Andrei Gurtov

NS2 code for the F-RTO Algorithm: Pasi Sarolahti

NS2 code for ECN with a one-bit nonce: David Wetherall and Neil Spring

I am especially grateful to all my friends and colleagues who supported and helped me during this work.

Chapter 1

Introduction

The TCP Protocol is one of the core protocols of the Internet protocol suite. It provides reliable data transmission between two peers across an unreliable IP-based communication channel. The protocol guarantees reliable and in-order delivery of sender to receiver data.
The original specification of TCP is described in RFC793. (oSC81) TCP of this specification provides reliable full-duplex transmission of data streams between two peers. But in this specification there is no description of any congestion control features at all. TCP maintains its position as the dominant transport protocol in the Internet. TCP is stable, mature and probably the most thoroughly tested protocol of its kind. TCP performance is strongly influenced by its congestion control algorithms that limit the amount of transmitted traffic based on the estimated network capacity and utilization.

But there are some corner cases where TCP could still be improved. The problem of **spurious timeouts** or **spurious fast retransmits** i.e. timeouts that would not have occurred had the sender waited long enough, are examples of the above mentioned corner cases. Both effects can be named with the term **spurious loss events**.

Delays on Internet paths, especially on wireless links, can be highly variable. On the other hand, modern TCPs has implemented a fine grained retransmission timer with a lower minimum timeout value than 1 second. A retransmission timer is a prediction of the upper limit of the RTT (Round Trip Time [2.2.2]) .

A spurious timeout occurs when the RTT suddenly increases. The RTT can quickly return to normal, but can also stay high when available bandwidth of the bottleneck link has suddenly decreased. A spurious fast retransmit occurs when one or more segment(s) in the same window gets seemly lost, a specified number of DUPACKs arrived and the sender retransmits those segments again.

On a spurious loss event (spurious timeout and also on a spurious fast retransmit), TCP assumes that all outstanding (not acknowledged) segments are lost and retransmits them unnecessarily. The go-back-N [2.3.5] retransmission behavior triggered by spurious loss event has a root: **the retransmission ambiguity**. This happens when a sender is unable to distinguish an ACK for the original transmission of a segment from the ACK for its retransmission. In the case of spurious timeouts it looks like follow: Shortly after the timeout ACKs for the original transmission return to the TCP sender. On receipt of the first ACK after the timeout, the TCP sender must interpret this ACK as acknowledging the retransmission, and must assume that all other outstanding segments have also been lost. Thus, the sender enters the slow start phase [2.3.1] , and retransmits all outstanding seg-

ments in this fashion. The go-back-N [2.3.5] retransmission triggers the next problem: the receiver generates a DUPACK (duplicate ACK) for every segment received more than once. The receiver has to do this, because it must assume that its original ACKs had been lost. This can trigger a spurious fast retransmit [3.1.3] at the sender.

Another problem is that after a spurious loss event the congestion window and slow start threshold are reduced. This reduction is **unnecessary** as no data loss has yet been detected that would indicate congestion in the network.

Chapter 2

Background

2.1 TCP: Transmission Control Protocol

The TCP Protocol is one of the core protocols of the Internet. It provides reliable data transmission between two peers across an unreliable IP-based communication channel. The protocol guarantees reliable and in-order delivery of data from sender to receiver. The original specification of TCP is described in RFC793. (oSC81) TCP of this specification provides reliable full-duplex transmission of data streams between two peers. But in this specification there is **no description of any congestion control features** at all. TCP segments are carried in IP packets and begin with the header. The header is shown in Figure [2.1]

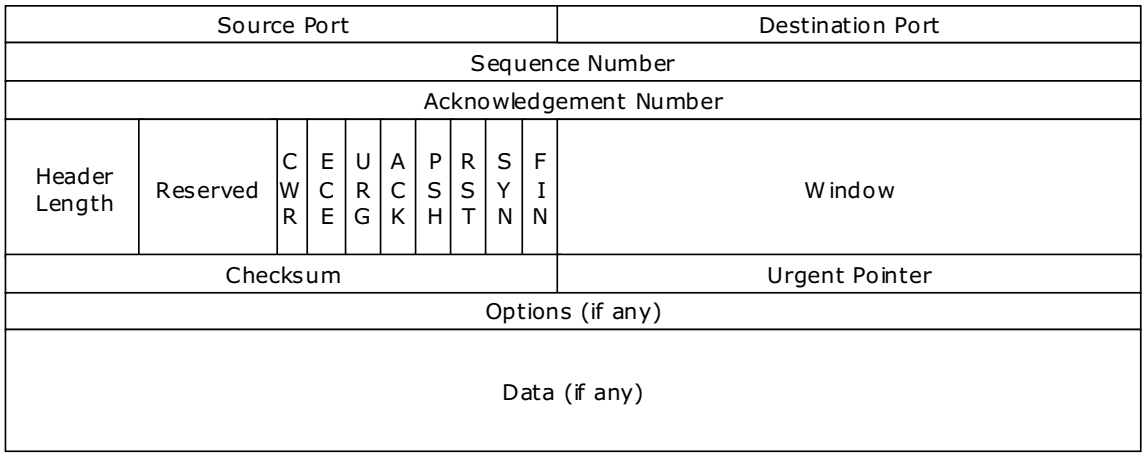

Figure 2.1: Standard TCP header

- **Source Port / Destination Port (16 bit each)** Each TCP segment contains the source and destination port number to identify the sending and the receiving application. These two values, along with the source and the destination IP addresses in the IP header, uniquely identify each *connection*. This combination of an IP address and a port number is sometimes called a *socket*. The term sockets also appears in the original TCP specification (oSC81)

- **Sequence Number (32 bit)** In order to realize retransmission, lost data must be identifiable. This is achieved via this field in TCP. This field does not count segments but bytes.
- **Acknowledgement Number (32 bit)** Since every byte that is exchanged is numbered, the *acknowledgement number* contains the next sequence number that the sender of the acknowledgement expects to receive. This is therefore the sequence number plus 1 of the last successfully received byte of data.
- **Header Length (4 bit)** gives the length of the header in 32-bit words. This is required because the length of the option field is variable. With this 4-bit field, TCP is limited to a 60 byte header. Without options the normal size is 20 bytes.
- **Reserved (4 bit)** Reserved for future use, must be zero.
- **Flags (8 bit)** There are flags bits in the TCP header. One or more can be turned on at the same time.
  - URG: The uregent pointer is valid
  - ACK: The acknowledgment number is valid
  - PSH: The receiver should pass this data to the application as soon as possible.
  - RST: Reset a connection
  - SYN: Synchronize sequence number to initiate a connection.
  - FIN: The sender is finished sending data
  - CWR: is used for ECN
  - ECE: is used for ECN
- **Window (16 bit)** TCP flow control mechanism. This is the number of bytes, starting with the one specified in the acknowledgement number field, that the receiver is willing to accept. This is a 16-bit field, limiting the window to 65535 bytes.
- **Checksum (16 bit)** This field covers the TCP segment: the TCP header and the TCP data. This is a mandatory field that must be calculated and stored by the sender, and then verified by the receiver. The TCP checksum is calculated similar to the UDP and IP checksum (Adler 32), using a pseudo header.
- **Urgent Pointer (16 bit)** The *urgent pointer* is valid only if the URG flag is set. TCP's urgent mode is a way for the sender to transmit emergency data to the other end.
- **Options (variable)** The most common *option* field is the maximum segment size option, called the *MSS*. This can be used by the receiver to inform the sender of the maximum allowed segment size at connection setup.
- **Data (variable)** This field carries the application data.

2.2 TCP Services

Even TCP and UDP use the same network layer, namely IP, TCP provides a totally different service to the application layer than UDP does. TCP provides a connection-oriented, reliable, byte stream service. The term connection-oriented means the two applications using TCP (normally considered a server and a client) must establish a TCP connection with each other before the can exchange data. TCP provides reliability by doing the following:

- The application data is broken into what TCP considers the best sized chunks to send. This is totally different from UDP, where each write by the application generates a UDP datagram of that size. The unit of information passed by TCP to IP is called **segment** .
- When TCP sends a segment it maintains a timer, waiting for the other end to acknowledge reception of the segment.
- When TCP receives data from the other end of the connection, it sends an acknowledgement.
- TCP maintains a checksum on its header and data. This is an end-to-end checksum whose purpose is to detect any modification of the data in transit. If the checksum is not valid, there is no Ack of receiving it.
- Since IP segments are transmitted as IP datagrams , and since IP datagrams can arrive out of order, TCP segments can arrive out of order.
- Since IP datagrams can get duplicated, a receiving TCP must discard duplicate data.
- TCP also provides flow control, each end of a TCP connection has a finite amount of buffer space, a receiving TCP only allows the other end to send as much data as the receiver has buffer for.

TCP's flow control is provided by each end advertising a **window size** . This is the number of bytes, starting with the one specified by the acknowledgment number field, that the receiver is willing to accept. This is a 16-bit field, limiting the window to 65635 bytes. The checksum covers the TCP segment: the TCP header and the TCP data. This is a mandatory field that must be calculated and stored by the sender, and then verified by the receiver. The most common option field is the maximum segment size option, called MSS . Each end of the connection normally specifies this option on the first segment exchanged. It specifies the maximum size of a segment that the sender sends.

2.2.1 Connection handling

The following listing shows a trace of the TCP connection establishment process.

```
1  0.000000  24.53.4.252 -> 12.6.230.5   TCP 49557 > http [SYN] Seq=1700072725
       Ack=0 Win=32768 Len=0
2  0.071592   12.6.230.5 -> 24.53.4.252  TCP http > 49557 [SYN, ACK] Seq
       =407926337 Ack=1700072726 Win=5792 Len=0
3  0.072185  24.53.4.252 -> 12.6.230.5   TCP 49557 > http [ACK] Seq=1700072726
       Ack=407926338 Win=33304 Len=0
4  0.072786  24.53.4.252 -> 12.6.230.5   HTTP GET /~mdailey/test.html HTTP/1.1
5  0.170926   12.6.230.5 -> 24.53.4.252  TCP http > 49557 [ACK] Seq=407926338
       Ack=1700072953 Win=6432 Len=0
6  0.177008   12.6.230.5 -> 24.53.4.252  HTTP HTTP/1.1 200 OK
7  0.236263  24.53.4.252 -> 12.6.230.5   TCP 49557 > http [FIN, ACK] Seq
       =1700072953 Ack=407926657 Win=33304 Len=0
8  0.342792   12.6.230.5 -> 24.53.4.252  TCP http > 49557 [FIN, ACK] Seq
       =407926657 Ack=1700072954 Win=6432 Len=0
9  0.343138  24.53.4.252 -> 12.6.230.5   TCP 49557 > http [ACK] Seq=1700072954
       Ack=407926658 Win=33304 Len=0
```

As mentioned before, TCP provides a connection-oriented, reliable, byte stream service. In order to do this, a logical connection has to be established, using a common **3-way handshake** .

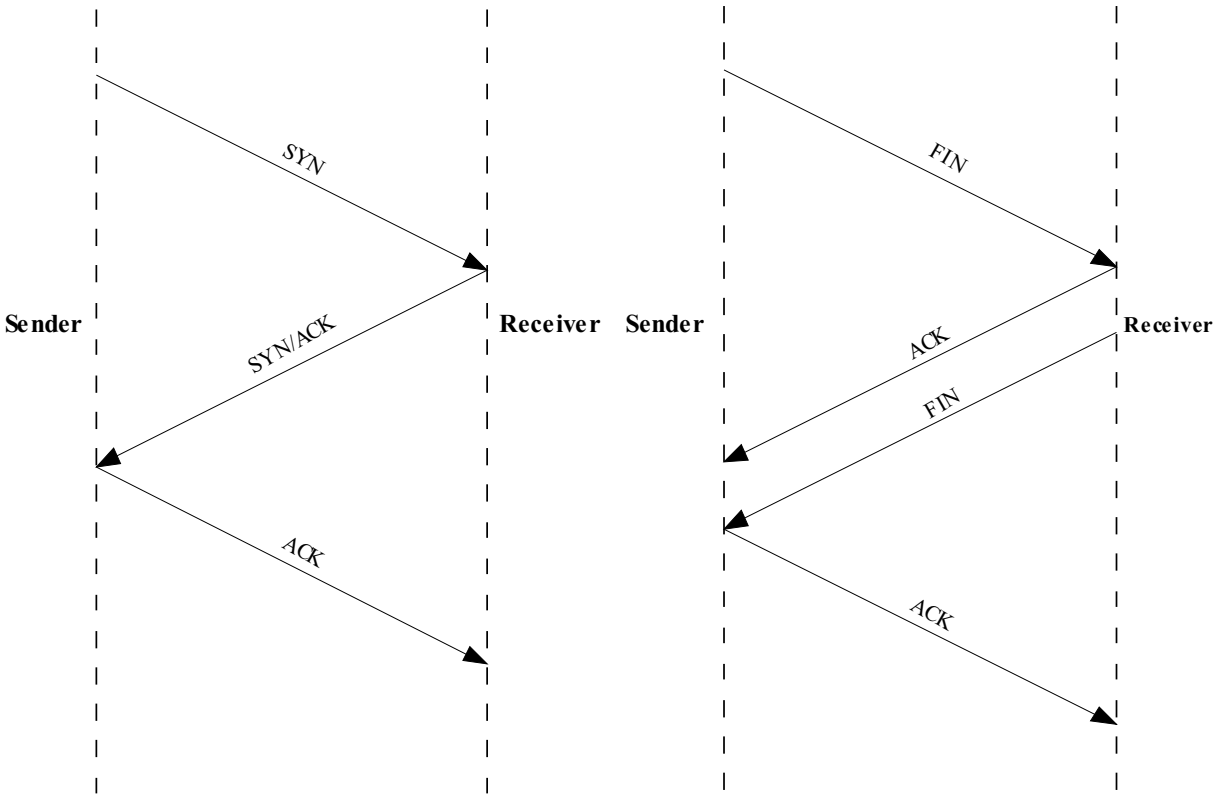

Figure 2.2: TCP connection establishment and TCP connection closing

2.2.2 Round Trip Time (RTT) estimation

When a host transmits a TCP packet to its peer, it must wait a period of time for an acknowledgment. If the reply does not come within the expected period, the packet is assumed to have been lost and the data is retransmitted. The obvious question - **How long do we wait**? - lacks a simple answer.

Over an Ethernet, no more than a few microseconds should be needed for a reply. If the traffic must flow over the wide-area Internet, a second or two might be reasonable during peak utilization times. If we're talking to an instrument packet on a satellite hurtling toward Mars, minutes might be required before a reply. There is no one answer to the question - How long?

All modern TCP implementations seek to answer this question by monitoring the normal exchange of data packets and developing an estimate of how long is "**too long**". This process is called Round-Trip Time estimation. RTT estimates are one of the most important performance parameters in a TCP exchange, especially when you consider that on an indefinitely large transfer, all TCP implementations eventually drop packets and retransmit them, no matter how good the quality of the link. If the RTT estimate is too low, packets are **retransmitted unnecessarily**; if it is too high, the connection can sit idle while the host waits to timeout.

2.2.3 Retransmission timeout (RTO) calculation

It is crucial to have a good estimate of the RTT [2.2.2] because it is used for calculating the retransmission timeout. Measure the elapsed time between sending a data octet with a particular sequence number and receiving an acknowledgement that covers that sequence number. This is the Round Trip Time (RTT). Next compute a Smoothed Round Trip Time (SRTT) as:

*SRTT= a * SRTT + (1 - a) * RTT*

and based on SRTT compute the RTO as:

*RTO = min(UBOUND, max(LBOUND,b * SRTT))*

where UBOUND is an upper bound on the timeout (e.g. 1 min), LBOUND is a lower bound on the timeout (e.g. 1 s), a is a smooting factor and b is a delay variance factor.

2.2.4 Flow control

The main goal of this principle is to protect the receiver from overload. TCP uses window-based flow control so called the **sliding window**. Figure [2.2.4] shows this principle. It means that the sender must not send more than the full **window** without waiting for acknowledgements at any time. A window is the maximum number of unacknowledged bytes that are allowed in any one transmission sequence, or to put it another way, it is the range of sequence numbers across the whole chunk of data that the receiver (the sender of the window size) is prepared to accept in its buffer. The corresponding RFC813 (Cla82) describes TCP window strategies in a more general context.

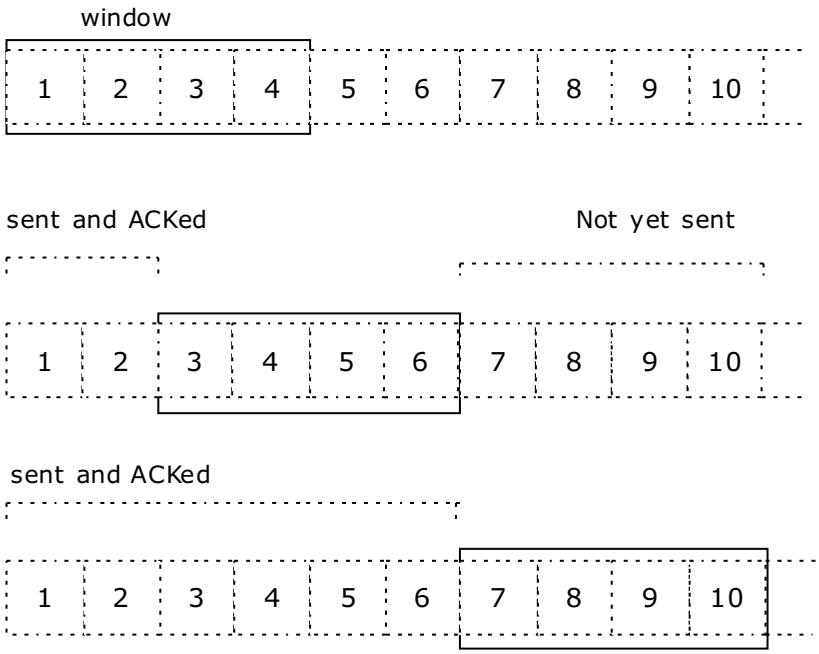

Figure 2.3: Flow control - sliding window

2.2.5 Timeouts and retransmissions

Because every TCP network has its own characteristics, the delay between sending a segment and receiving an acknowledgement varies. TCP achieves reliability by sending a segment, waiting for its ACK, and retransmitting it after a *while* if the ACK does not arrive. Is this value too small, a sender could spuriously retransmit a segment that is still in flight (or not yet acknowledged), while the value is to large has the sender wait too long and is therefore inefficient.

Timeouts and retransmissions is the core topic of this work, it is described in section [3.1] in much more detail.

2.3 TCP Congestion Control - Introduction

TCP has its own congestion control schema that contains two methods to limit the amount of data that a TCP sends into the network on the basis of the end-to-end feedback . Figure [2.3] shows this schema.

2.3.1 Slow start

Old TCPs would start a connection with the sender injecting multiple segments into the network, up to the window size advertised by the receiver. That is fine when the two hosts are on the same LAN but if there are routers and slower links between the sender and the receiver, problems can arise. Some intermediate router must queue the packets, and it is possible for that router to run out of space. The

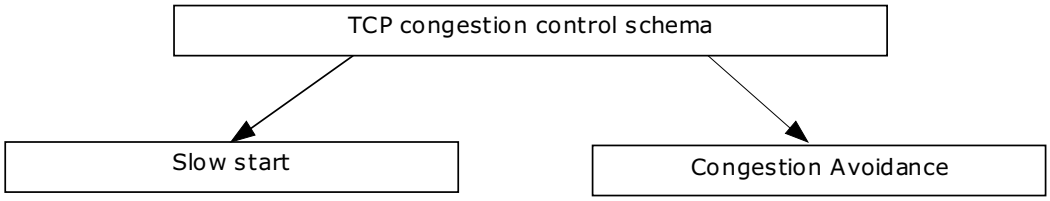

Figure 2.4: TCP congestion control schema

algorithm to avoid this is called slow start. It operates by observing that the rate at which new packets should be injected into the network is the rate at which the acknowledgments are returned by the other end. Slow start adds another window to the sender's TCP: the **congestion window**, called *cwnd* . When a new connection is established with a host on another network, the congestion window is initialized to **one** segment (i.e., the segment size announced by the other end, or the default, typically is 536 or 512). Each time an ACK is received, the congestion window is increased by one segment. The sender can transmit up to the minimum of the congestion window and the advertised window.
The congestion window is flow control imposed by the sender, while the advertised window is flow control imposed by the receiver. The former is based on the sender's assessment of perceived network congestion; the latter is related to the amount of available buffer space at the receiver for this connection. The sender starts by transmitting one segment and waiting for its ACK. When that ACK is received, the congestion window is incremented from one to two, and two segments can be sent. When each of those two segments is acknowledged, the congestion window is increased to four. This provides an exponential growth, although it is not exactly exponential because the receiver may delay its ACKs, typically sending one ACK for every two segments that it receives. At some point the capacity of the internet can be reached, and an intermediate router will start discarding packets. This tells the sender that its congestion window has become too large. Early implementations performed slow start only if the other end was on a different network. Current implementations always perform slow start.

2.3.2 Congestion avoidance

Congestion can occur when data arrives on a big pipe (a fast LAN) and is sent through a smaller pipe (a slower WAN). Congestion can also occur when multiple input streams arrive at a router whose output capacity is less than the sum of the inputs. Congestion avoidance is a way to deal with lost packets.

The assumption of the algorithm is that packet loss caused by damage is very small (much less than 1 percent), therefore the loss of a packet signals congestion somewhere in the network between the source and destination. There are two indications of packet loss: a **timeout** occurring and the receipt of **duplicate ACKs** (DUPACKs).

Congestion avoidance and slow start are independent algorithms with different objectives. But when congestion occurs TCP must slow down its transmission rate of packets into the network, and then invoke slow start to get things going again. In practice they are **implemented together**.

2.3.3 Slow start and Congestion avoidance together

Congestion avoidance and slow start require that the variables to be maintained for each connection: a **congestion window**, **cwnd** , and a **slow start threshold** size, **ssthresh**. The combined algorithm operates as follows:

1. Initialization for a given connection sets cwnd to one segment and ssthresh to 65535 bytes.
2. The TCP output routine never sends more than the minimum of cwnd and the receiver's advertised window. .
3. When congestion occurs (indicated by a timeout or the reception of duplicate ACKs), one-half of the current window size (the minimum of cwnd and the receiver's advertised window, but at least two segments) is saved in ssthresh. Additionally, if the congestion is indicated by a timeout, cwnd is set to one segment (i.e., slow start).
4. When new data is acknowledged by the other end, increase cwnd, but the way it increases depends on whether TCP is performing slow start or congestion avoidance.

If cwnd is less than or equal to ssthresh, TCP is in slow start; otherwise TCP is performing congestion avoidance. Slow start continues until TCP is halfway to where it was when congestion occurred (since it recorded half of the window size that caused the problem in step 2), and then congestion avoidance takes over.

In Slow start the cwnd begin with one segment, and increments by one segment every time an ACK is received. As mentioned earlier, it makes the growth of the window exponential by sending one segment, then two, then four, and so on. Congestion avoidance dictates that cwnd be incremented by segsize*segsize/cwnd each time an ACK is received, where segsize is the segment size and cwnd is maintained in bytes. This is a linear growth of cwnd, compared to slow start's exponential growth. The increase in cwnd should be at most one segment each round-trip time (regardless of how many ACKs are received in that RTT), whereas slow start increments cwnd by the number of ACKs received in a round-trip time.

Many implementations incorrectly add a small fraction of the segment size (typically the segment size divided by 8) during congestion avoidance. This is wrong and should not be emulated in future releases.

- if **cwnd** $<$ **sstresh** then do **slow-start**
- if **cwnd** $>$ **sstresh** then do **congestion avoidance**

2.3.4 Fast retransmit and fast recovery

These two algorithms were mainly intended as a solution for poor performance across **long fat pipes** (links with large bandwidth * delay product). The idea is to use **duplicate ACKs** (DUPACKs) as an **indication of packet loss**. For Example: if a sender transmits segments 1,2,3,4,5 and only 1,3,4,5 make it to the other end, the receiver will typically respond to segment 1 with an ACK 2 (i expect segment 2 now) and send three more such ACKs (duplicate ACKs) in response to segments 3,4 and 5.

TCP may generate an immediate acknowledgment (a DUPACK) when an out-of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected.

Since TCP **does not know** whether a DUPACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only **one or two** DUPACKs before the reordered segment is processed, which will then generate a new ACK. If **three or more** DUPACKs are received in a row, it is a strong indication that a segment has been **lost**. TCP then performs a retransmission of what appears to be the missing segment, **without waiting for a retransmission timer to expire**. This behavior is called **fast retransmit**.

After fast retransmit sends what appears to be the missing segment, **congestion avoidance, but not slow start is performed**. This is the **fast recovery algorithm**. It is an improvement that allows high throughput under moderate congestion, especially for large windows.

RFC2581 (MA99) specifies the combined implementation of **fast retransmit and fast recovery**. The following algorithm is directly taken from RFC2581 (MA99)

1. When the third duplicate ACK (DUPACK) is received, set ssthresh to no more than the value given in equation 3.
2. Retransmit the lost segment and set cwnd to ssthresh plus 3*SMSS. (SMSS is the size of the largest segment that the sender can transmit) This artificially *inflates* the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
3. For each additional DUPACK received, increment cwnd by SMSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
4. Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.
5. When the next ACK arrives that acknowledges new data, set cwnd to ssthresh (the value set in step 1). This is termed *deflating* the window. This ACK should be the acknowledgment elicited by the retransmission from step 1, one RTT after the retransmission (though it may arrive sooner in the presence of significant out-of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.

   Note: This algorithm is known to generally not recover very efficiently from multiple losses in a single flight of packets

2.3.5 Go-back-N

Go-Back-N ARQ is a specific instance of the Automatic Repeat-reQuest (ARQ) Protocol, in which the sending process continues to send a number of frames specified by a window size without receiving an ACK packet from the receiver.

The receiver process keeps track of the sequence number of the next frame it expects to receive, and sends that number with every ACK it sends. If a frame from the sender does not reach the receiver, the receiver will stop acknowledging received frames. Once the sender has sent all of the frames in its

window, it will detect that all of the frames since the first lost frame are outstanding, and will go back to sequence number of the last ACK it received from the receiver process and fill its window starting with that frame and continue the process over again.

2.4 TCP Variants

2.4.1 Tahoe TCP

Tahoe TCP was first implemented around 1988 and contains 2 major congestion control schemes:

- Slow Start and Congestion Avoidance
- Fast Retransmit

Tahoe TCP is very sensitive to packet loss. 1 percent packet loss rate may cause 50-75 percent decrease in throughput.

The major problem of TCP Tahoe is: Windows Size is set to minimum value after packet loss.
Every packet loss is assumed to be **serious congestion**. Initially, the connection increases its window using Slow-Start [2.3.1]. The sender detects a dropped packet after receiving three DUPACKs; this is the Fast Retransmit algorithm [2.3.4]. The sender invokes Slow-Start and later increases the window using the Congestion Avoidance [2.3.2] algorithm.

2.4.2 Reno and NewReno TCP

Reno is an improved version of TCP Tahoe. It was developed around 1990. Reno halves the congestion window on triple-ACK (DUPACKs). Most modern TCP's are Reno based:

- Slow Start and Congestion Avoidance
- Fast Retransmit
- Fast Recovery

Problem of TCP Reno: If two or more segments are lost in the **current window**, the Fast Recovery algorithm [2.3.4] cannot retransmit all lost packets. (TCP has to wait for a retransmission timeout)
The selective ACK [2.4.3] option can solve this problem.

New Reno was developed around 1996 and it refined fast retransmit/recovery when partial acknowledgements are available. NewReno is an improvement for Reno.

- improves performance against multiple packet loss in the window.
- does not need Selective ACK
- requires modification to only data sender
- NewReno is a bit more aggressive scheme than Reno

2.4.3 TCP Selective Acknowledgement Option - SACK TCP

The Selective Acknowledgment Option - SACK is defined in RFC2018 (Mat96).
TCP may experience poor performance when **multiple packets** are lost from one window of data (TCP throughput). TCP uses a cumulative acknowledgement scheme. With the limited information available from cumulative acknowledgments, a TCP sender can only learn about a single lost packet per round trip time. This forces the sender to either wait a round trip time [2.2.3] to find out each lost packet, or to unnecessarily retransmit segments which have been correctly received. An aggressive sender could choose to retransmit packets early, but such retransmitted segments may have already been successfully received. SACK is a strategy which corrects this behavior in face of **multiple dropped packets**. A SACK mechanism, combined with a selective repeat retransmission policy, can help to overcome these limitations. The receiving TCP sends back SACK packets to the sender informing the sender of data that has been successfully received. The sender can then **retransmit only the missing data segments**. So the sender needs to retransmit only the segments that have actually been lost.

SACK uses two TCP options. The first is an enabling option called SACK-permitted, which may be send in a SYN segment to indicate that the SACK option can be used once a connection is established. The other is the SACK option itself, which may be sent over an established connection once permission has been given by SACK-permitted. The SACK option is included in segments from the receiving TCP to the sending TCP.

SACK-Permitted Option

The 2 byte TCP Sack-Permitted Option may be sent in a SYN segment, once the connection is established. It must not be sent on non SYN segments. The SACK option is to be used to convey extended acknowledgement information from the receiver to the sender over an established TCP connection.

TCP SACK Option

The SACK Option, see figure [2.5], is used by the data receiver to inform the data sender of non-coherent blocks of data that have been received and queued. The SACK option **does not change** the meaning of the Acknowledgement Number field in the TCP header [2.1].

Each coherent block of data queued at the data receiver is defined in the SACK option by two 32-bit unsigned integers in network bytes order (Figure [2.5]):

- Left Edge of Block - the first sequence number of this block
- Right Edge of Block - the next sequence number of the last sequence number of this block

Each block represents received bytes of data that are coherent and isolated. In other words bytes just below the block (Block -1) and just above (Right Edge of the Block) have not been received.

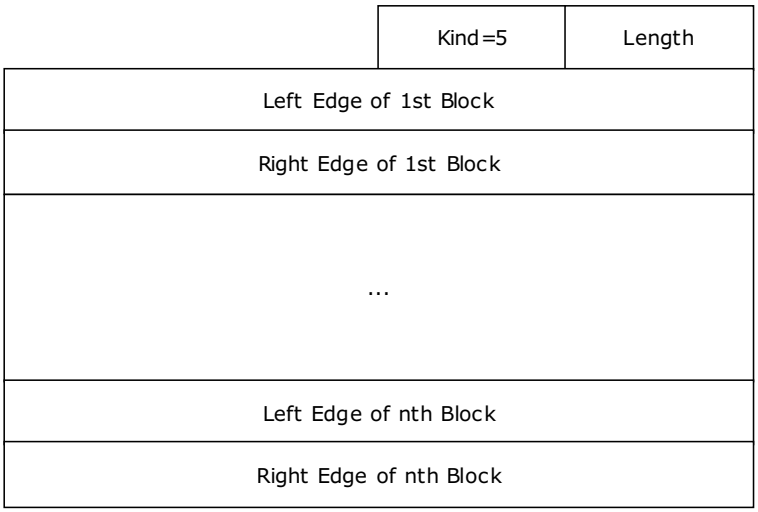

Figure 2.5: TCP SACK Option

Receiver Behavior

When the data receiver has received a SACK-Permitted option on a SYN packet for this connection, the data receiver may generate SACK options. If there is no SACK-Permitted option on a SYN packet for this connection, it must not send SACK options on that connection.
When the data receiver want to send a SACK option the following rules apply:

- The data receiver should include as many as possible distinct SACK blocks in the SACK option.
- The first SACK block must specify the contiguous block of data containing the segment which triggered this ACK, unless that segment advanced the Acknowledgment Number field in header.
- The SACK option should be filled out by repeating the most recently reported SACK blocks that are not subsets of SACK block already included in the SACK option being constructed.

Sender Behavior

When the sender receives an ACK containing a SACK option, the data sender should record the selective acknowledgement for future reference. The sender is assumed to have a retransmission queue that contains the segments that have been transmitted but not yet acknowledged, ordered by sequence numbers. When an acknowledgment segment arrives containing a SACK option, the data sender will turn on the SACKed bits for segments that have been selectively acknowledged.

SACK Option Examples

Assume that the left window edge is 5000 and that the data transmitter sends a bust of 4 segments, each containing 500 bytes of data.

- The first 2 segments are received but the last 2 are dropped: The data receiver will return a normal TCP ACK segment acknowledging sequence number 6000, with no SACK option.
- The first segment is dropped but the remaining 3 are received: The data receiver will return a TCP ACK segment that acknowledges sequence number 5000 and contains a SACK option specifying one block of queued data.

Triggering Segment	ACK	Left Edge	Right Edge
5000	(lost)	5500	6000
5500	5000	5500	6500
6000	5000	5500	7000
6500	5000	5500	7500

There are more blocks if the count of segments for example is 8 and segments 2,4,6 and 8 are dropped. Such an example is specified in RFC2018. (Mat96)

2.5 RED

RED - Random Early Detection is one mechanism for Active Queue Management that has been proposed to detect incipient congestion and is currently deployed in the Internet. (rfc98) Random Early Detection gateways- congestion avoidance in packet switched networks. The gateway detects incipient congestion by computing the **average queue size**. The gateway could notify connections of congestion either by dropping packets arriving at the gateway or by setting a bit in packet header, the congestion experienced codepoint CE [2.6]. When the average queue size exceeds a present threshold, the gateway drops or marks each arriving packet with a certain probability, where the exact probability is a function of the average queue size.

2.6 Addition of Explicit Congestion Notification - ECN to IP

ECN: Explicit Congestion Notification (RFC3168 (Ram01)) is a TCP feature which provides a method for an intermediate router to notify the end hosts of **impending network congestion**. It also provides enhanced support for TCP sessions associated with applications that are sensitive to delay or packet loss including Telnet, web browsing and transfer of audio and video data. The main benefit of this feature is the **reduction of delay and packet loss** in data transmission. Advantages for using routers:

- End nodes can only determine congestion by sensing packet losses
- Router knows more about congestion than end nodes. (If queue length is the router exceeds a certain threshold, it can be assumed that the network is becoming congested)

2.6.1 ECN in IP

There is an ECN field user in the IP header with two bits, making four ECN codepoints.

ECT	CE	
0	0	Not ECN
0	1	ECT(1)
1	0	ECT(0)
1	1	CE

The ECN capable Transport (ECT) codepoints [10],[01] are set by the data sender to indicate that the end-points of the transport protocol are **ECN capable**.
Routers treat ECT(0) and ECT(1) as equivalent. The not ECN codepoint [00] is use to indicate that the packet is not using ECN. The CE codepoint [11] is set by the router to indicate congestion to the end nodes.

2.6.2 ECN in TCP

ECN-Echo Flag: ECE
CWR Flag : to enable the TCP receiver to determine when to stop setting the ECE flag.

TCP Initialization

- No ECN: sender sets ECN codepoint to [00] and receiver ignores the CE codepoint in the received packet.
- ECN: Host A: is the initiating Host and Host B is the responding Host. Now there are 4 types of different packets.
  - ECN-setup SYN packet: SYN packet with ECE and CWR flag set
  - Non-ECN-setup SYN packet: at least one of the ECE or CWR flag is not set
  - ECN-setup SYN-ACK packet: ECE flag set but CWR not
  - non-ECN-setup SYN-ACK packet: SYN-ACK with any other configuration

Before a TCP connection can use ECN: Host A sends an ECN-setup SYN packet and Host B sends an ECN-setup SYN-ACK packet.

Reaction with ECN

- Receiver: responds to incoming packet that have set the CE codepoint by setting ECE in outgoing ACKs.
- Sender: responds to incoming data packets with ECE flag on by reducing congestion windows and setting CWR.

Rules for sending Host with ECN-setup packets

- If a host has received an ECN-setup SYN packet, then it MAY send an ECN-setup SYN-ACK packet. Otherwise, it MUST NOT send an ECN-setup SYN-ACK packet.

- A host MUST NOT set ECT on data packets unless it has sent at least one ECN-setup SYN or ECN-SYN-ACK packet, and has received at least one ECN-setup SYN or ECN-setup SYN-ACK packet, and has sent no non-ECN-setup SYN or non-ECN setup SYN-ACK packet. If a host has received at least one non-ECN setup SYN or non-ECN setup SYN-ACK packet, then it SHOULD NOT set ECT on data packets.
- If a host ever sets the ECT codepoint on a data packet, then that host MUST correctly set/clear the CWR TCP bit on all subsequent packets in the connection.
- If a host has sent at least one ECN-setup SYN or ECN-setup SYN-ACK packet, and has received no non-ECN setup SYN or non-ECN setup SYN ACK packet, then if that host receives TCP data packets with ECT and CE codepoints set in the IP header, then that host MUST process these packets as specified for an ECN-capable transport.
- A host that is not willing to use ECN on a TCP connection SHOULD clear the ECE and CWR flag in all non-ECN setup SYN and/or SYN-ACK packets that it sends to indicate this unwillingness.
- A host MUST NOT set ECT on SYN or SYN-ACK packets.

2.6.3 Sequence of events in an ECN based reaction to congestion

- An ECT codepoint is set in packets by the sender to indicate that ECN is supported by the transport entities.
- An ECN capable router detects impending congestion and detects that the ECT codepoint is set in the packet it is about to drop. Instead of dropping the packet, the router sets the CE codepoint in the IP header and forward the packet.
- The receiver receives the packet with the CE codepoint set, and sets the ECE flag in its next TCP-ACK sent to the sender.
- The sender receives TCP ACK with ECE flag and reacts to the congestion as if a packet has been dropped.
- Sender sets the CWR-flag of the next packet sent to the receiver to acknowledge the receipt of ECE.

2.6.4 ECN with one bit Nonce

ECN-nonce (NS03) is an **optional addition to Explicit Congestion Notification Mechanism** (Ram01) that enables a router to signal congestion to the sender without trusting the receiver or other network devices along the signaling path. The ECN-nonce uses the two ECN-Capable Transport (ECT)codepoints in the ECN field of the IP header, and requires a flag in the TCP header. Without this mechanism, ECN-based transports **can be manipulated** to undermine congestion control. The correct operation of ECN requires the cooperation of the receiver to return Congestion Experienced signals (CE bit) to the sender, but the protocol lacks a mechanism to enforce this cooperation.

This raises the possibility that an unscrupulous or poorly implemented receiver could always clear ECN-Echo (ECE flag) and simply not return congestion signals to the sender. This would give

the receiver a **performance advantage**.

Explicit Congestion Control with active queue management in the form of RED [2.5] gateways, has been proposed as a standard mechanism to improve congestion control in the Internet. With ECN routers are able to mark packets to signal congestion, as well as simply drop them during congestion. This **avoids loss and improves performance**.

Figures [2.6] and [2.7] show the ECN behavior and how ECN can manipulated. Figure [2.6] shows the correct behavior of ECN by allowing routers to signal congestion by setting the Congestion Experienced (CE) bit in the IP header. In TCP, the receiver returns congestion signals by setting the ECN-Echo (ECE) flag. The sender sets the Congestion Window Reduction (CWR) flag to inform the receiver that it has **reacted to congestion**. Figure [2.7] shows a wrong scenario. By hiding congestion signals TCP receiver can cajole the sender into **increasing the congestion window**. Because the data packets are ECN-capable, they will not be dropped by the router until the link becomes congested.

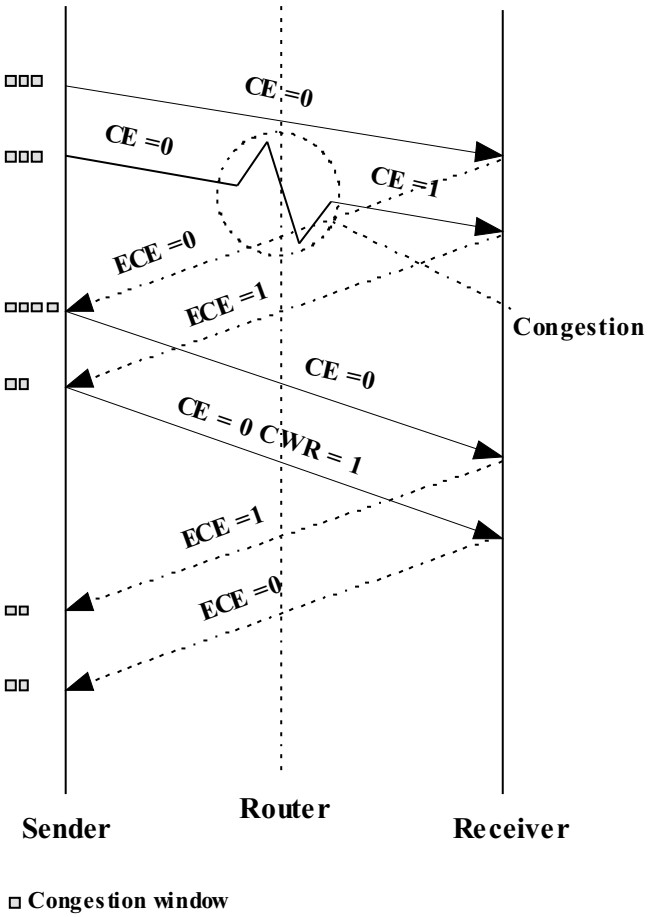

Figure 2.6: ECN with one bit nonce - correct Receiver behavior

The solution with with ECN-nonce has following properties:

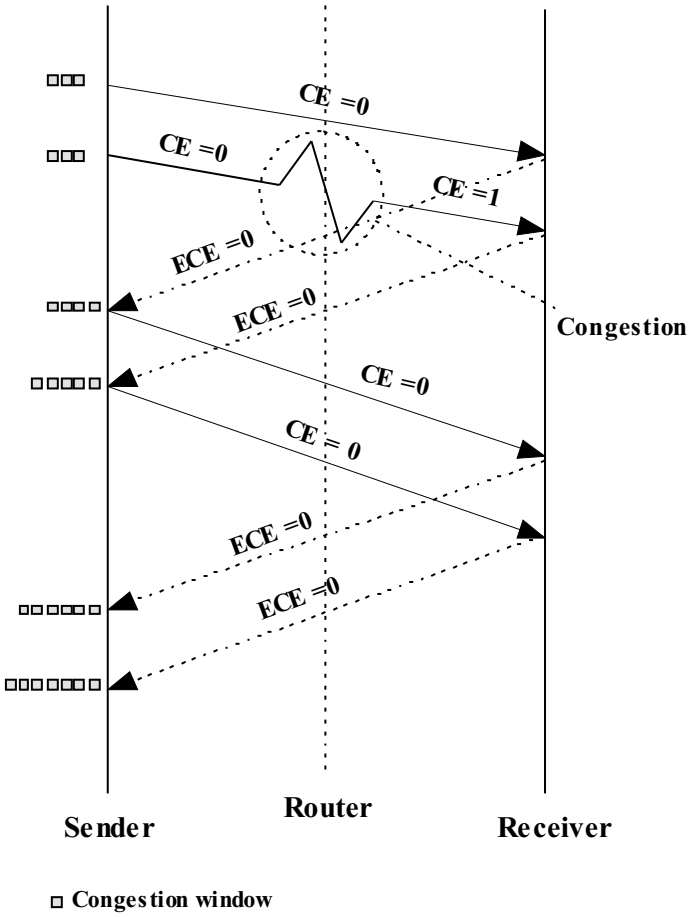

Figure 2.7: ECN with one bit nonce - incorrect Receiver behavior

- It does not remove any of the benefits of ECN
- It detects misbehavior with a high probability
- It never falsely implicates behaving receivers and networks
- It is efficient in terms of packet overhead, per-packet processing

A one-bit nonce per packet is enough to **detect misbehavior** in the congestion control loop, since each mark of congestion is a separate trial. A random one-bit value (a nonce) is encoded in the two ECT codepoints. The **one-bit sum** of these nonces is returned in a TCP header flag, the nonce sum (NS) bit .
Each acknowledgement carries a nonce sum, which is the one bit sum (exclusive-or, or parity) of nonces over the byte range represented by the acknowledgement. The sum is used because not every packet is acknowledged individually, nor are packets acknowledged reliably. If a sum is not used, the nonce in an unmarked packet could be echoed to prove to the sender that the individual packet arrived unmarked.

However, since these ACKs are not reliably delivered, the sender could not distinguish a lost ACK from one that was never sent in order to conceal a marked packet. The nonce sum prevents the receiver from concealing individual marked packets by not acknowledging them. Because the nonce and nonce sum are both one bit quantities, the sum is no easier to guess than the individual nonces.

Figure [2.8] shows how ECN-with one-bit nonce works.

1. The sender attaches a random nonce (N=0 or N=1) to each packet.
2. The receiver always returns the one-bit cumulative nonce, which is the one-bit sum of these nonces.
3. When a router marks the congestion experienced bit (CE) in the IP header, the nonce is cleared.
4. The sender can verify that no congestion signals are concealed by checking that the sum reported by the receiver is correct.
5. When congestion is experienced the receiver's nonce sum will be incorrect and is ignored.
6. and the sender must resynchronize with the receiver's sum by resetting its own sum to the receiver's sum.

The Experimental Standard ECN (RFC2481 (Ram99)) encoding is compared with the Proposed Standard ECN which supports the ECN-nonce. The ECN-Capable Transport (ECT) codepoint 10 and 01 are called ECT(0) and ECT(1)

NONCE	RFC2481	PROPOSED
00	ECN-incapable	ECN-incapable
10	No congestion	No congestion, Nonce=0 – ECT((0))
01	undefined	No congestion, Nonce=1 – ECT(1)
11	Congestion	Congestion, Nonces cleared

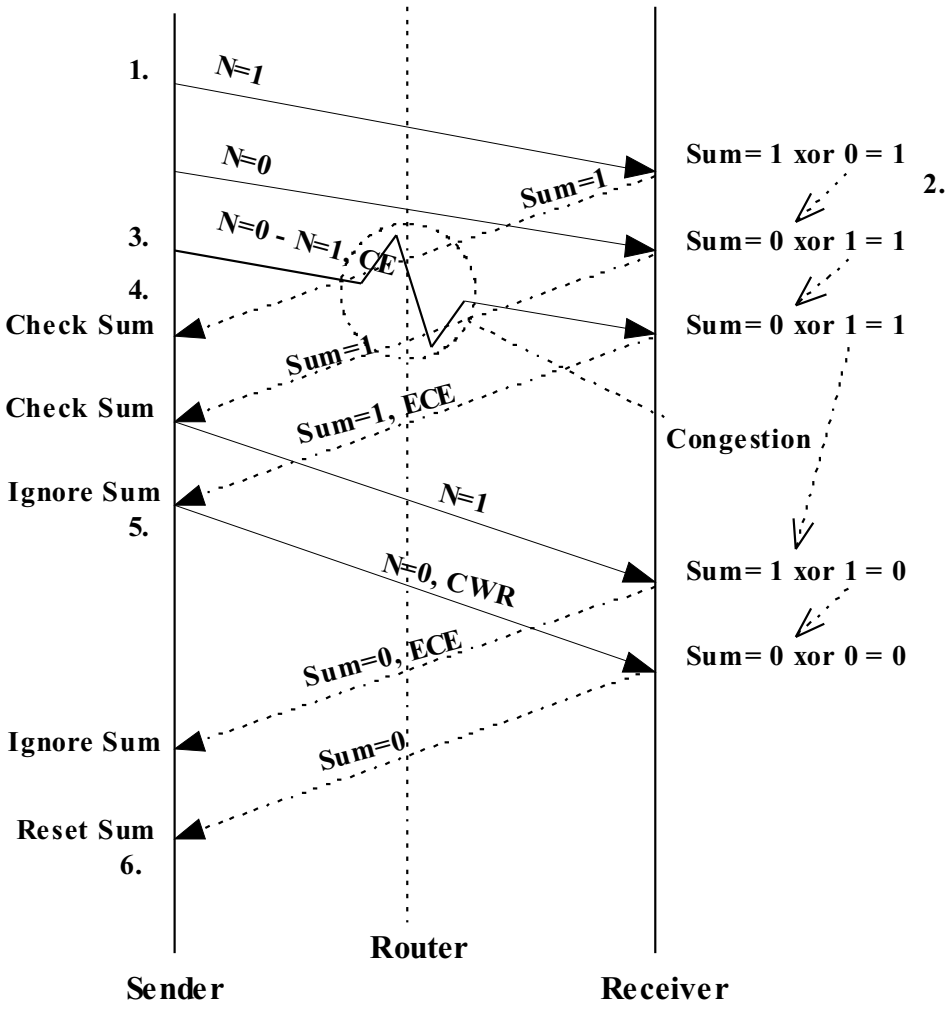

Figure 2.8: ECN with one-bit nonce

Sender Behavior

Senders manage CWR and ECN-Echo as specified in RFC3168 (Ram01). In addition, they must place nonces on packets as they are transmitted and **check the validity** of the nonce sums in acknowledgments as they are received. To place a one bit nonce value on every ECN-capable IP packet, the sender uses the two ECT codepoints: ECT(0) represents a nonce of 0 and ECT(1) a nonce of 1. As in ECN, retransmissions carry no nonce.

Router Behavior

Routers behave as specified in RFC3168 (Ram01). By marking packets to signal congestion, the original value of the nonce, in ECT(0) or ECT(1), is removed. Neither the receiver nor any other party can unmark the packet without successfully guessing the value of the original nonce.

Receiver Behavior

ECN-nonce receivers **return the current nonce sum** in each acknowledgement. Receiver behavior is otherwise unchanged from RFC3168 (Ram01). Returning the nonce sum is optional, but recommended, as senders are allowed to discontinue sending ECN-capable packets to receivers that do not support the ECN-nonce. As packets are removed from the queue of out-of-order packets to be acknowledged, the nonce is recovered from the IP header. The nonce is added to the current nonce sum as the acknowledgement sequence number is advanced for the recent packet. In the case of marked packets, one or more nonce values may be unknown to the receiver. In this case the missing nonce values are ignored when calculating the sum (or equivalently a value of zero is assumed) and ECN-Echo will be set to signal congestion to the sender. Returning the nonce sum corresponding to a given acknowledgement is straightforward. It is carried in a single ”NS” (Nonce Sum) bit in the TCP header. This bit is adjacent to the CWR and ECN-Echo bits, set as Bit 7 in byte 13 of the TCP header, as shown in Figure [2.9]:

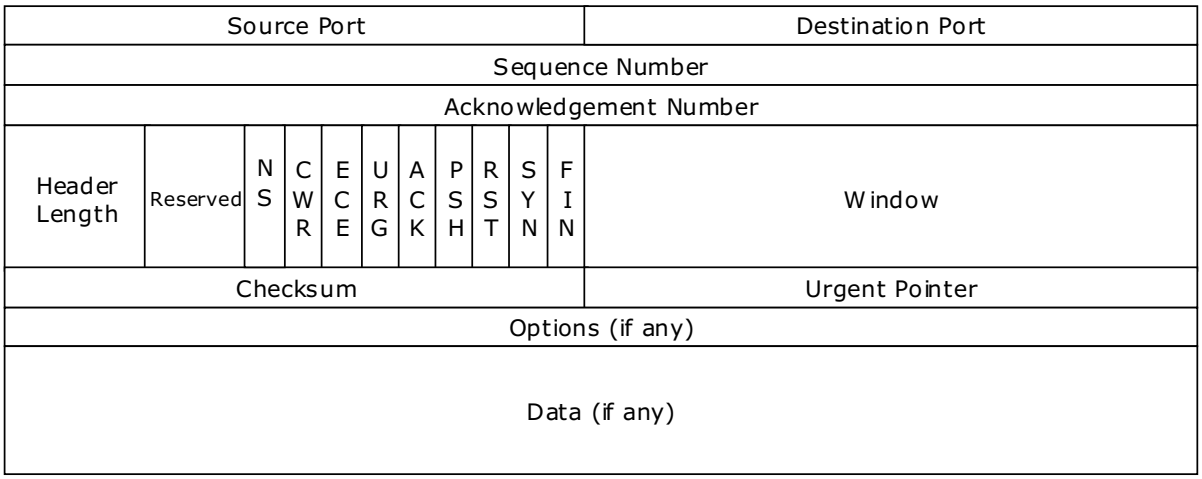

Figure 2.9: TCP header with the ECN-nonce bit

Chapter 3

Spurious Loss Events in TCP/IP

3.1 The Problem

The TCP Protocol is one of the core communication protocol in the Internet today. A lot of research is done in the past, to make TCP probably the best communication protocol of its kind. But there are some cases where TCP is improvable. Such a case is called spurious timeouts. A spurious timeout is a timeout that would not have occurred when the sender had waited long enough for the Acknowledgement.

A TCP receiver sends two types of Acknowledgements when receiving segments from a sender.

- **positive Acknowledgements** - for segments that are received correct and in-order.
- **duplicate Acknowledgements** - for segments that are received correct but out-of-order. A DUPACK acknowledges the same sequence number than the last sent ACK acknowledged.

As you can see in Figure [3.1] a TCP sender uses two different error recovery strategies.

1. timeout based retransmission
2. DUPACK based retransmission

After a timeout based retransmission (1) a TCP sender decreases its load (the maximum number of unacknowledged segments it may send into the network per RTT [2.2.2]) to 1 segment. It then enters the slow start phase (3), where it increases its load exponentially until the load reaches one half of what its value was before the timeout occurred. Then the TCP sender enters the congestion avoidance phase (4) where it increases the load linearly.
Spurious timeouts affect TCP performance in two ways: (1) The TCP Sender reduces its load unnecessary and (2) the TCP Sender is forced into a go-back-N retransmission mode.

Congestion Control is implemented in TCP through the use of two variables, slow start threshold and congestion window [2.3.3].

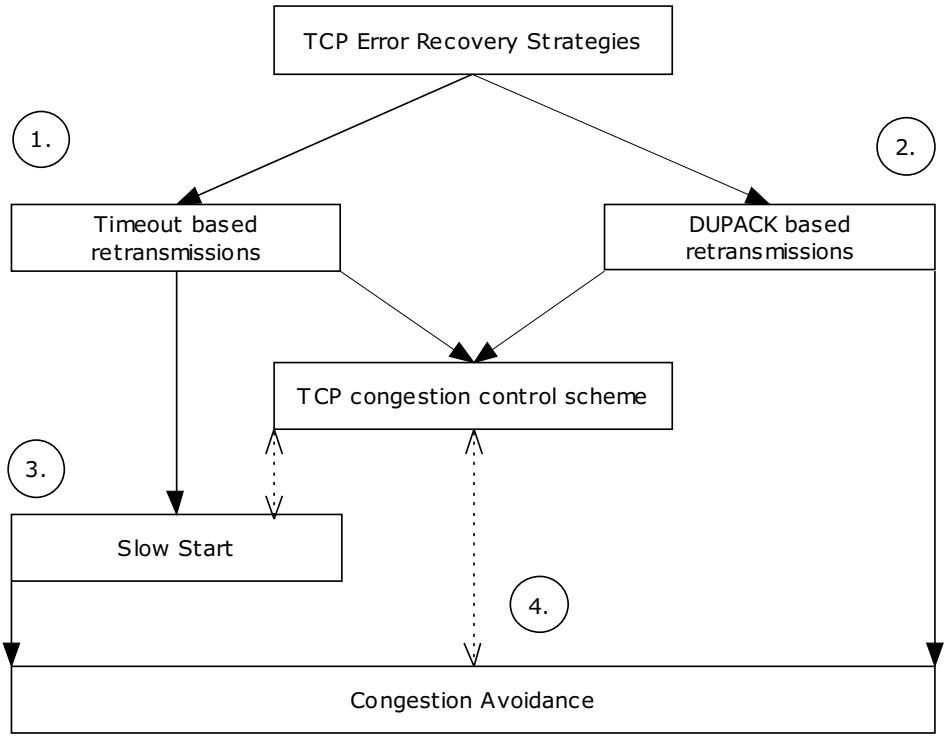

Figure 3.1: TCP error recovery strategies

Let us take a closer look to the problem of spurious timeouts. The main problem which leads to the go-back-N retransmissions is the retransmission ambiguity. On receipt of the first ACK after a timeout, the sender must interpret this ACK as **acknowledging the retransmission** (not really the original ACK), and must assume that all other outstanding segments **have also been lost**. Alternatively, if the ACK acknowledges a packet that has not yet been retransmitted, this ACK is ignored. In any case the sender enters the **slow start phase** and retransmits outstanding segments immediately. This continues until the entire window has been transmitted.

After a DUPACK based retransmission (2) a so called **fast retransmit** [2.3.4] is triggered. This occurs when three successive DUPACKs for the same sequence number have been received (without waiting for the retransmission timer to expire [2.2.3]). After such a retransmission the TCP sender enters immediately the **congestion avoidance phase** (4). This behavior is also called **fast recovery** [2.3.4].

3.1.1 Spurious loss event

A single packet is identified as lost (Fast Retransmit / Fast Recovery) even though it is a falsity.

3.1.2 Spurious Timeouts

A timeout is considered spurious if it **would have been avoided** had the sender waited longer for an acknowledgment to arrive (Lud03). A spurious timeout occurs when the RTT [2.2.2] **suddenly**

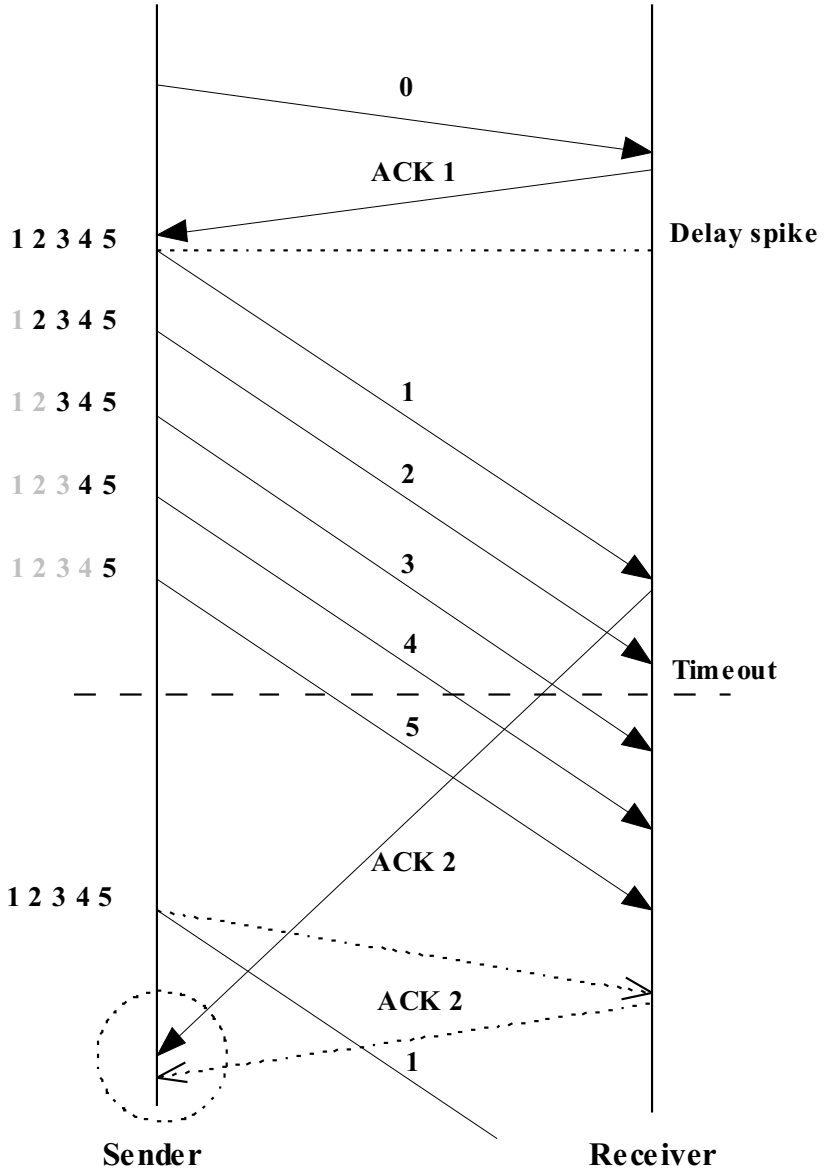

Figure 3.2: A spurious timeout

increases, to the extent that it exceeds the retransmission timer that had determined a priori. Spurious timeouts can be occur due to route changes or rapidly increasing congestion at the bottleneck link.

Figure [3.2] shows how spurious timeouts can accrue:

- Sender transmits segments 1 to 5
- A sudden delay spike happens, thus the receiver is not able to acknowledge segments 1 to 5 immediately
- A timeout for segment 1 occurs
- The sender retransmits segment 1
- An ACK that acknowledges reception of segment 1 arrives (in this figure that ACK relates to the original transmitted segment 1)
- The sender now is not able to distinguish those 2 ACKs (it is an acknowledgement for the original segment 1 or an acknowledgement for the retransmitted segment 1)

This ambiguity leads the following two cases:

- The ACK relates to the retransmitted segment 1.
- The ACK relates to the original transmitted segment 1, the timeout was spurious

Due to a spurious timeout TCP must assume that all other outstanding segments are lost too. Thus the sender enters the **slow start phase** and retransmits all outstanding segments in this fashion. The go-back N retransmission triggers the next problem: The receiver generates DUPACK for every segment received more than once.This may trigger a spurious fast retransmit at the sender [3.1.3].
Another problem is the **distortion of congestion control**: The congestion windows and the slowstart threshold (ssthresh) are reduced after a spurious timeout. This **reduction is unnecessary** as no data loss occurs in the network, that would indicate congestion.

By default a TCP sender is **not able** to distinguish those two cases. Spurious timeouts affect TCP performance in two ways:

- the TCP sender reduces its load unnecessarily
- the TCP sender is forced into a go-back N retransmission mode

3.1.3 Spurious Fast Retransmits

The IP protocol is connection-less and therefore it does **not guarantee** in-order delivery of packets. The sequence of packets as generated by the source does not need to be preserved when the packets are delivered to the destination. That responsibility is left to TCP. The fact that IP does not care about the order of the packets provides also more memory-efficient link layer implementations. A so called **packet re-ordering** event happens when a packet arrives the receiver after one or more packets, that had left the source later, have already arrived. (the length is the count of out-of-order packets) For example, if packets 1-10 are sent but packet 1 arrives at last, then the re-ordering length is 9. Packet re-ordering with a re-ordering length greater or equal to the DUPACK threshold (with DUPACK based error recovery) causes a spurious fast transmission.

3.2 Methods of resolution

3.2.1 EIFEL Algorithm

The Eifel Detection Algorithm is specified in RFC3522. (Lud03)

- **Advantages**: works always with spurious timeouts, and also with spurious loss events. This algorithm detects Errors immediately.

- **Disadvantages**: needs the timestamp option, (Jac92) and so it produces more overhead in the TCP header.

The Eifel Algorithm is an enhancement to TCP's error recovery scheme. It **eliminates the retransmission ambiguity**, thereby solving the problems caused by spurious timeouts and spurious fast transmits. It can be incrementally deployed as it is backward compatible and does not change the TCP's congestion control semantics. It can improve the end-to-end throughput in environments where spurious timouts occur frequently by several tens of percent. Another key of Eifel is that it provides an more optimistic retransmission timer because it reduces the penalty of a spurious timeout to a single (common case) spurious retransmission.

Eliminating the retransmission ambiguity requires extra information in ACKs that the sender can use to unambiguously distinguish an ACK for the original transmission of a segment from that of a retransmission. Eifel uses therefore the TCP timestamp option. The sender writes current value of the timestamp clock **in each outgoing segment**. The receiver then echos those ACKs according to the rules defined in (Jac92).

Eliminating the retransmission ambiguity is then implemented as follows. The sender always stores the timestamp of the first retransmission triggered by an expiration of the retransmission timer. Then, when the first ACK that acknowledges the retransmission arrives, the sender compares the timestamp of that ACK with the stored value. If it is smaller than the stored values, this indicates that the retransmission was spurious. One disadvantages of the Eifel Algorithm is the overhead, because in each segment there are 12 bytes only for the timestamp option. But the advantage of the timestamp option is already a proposed standard and that it is widely deloyed (All00a).
Existing TCP/IP header compression schemes (MD99) (Jac90) do not support compression of TCP options, but there is ongoing work to enable it.

The original Eifel proposal simply specified that the retransmission after detecting a spurious timeout always resumes with the next unsent segment (Lud03). This method works fine when none of the outstanding segments are lost, but in reality delay spikes are often coupled with data losses. RFC4015 (AG05) proposes a little bit different way for this behavior.

Recovery methods :

- *A single lost segment*: in this simple case one delayed segment is lost. This scheme only needs three DUPACKs to enter the loss recovery phase. The transmission is resumed by the Eifel Algorithm with the next unsent segment (if fast retransmit is allowed in TCP Reno). There are no unnecessary retransmissions and thus a DUPACK series can only indicate a lost segment.

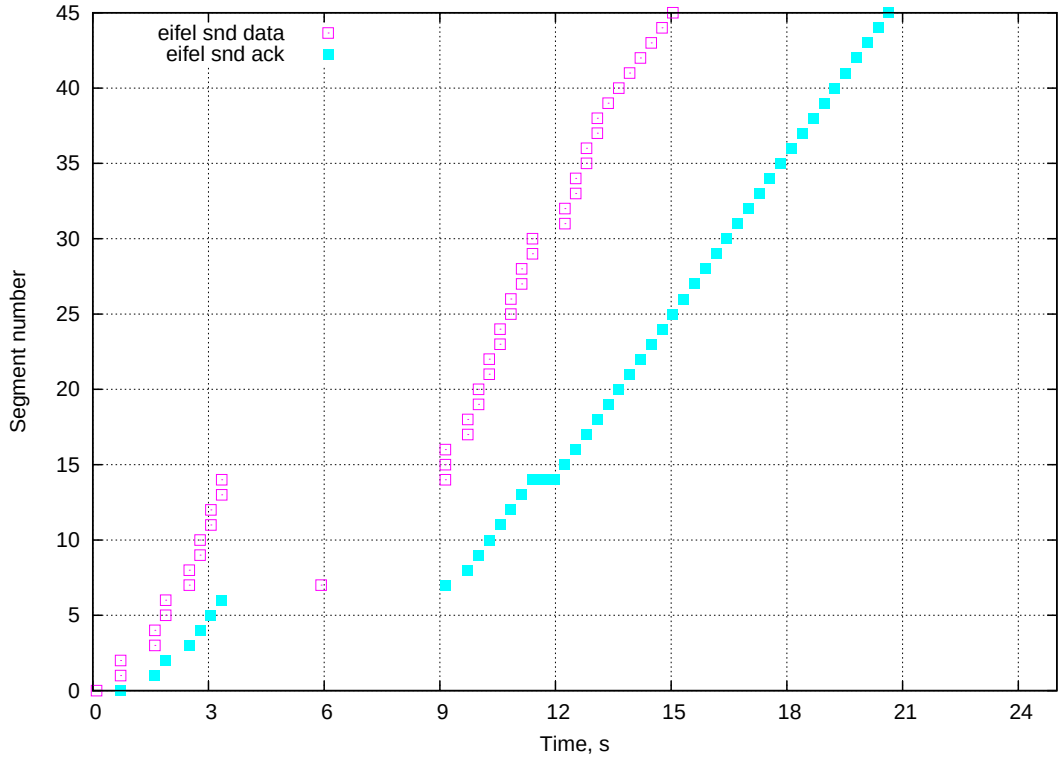

Figure 3.3: Response of TCP Reno with Eifel to a spurious timeout

- *Loss of all but one segment*: This case is described in RFC 4015 (AG03). In this case all segments are lost except the oldest segment that is delayed. It is shown that NewReno-SACK and FACK TCP are the best solutions for such a scenario.

There are also 3 methods how to restore the Congestion Control State after a spurious timeout. The paper (Lud03) proposed the following options for restoring the congestion control state:

- ssthresh=sstresh-old, cwnd=cwnd-old
- ssthresh=cwnd-old/2, cwnd=ssthresh
- ssthresh=cwnd-old/2, cwnd=1

The first option is for complete restoration. Here the slow start threshold and the congestion windows are restored to the values stored before the timeout. The second option is for partial restoration. (as done normally by TCP) The third option is for no restoration of the slow start threshold and congestion window. Figure [3.4] shows these different options, proposed by RFC 3522 (Lud03)

In practice option 2 and 3 do not perform well, so option 1 should in general be the best solution, because if the congestion window is not restored fully, the sender cannot transmit on original ACKs after a spurious timeout.

The Eifel Algorithm works fine, but a big disadvantage is the additional overhead for the timestamp option in the TCP Header. It detects spurious timeouts and also spurious loss events immediately.

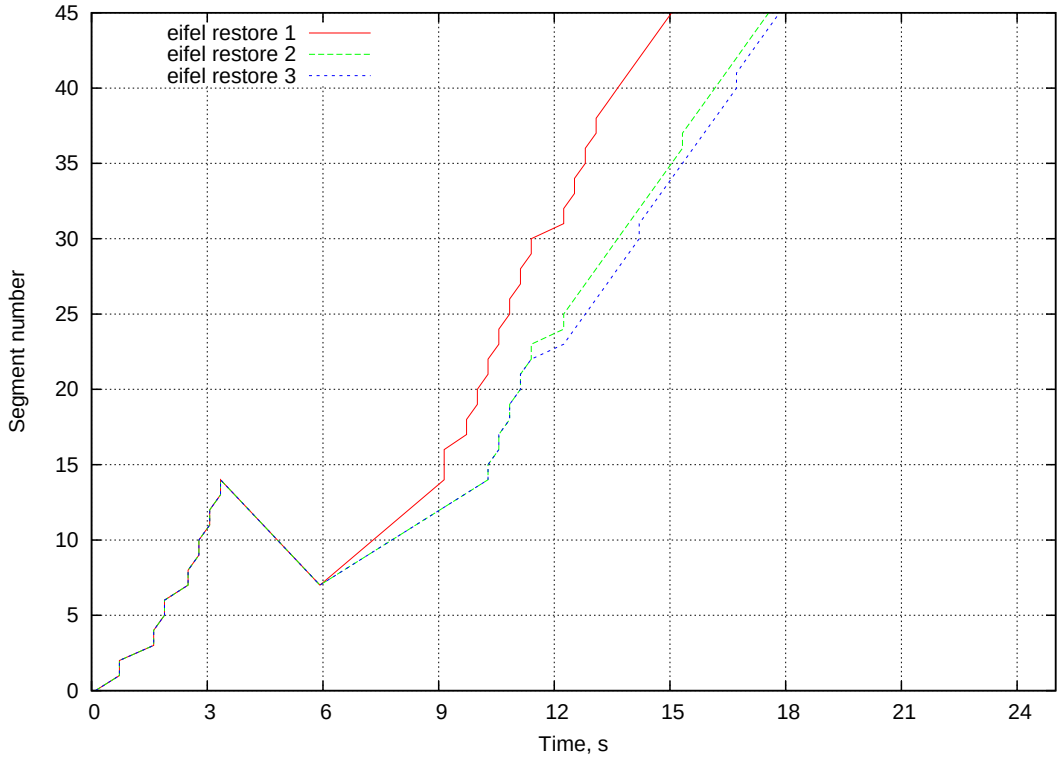

Figure 3.4: Eifel: 3 different options for restoring the congestion control state

3.2.2 D-SACK

- **Advantages**: does not need the timestamp option and therefore there is no extra overhead in the TCP header. Discovers also spurious loss events.
- **Disadvantages**: Needs more time to react than EIFEL

The Selective Acknowledgement (SACK) option defined in RFC 2018 (Mat96) is used by the TCP data receiver to acknowledge non-contiguous blocks of data not covered by the Cumulative Acknowledgement field. However, RFC 2018 (Mat96) does not specify the use of the SACK option when duplicate segments are received. We use the term D-SACK (for duplicate-SACK) to refer to a SACK block that reports a duplicate segment. This extension of SACK is specified in RFC2883 (Flo00). The D-SACK algorithm is not part of the simulation made in section [4.5], because there is no NS2 code available.

3.2.3 F-RTO - Forward RTO-Recovery

The F-RTO algorithm is specified in RFC4138. (PS05)

- **Advantages**: needs no extra options in the TCP header
- **Disadvantages**: Not reliable as EIFEL. Discovers only spurious timeouts but not spurious loss events.

TCP uses the fast retransmit [2.3.4] mechanism to trigger retransmissions after receiving three duplicate acknowledgements (DUPACKS). If for a certain time period TCP sender does not receive ACKs that acknowledge new data, the TCP retransmission timer expires as a backoff mechanism. When the retransmission timer expires, the TCP sender retransmits the first unacknowledged segment assuming it was lost in the network. Because a retransmission timeout (RTO) can be an indication of congestion in the network, the TCP sender resets its congestion window to one segment (default, there are more possible solutions [3.2.1]) and starts increasing it according to the slow start algorithm. [2.3.1]

Therefore the TCP RTO may be triggered spuriously having a bad effect on TCP performance, since it often results in unnecessary retransmissions of many segments. Forward RTO-Recovery is a algorithm for a TCP sender to recover after a retransmission timeout and to recover efficiently from a spurious RTO. This algorithm requires modification only at the sender, and does not require use of any TCP options or additional bits in the TCP header. The F-RTO algorithm uses a set of simple rules for avoiding unnecessary retransmissions after a spurious RTO. When the first acknowledgement arrive after retransmitting the segment for which the RTO expired, the F-RTO sender does not immediately continue with retransmissions like the regular RTO recovery does, but it first checks if the acknowledgements advance the windows to determine whether it needs to retransmit, or whether it can continue sending new data.

When the timeout is not spurious, the F-RTO algorithm reverts back to the conventional RTO recovery algorithm, and therefore has similar behavior and performance. In contrast to alternative algorithms proposed for detecting unnecessary retransmissions, F-RTO does not require any TCP options for its operation, and it can be implemented by modifying only the TCP sender.

The Algorithm

A timeout is considered spurious if it would have been avoided had the sender waited longer for an acknowledgment to arrive (Lud03). F-RTO affects the TCP sender behavior only after a retransmission timeout, otherwise the TCP behavior remains the same. When the RTO expires, the F-RTO algorithm monitors incoming acknowledgments and if the TCP sender gets an acknowledgment for a segment that was not retransmitted due to timeout, the F-RTO algorithm declares a timeout spurious.

The basic F-RTO algorithm

This algorithm is taken from RFC4138. (PS05)

1. When RTO expires, retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE. Also, store the highest sequence number transmitted so far in variable *recover*.
2. When the first acknowledgment after the RTO retransmission arrives at the sender, the sender chooses one of the following actions, depending on whether the ACK advances the window or whether it is a duplicate ACK.
    - If the acknowledgment is a duplicate ACK OR it acknowledges a sequence number equal to the value of ”recover” OR it does not acknowledge all of the data that was retransmitted in step 1, revert to the conventional RTO recovery and continue by retransmitting

unacknowledged data in slow start. Do not enter step 3 of this algorithm. The Spurious-Recovery variable remains as FALSE.

- Else, if the acknowledgment advances the window AND it is below the value of ”recover”, transmit up to two new (previously unsent) segments and enter step 3 of this algorithm. If the TCP sender does not have enough unsent data, it can send only one segment. In addition, the TCP sender MAY override the Nagle algorithm (Nag84) and immediately send a segment if needed. Note that sending two segments in this step is allowed by TCP congestion control requirements (MA99): An F-RTO TCP sender simply chooses different segments to transmit.

  If the TCP sender does not have any new data to send, or the advertised window prohibits new transmissions, the recommended action is to skip step 3 of this algorithm and continue with slow start retransmissions, following the conventional RTO recovery algorithm.

3. When the second acknowledgment after the RTO retransmission arrives at the sender, the TCP sender either declares the timeout spurious, or starts retransmitting the unacknowledged segments.

   - If the acknowledgment is a duplicate ACK, set the congestion window to no more than 3 * MSS, and continue with the slow start algorithm retransmitting unacknowledged segments. The congestion window can be set to 3 * MSS, because two round-trip times have elapsed since the RTO, and a conventional TCP sender would have increased cwnd to 3 during the same time. Leave SpuriousRecovery set to FALSE.
   - If the acknowledgment advances the window (i.e., if it acknowledges data that was not retransmitted after the timeout), declare the timeout spurious, set SpuriousRecovery to SPUR-TO, and set the value of the ”recover” variable to SND.UNA (the oldest unacknowledged sequence number).

SACK-Enhanced Version of the F-RTO Algorithm

This algorithm is taken from RFC4138. (PS05)
By using the SACK option, the TCP sender detects spurious timeouts in most of the cases when packet reordering or packet duplication is present. If the SACK blocks acknowledge new data that was not transmitted after the RTO retransmission, the sender may declare the timeout spurious, even when duplicate ACKs follow the RTO.

1. When the RTO expires, retransmit the first unacknowledged segment and set SpuriousRecovery to FALSE. Set variable *recover* to indicate the highest segment transmitted so far. Following the recommendation in SACK specification [MMFR96], reset the SACK scoreboard.

2. Wait until the acknowledgment of the data retransmitted due to the timeout arrives at the sender. If duplicate ACKs arrive before the cumulative acknowledgment for retransmitted data, adjust the scoreboard according to the incoming SACK information. Stay in step 2 and wait for the next new acknowledgment. If RTO expires again, go to step 1 of the algorithm.

   - if a cumulative ACK acknowledges a sequence number equal to recover, revert to the conventional RTO recovery and set the congestion window to no more than 2 * MSS, like a regular TCP would do. Do not enter step 3 of this algorithm.

- else, if a cumulative ACK acknowledges a sequence number (smaller than recover, but larger than SND.UNA) transmit up to two new (previously unsent) segments and proceed to step 3. If the TCP sender is not able to transmit any previously unsent data – either due to receiver window limitation, or because it does not have any new data to send – the recommended action is to refrain from entering step 3 of this algorithm. Rather, continue with slow start retransmissions following the conventional RTO recovery algorithm.

3. The next acknowledgment arrives at the sender. Either a duplicate ACK or a new cumulative ACK (advancing the window) applies in this step.

   - if the ACK acknowledges a sequence number above *recover*, either in SACK blocks or as a cumulative ACK, set the congestion window to no more than 3 * MSS and proceed with the conventional RTO recovery, retransmitting unacknowledged segments. Take this branch also when the acknowledgment is a duplicate ACK and it does not acknowledge any new, previously unacknowledged data below recover in the SACK blocks. Leave SpuriousRecovery set to FALSE.
   - if the ACK does not acknowledge sequence numbers above recover AND it acknowledges data that was not acknowledged earlier (either with cumulative acknowledgment or using SACK blocks), declare the timeout spurious and set SpuriousRecovery to SPUR-TO. The retransmission timeout can be declared spurious, because the segment acknowledged with this ACK was transmitted before the timeout.

Figure [3.5] shows a response to a spurious timeout using the SACK-enhanced version of F-RTO.

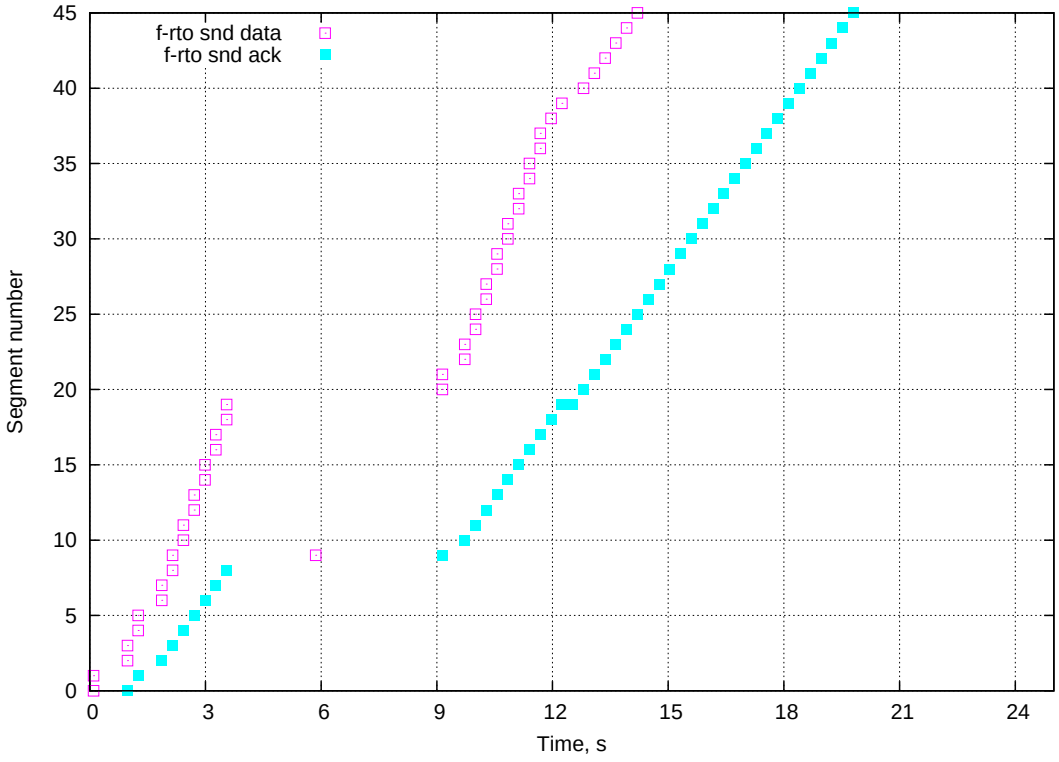

Figure 3.5: F-RTO response to a spurious timeout (SACK-enhanced)

Chapter 4

ECN-based Spurious Loss Event Detection

4.1 Abstract

There are several known methods to detect spurious timeouts and spurious fast retransmits [3.1.2]. (Eifel [3.2.1], D-SACK [3.2.2], F-RTO [3.2.3])
Here is a new one. At first lets take a closer look to the root problem and why spurious timeouts occur. Delays (the measurement of time for a segment to reach its receiver) on Internet paths, especially on wireless links, can be highly variable. On the other hand, modern TCPs have implemented a **fine grained retransmission timer** with a lower minimum timeout value than 1s.
Such a retransmission timer value is recommended in RFC2988. (MA00) This can lead to expiry the RTO timer [2.2.3].

A retransmission timer is a prediction of the upper limit of the RTT (Round Trip Time [2.2.2]) . A spurious timeout occurs when the **RTT suddenly increases**. The RTT can quickly return to normal, but can also stay high when available bandwidth of the bottleneck link has suddenly decreased.
On a spurious timeout, TCP assumes that all outstanding (not acknowledged) segments are lost and retransmits them unnecessarily.

The go-back-N [2.3.5] retransmission behavior triggered by spurious timeouts has a root: the retransmission ambiguity. This happens when a sender is unable to distinguish an ACK for the original transmission of a segment from the ACK for its retransmission.

Shortly after the timeout ACKs for the original transmission return to the TCP sender. On receipt of the first ACK after the timeout, the TCP sender must interpret this ACK as acknowledging the retransmission, and must assume that all other outstanding segments have also been lost.
Thus, the sender enters the slow start phase [2.3.1] , and retransmits all outstanding segments in this fashion.

The go-back-N [2.3.5] retransmission triggers the next problem: the receiver generates a DUPACK (duplicate ACK) for every segment received more than onces. The receiver has to do this, because it must assume that its original ACKs had been lost. This triggers a spurious fast retransmit [3.1.3] at the sender.

Another problem is that after a spurious timeout or a spurious fast retransmit the **congestion window and slow start threshold are reduced**. This **reduction is unnecessary** as no data loss has yet been detected that would indicate congestion in the network.

To avoid all these problems, caused by a spurious timeout or a spurious fast retransmit, the TCP Sender needs more information to **eliminate the retransmission ambiguity**. The TCP Sender needs the possibility to unambiguously distinguish an ACK for the original transmission of a segment from that of a retransmission. There still are Algorithms like (Eifel [3.2.1], D-SACK [3.2.2], F-RTO [3.2.3]), but they all have big disadvantages.

4.2 The new Algorithm - ECNSP

Eliminating the retransmission ambiguity is the main goal of this algorithm. Then, if there is no ambiguity, all problems described above are lapsed. The algorithm uses ECN, especially ECN with a one bit nonce. [2.6.4] Therefore an ECN capable sender, receiver and router is needed.

To encode the one bit nonce the following ECN codepoints are used:

NONCE	RFC2481	PROPOSED
00	ECN-incapable	ECN-incapable
10	No congestion	No congestion, Nonce=0 – ECT((0))
01	undefined	No congestion, Nonce=1 – ECT(1)
11	Congestion	Congestion, Nonces cleared

4.2.1 Normal Sender and Receiver behavior

This case is described in Figure [4.1]. The sender attaches to each packet a random nonce [0 or 1] and transmits this segment. (A=1,B=1,C=0) The one bit nonce is coded as an ECN codepoint. For example the nonce 0 is the ECN codepoint ECT(0) and nonce 1 is the ECN codepoint ECT(1). The receiver receives the segment and sends an ACK with the received nonce back (written in the NS Flag) to the sender.

This behavior differs a little bit from the original specification of ECN with one bit nonce [2.6.4]. Instead of calculating the NS sum, the receiver only acknowledges the received nonce.

4.2.2 Retransmission - Sender and Receiver behavior

This case is described in Figure [4.2]. It takes place if a timeout, because of a sudden delay spike or bandwidth change, happened. Retransmitted segments have no nonce. Instead of setting ECT codepoints ECT(0) or ECT(1), the ECT and CE is set to 0 (Not ECN or retransmission of segments). That means, that no nonce is decoded. For the first retransmitted segment the TCP Flag CWR is also set. The receiver notices that ECT and CE is set to 0 and CWR is set to 1. That means that the receiver does not react as usual by acknowledging the received nonce. He fills up the NS Flag with 0. (V=0,W=0,X=0) That means that the receiver responses a segment with no nonce always with NS=0.

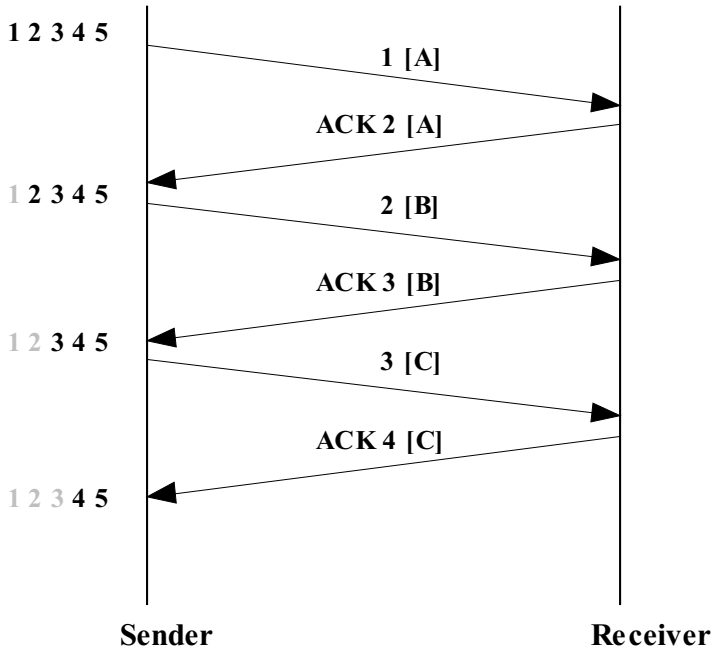

Figure 4.1: Normal sender and receiver behavior

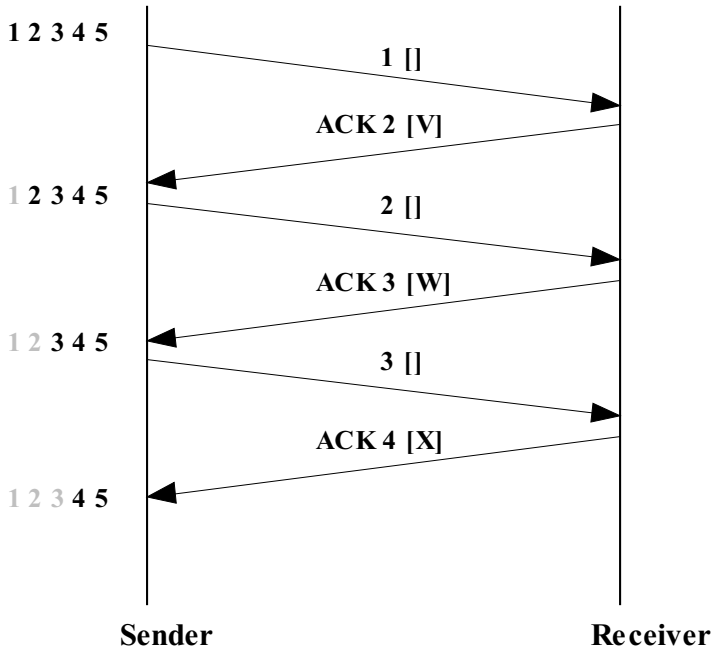

Figure 4.2: Sender and receiver retransmission behavior

4.3 Detection of a spurious timeout

Assumption: All involved parties (Sender, Receiver, Router) are ECN capable, and they support ECN with one bit nonce [2.6.4]. Figure [4.3] shows how a spurious timeout detection with the new Algorithm works. As mentioned above the main goal of this algorithm is **eliminating the retransmission ambiguity**. All we have is a nonce (0 or 1) coded as ECN Codepoint. So, if an ACK arrives after a timeout we have to check if it is an ACK for the original transmission or an ACK for the retransmission. When the sender checks only the nonce of the first ACK after the timeout the chance is 50% to determine if it was a spurious timeout or not. For this reason we need a parameter which determines how many ACKs after a timeout has to be checked by the sender. This parameter is named **ecnsp_param**. The following table illustrates the effect of using such a parameter.

ecnsp_param	chance	Remark
0	50%	nonce of the first ACK after the timeout is checked
1	75%	nonce of the first and second ACK after the timeout is checked
2	87,5%	nonce of the first, second and third ACK after the timeout is checked
3	93,7%	check nonce values of 4 ACKs in series after a timeout
4	96,8%	check nonce values of 5 ACKs in series after a timeout
5	98,4%	check nonce values of 6 ACKs in series after a timeout

We can see if **ecnsp_param=0** only the nonce of the first ACK is checked. The chance to detect the spurious timeout is 50%. This variant of the new algorithm is the fastest but the one with the lowest chance to detect a spurious timeout. As described in table above the probability can be increased. When we set **ecnsp_param to 1**, the nonce values of the first and second ACK after a timeout will be checked. This variant needs more time to detect a spurious timeout (the sender has to wait till 2 ACKs after the timeout arrive), but the chance to detect the potential spurious timeout is now 75%. A other variant is thinkable, we set **ecnsp_parameter to 2**. Now the sender has to check the nonce values of the first, sencond and third ACK after a timeout. This increases the chance to 87,5%, but takes more time, because the sender has to wait until 3 ACKs after a timeout arrive. In the section [4.5] we will use the fastest variant, namely we will use the ecnsp_param of 0, because it is the fastest variant. To use a **ecnsp_param** of **3 or 4** is also thinkable. The chance to detect a spurious timeout would be 93,7% or 96,8%. We will see a comparison of using different **ecnsp_param** parameters in section [4.5].

It is clear that the nonce value(s) could be guessed by a receiver with bad faith. The following table shows the chance to guess the correct nonce value(s).

ecnsp_param	chance
0	50%
1	25%
2	12,5%
3	6.25%
4	3.20%
5	1.60%

If we use **ecnsp_param=0** the chance to guess the correct nonce is 50%, which might be a little bit too high. When we take **ecnsp_param=1** the chance to guess the correct nonces (there are 2 ACKS

to check) is now 25%. This is much better. The other variants, **ecnsp_param=2 or ecnsp_param=3** can make the chance to guess the correct nonce values lower and are probably the **best choices**, because this variants are very secure and also very fast (figure [4.16] shows a comparison of different ecnsp_param configurations). When we use **ecnsp_param 4 or 5** the advantages of using ECNSP would be decreased [4.5.8][4.16].

Figure [4.3] shows a spurious timeout detection with the new algorithm using an **ecnsp_param of 0**. That means that the sender has to check and save the nonce value of one segment after a timeout. The chance to detect a spurious timeout with this configuration is 50%.

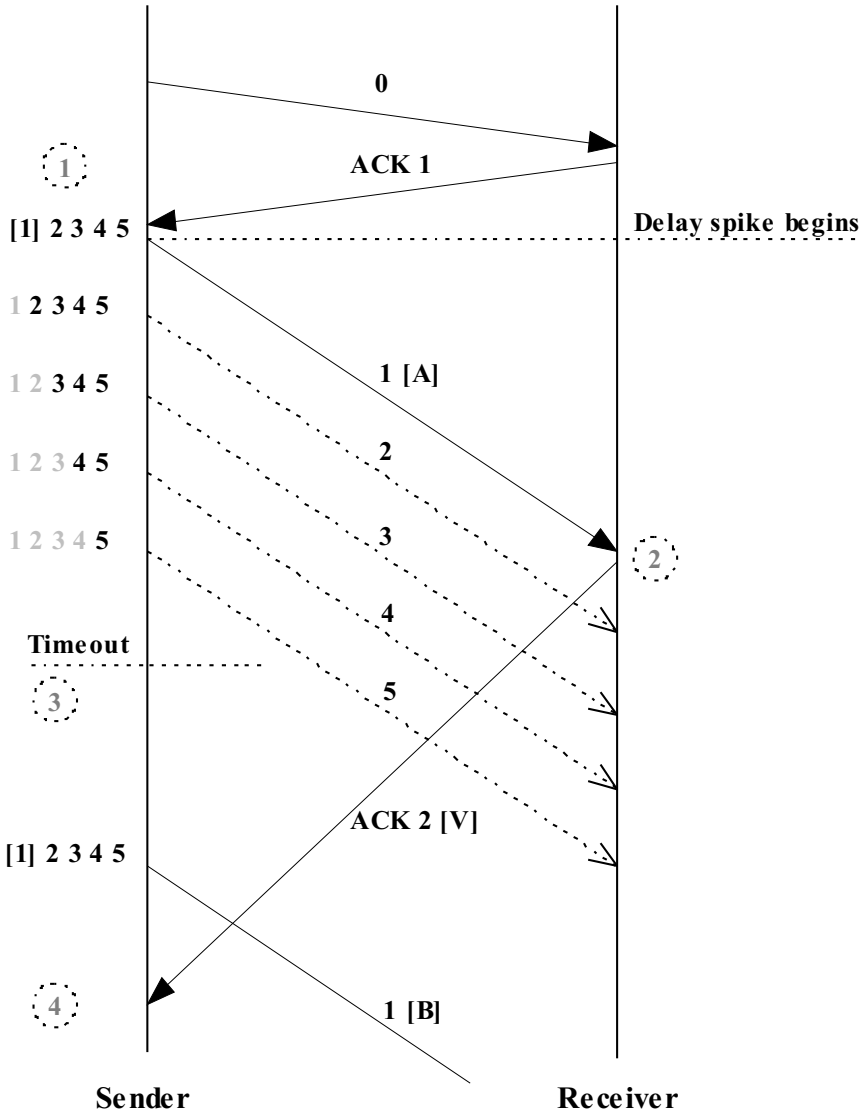

Figure 4.3: Spurious timeout detection with ecnsp_param=0

1. The ECN - capable Sender transmits segment 1..5 and attaches to each segment a random nonce. As shown in the figure [4.3] in this case only the first segment is important, because the ecnsp_param is 0. The random nonce is decoded as an ECN codepoint. (assume A == 1 = ECT(1))

When sending segment 1 a delay spike begins. The ECN capable router detects that the ECT codepoint is set and forwards each segments.

2. The ECN - capable Receiver receives segment 1..5. Instead of calculating the Nonce Sum as specified in [2.6.4] he only sends the received nonce back to the sender. (V == A)

3. Because of the delay spike the retransmission timer expires and a timeout happens. The TCP sender retransmits the first unacknowledged segments assuming it was lost in the network. Now a special convention for all retransmitted segments takes place: ECT=0 and CE=0 for all segments, and CWR=1 only for the first retransmitted segment. That means that for all retransmitted segments no nonce is decoded. The receiver reacts on segments with no nonce, by default, with sending back ACKs with nonce 0.

4. Now an ACK arrives at the sender. The sender knows that ecnsp_param = 0, so he has to check this nonce only, further on he knows, if this ACK is the acknowledgement for the retransmitted segment then it has a nonce value of 0. But that is not the case (V == 1). So the sender can determine that the received ACK is the acknowledgement for the original segments 1. Therefore the **timeout was spurious**. If the nonce value of A would have been 0, the spurious timeout would not have been detected. With the ecnsp_param of 0 the chance to detect a spurious timeout is 50%. This chance might be very low, but in comparison with other algorithms it is the fastest one. We will see this in section [4.5]

We have seen that the use of **ecnsp_param of 0** is very fast, but the chance to detect a spurious timeout is 50%. Therefore we try it with an **ecnsp_param of 1**. Here the sender has to check the nonce values of the first and second ACK after a timeout. The chance to detect a spurious timeout in is case is 75%.

1. The ECN - capable Sender transmits segment 1..5 and attaches to each segment a random nonce. As shown in the figure [4.4] in this case only the first and second segments are important, because the ecnsp_param is 1. The random nonce is decoded as an ECN codepoint. (assume A == 1 and B == 0) When sending segment 1 a delay spike begins. The ECN capable router detects that the ECT codepoint is set and forwards each segments.

2. The ECN - capable Receiver receives segment 1..5. Instead of calculating the Nonce Sum as specified in [2.6.4] he only sends the received nonce back to the sender. (V == A and W == B)

3. Because of the delay spike the retransmission timer expires and a timeout happens. The TCP sender retransmits the first unacknowledged segments assuming it was lost in the network. Now a special convention for all retransmitted segments takes place: ECT=0 and CE=0 for all segments, and CWR=1 only for the first retransmitted segment. That means that for all retransmitted segments no nonce is decoded. The receiver reacts on segments with no nonce, by default, with sending back ACKs with nonce 0.

4. Now an ACK arrives at the sender. The sender knows: ecnsp_param = 1, so he has to check this nonce and has to wait for the next ACK arriving at the sender. He also knows if this ACKs are the acknowledgements for the retransmitted segments than they have a nonce value of 0. But that is not the case (V == 1 and W == 0). So, after the second ACK has arrived at the sender it can determine that the received ACKs are the acknowledgement for the original segments 1 and 2. Therefore the **timeout was spurious**. If the nonce values of A and B were 0, the spurious timeout would not have been detected.

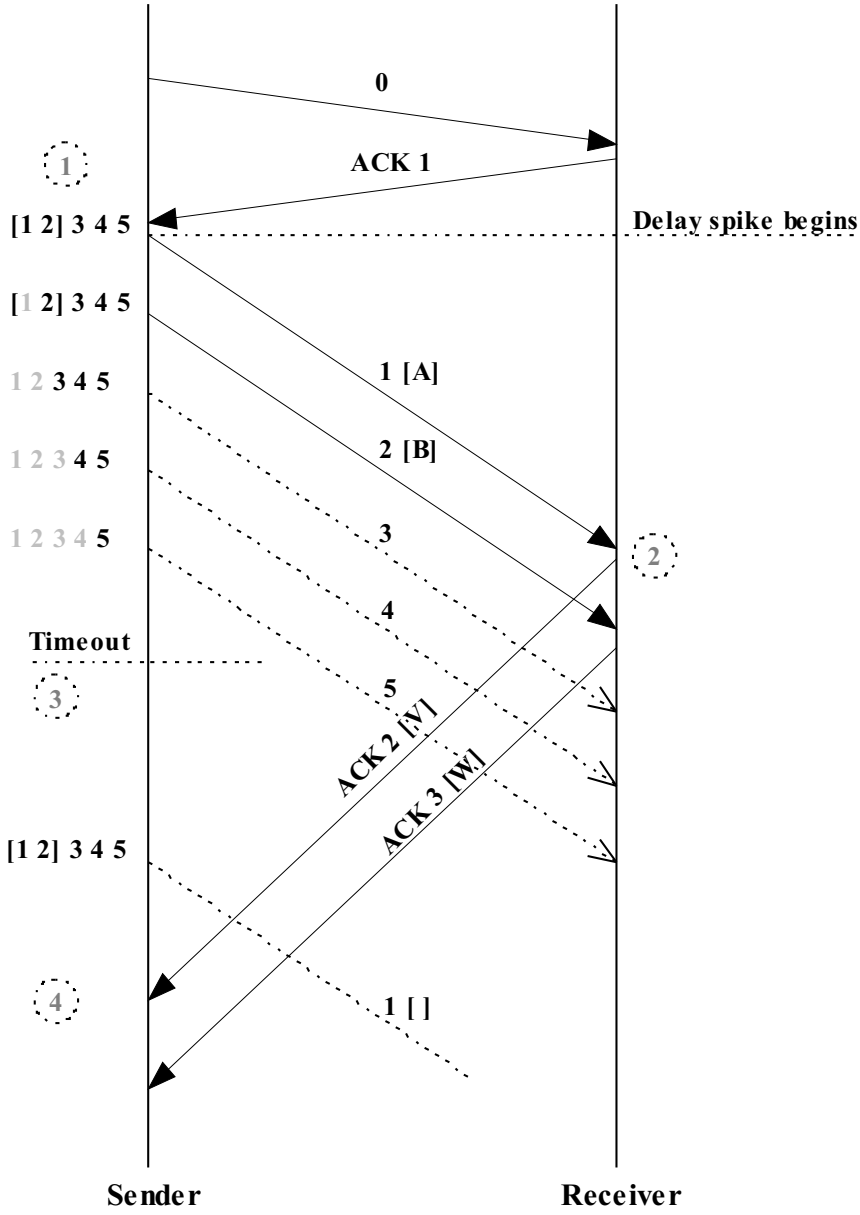

Figure 4.4: Spurious timeout detection with ecnsp_param=1

At least we try it with an **ecnsp_param of 2**. Here the sender has to check the nonce values of the first, second and third ACK, after a timeout. The chance to detect a spurious timeout in this case is 87,5%.

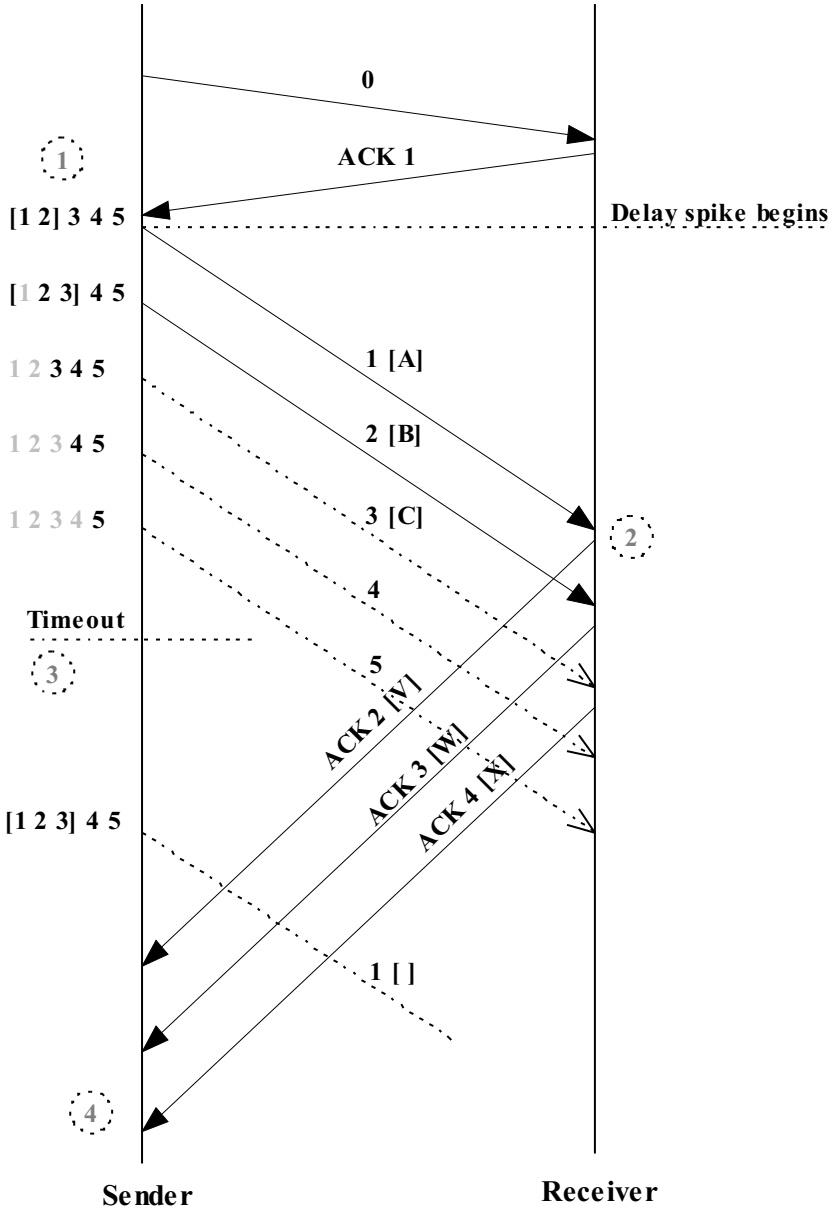

Figure 4.5: Spurious timeout detection with ecnsp_param=2

1. The ECN - capable Sender transmits segment 1..5 and attaches a random nonce to each segment. As shown in the figure [4.5] in this case only the first, second and third segment are important, because the ecnsp_param is 1. The random nonce is decoded as an ECN codepoint. (assume A == 1,B == 0 and C == 1) When sending segment 1 a delay spike begins. The ECN capable router detects that the ECT codepoint is set and forwards each segments.

2. The ECN - capable Receiver receives segment 1..5. Instead of calculating the Nonce Sum as

specified in [2.6.4] he only sends the received nonce back to the sender. (V == A,W == B and X == C)

3. Because of the delay spike the retransmission timer expires and a timeout happens. The TCP sender retransmits the first unacknowledged segments assuming that it was lost in the network. Now a special convention for all retransmitted segments takes place: ECT=0 and CE=0 for all segments, and CWR=1 only for the first retransmitted segment. That means that for all retransmitted segments no nonce is decoded. The receiver reacts on segments with no nonce, by default, with sending back ACKs with nonce 0.

4. Now an ACK arrives at the sender. The sender knows: ecnsp_param = 2, so it has to check this nonce and has to wait for the next 2 ACKs arriving at the sender. It also knows if this ACKs are the acknowledgements for the retransmitted packets than they have a nonce value of 0. But that is not the case (V == 1,W == 0 and X == 1). So, after the third ACK has arrived at the sender, it can determine that the received ACKs are the acknowledgement for the original segments 1, 2 and 3. Therefore the **timeout was spurious**. If the nonce values of A, B and C were 0, the spurious timeout would not have been detected.

As we see, this variant is the slowest but also the one with the highest chance to detect a spurious timeout.

The TCP sender often transmits several segments unnecessarily, because after the RTO expires and after the first retransmission the cumulative acknowledgements for the original transmissions appear at the TCP sender at a time. To avoid the unnecessary sending of segments, we will use a ecnsp_param of 0. A further reason to do this is that there is a possibility that there arrive ACKs for the retransmission while waiting for the original ACKs. (ecnsp_param 1 or ecnsp_param 2) This case will be described in the next Figure [4.6]. It is possible that the sender retransmits the whole window again, before the original ACKs arrive at the sender. Then the default TCP error recovery schema would be faster.

As you can see in this Figure [4.6] the original ACKs arrive later at the sender, than the retransmitted ACKs. In such a case the advantages of the new algorithm would not be very distinctive. The sender has to wait a long time before it can determine if the timeout was spurious or not.

4.4 Detection of a spurious fast retransmit

The other TCP error strategy than timeout based retransmission is **DUPACK based retransmission**. The idea is to use duplicate ACKs as an indication of packet loss. EX: if a sender transmits segments 1,2,3,4,5 and only 1,3,4,5 make it to the other end, the receiver will typically respond to segment 1 with an ACK 2 (i expect segment 2 now) and send three more such ACKs (duplicate ACKs) in response to segments 3,4 and 5.

TCP may generate an immediate acknowledgment (a duplicate ACK) when an out- of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it what sequence number is expected. Since TCP does not know whether a duplicate ACK is caused by a lost segment or just a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before

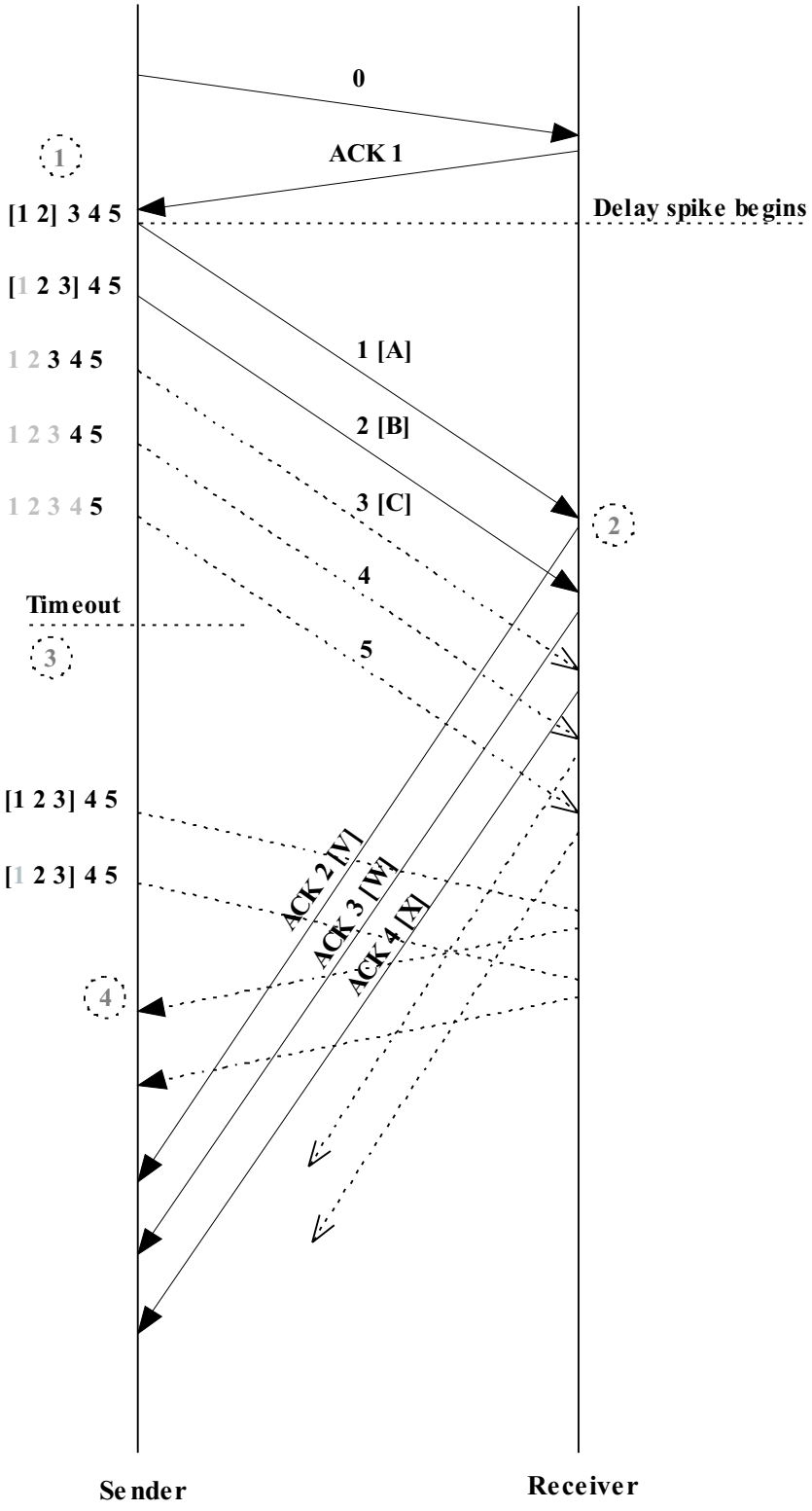

Figure 4.6: No spurious timeout detection with ecnsp_param=2

the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire. After a fast retransmit congestion avoidance, but not slow start is performed. This behavior is called fast recovery [2.3.4]. It is an improvement that allows high throughput under moderate congestion, especially for large windows. As we know after a fast retransmit congestion avoidance [2.3.2] is performed. When a fast retransmit occurred spurious, the cwnd window is reduced to half and will be increased linearly. To avoid this effect the new algorithm [4] with an ecnsp_param of 0 can be used to detect spurious fast retransmits. Figure [4.7] shows this scenario:

- The ECN - capable Sender transmits segment 1..5 and attaches a random nonce to each segment. The random nonce is decoded as an ECN codepoint.
- The receiver acknowledges the incoming segments and sends back the received nonces. Now it happens: Segment with the sequence number 4 does not reach the receiver.
- Later in the connection segment 5 arrives at the receiver. Segment 4 has not arrived yet, so the receiver generates an ACK 4, which means that the receiver is already waiting for segment 4. Segments 6 and 7 arrive at the receiver. The receiver has now generated 3 DUPACKs for segment 4.
- After this 3rd DUPACK the error recovery strategy called *fast retransmit* takes place. The sender sends segment with the sequence number 4 again. Shortly after sending this segment again an ACK arrives at the sender. Now there is the big question: is this ACK an ACK for the retransmitted segment 4 or an ACK for the original, spurious lost segment 4? The sender knows that all retransmitted segments have the nonce value of 0. But the received segment has the nonce value of 1, so the sender knows that the fast retransmit was spurious. The sender can now restore **cwnd** and **ssthresh** to the values before the spurious fast retransmit. (no complete fast recovery is performed) If the nonce value of the original segment was 0, the spurious fast retransmit would not have been detected.

  To detect such a spurious fast retransmit only an ecnsp_param of 0 can be considered. That means, the chance to detect is 50%

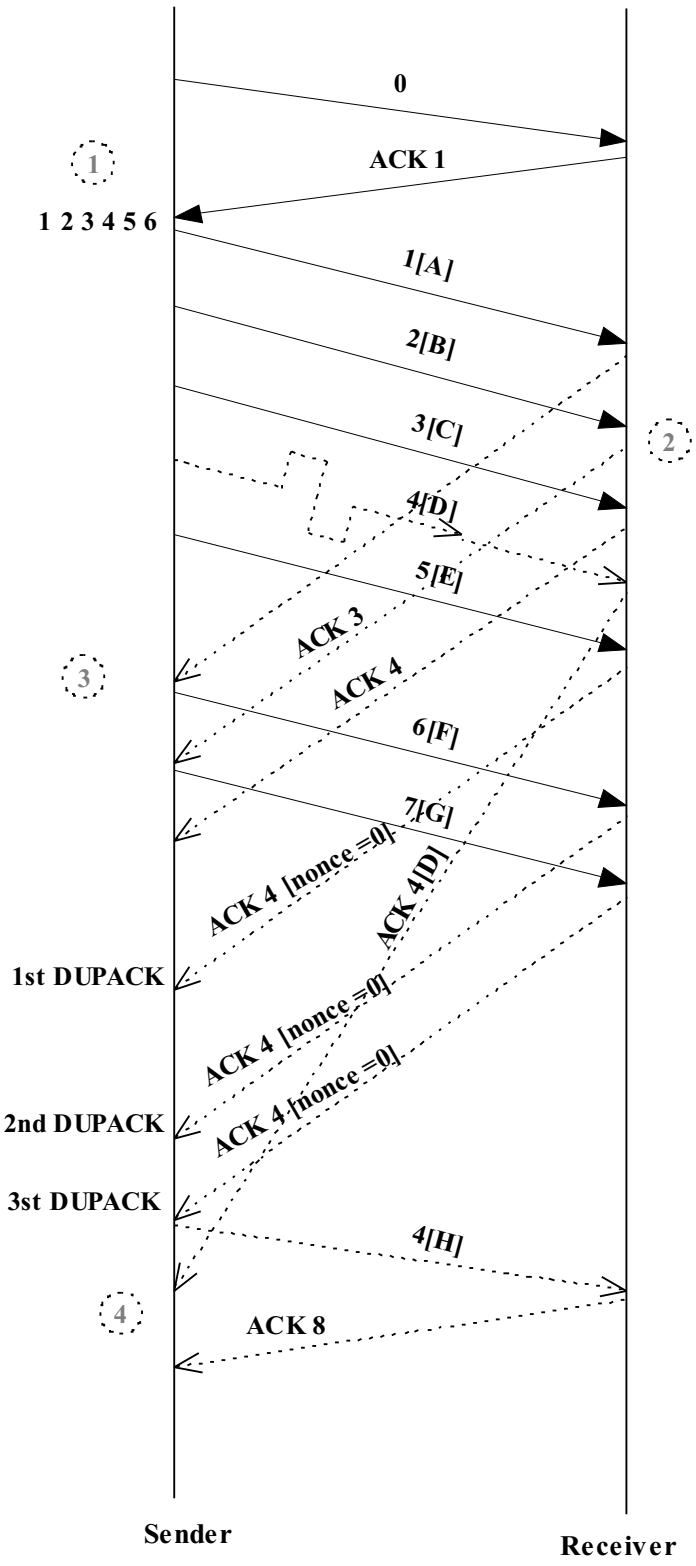

Figure 4.7: Spurious fast retransmit detection with ecnsp_param=0

4.5 Simulation results

All simulation results in this section are made with the network simulator NS2 - release 2.29 [(NS206)]. Andrei Gurtov's [(Gur02)] implementation of the Eifel algorithm [3.2.1] was the starting point for the simulation work. For simulating spurious timeouts it was important to have a tool for generating delay spikes over a simulated link. This tool was written by Andrei Gurtov [(Gur02)] and suspends the transmission in both directions simultaneously. The length of delay spikes is uniformly distributed between 3 and 15 seconds. Theses delay spikes occur at the interval of 20-40 seconds. So it was possible to generate spurious timeouts.

4.5.1 Spurious timeouts in TCP

This section describes the problem of spurious timeouts [3.1.2] in more detail. Figure [4.8] shows a spurious timeout simulated in NS2. On a spurious timeout TCP assumes that all outstanding (not acknowledged) segments are lost and retransmits them all unnecessarily. If a spurious timeout occurs TCP reacts with a so called go-back-N behavior. The root of this problem is the retransmission ambiguity, which means that the TCP sender is not able to distinguish an ACK for the original transmission of a segment from the ACK for its retransmission.

The listing below, a trace of ns2, shows how it comes about:

- No feature to detect spurious timeouts in TCP is enabled
- Till second 3 everything is alright, the sender sends segments the receiver receives them and generates ACKs which are then received by the sender.
- At second 3 in the connection the delay spike begins.
- At 5.92 seconds the timeout occurs. At this point 14 segments are sent and 6 segments are acknowledged. There are still 8 unacknowledged segments in the network.
- At the same time the sender starts retransmitting the first unacknowledged segment (in this case segment with the sequence number 7), because the sender has to assume that all outstanding segments are lost.
- At 9.14 seconds in the connection an ACK arrives at the sender. The big question is now: is this ACK an ACK for the original transmission of segment 7 or an ACK for the retransmission? Without spurious timeout detection features like the new algorithm [4] TCP is not able to answer this question.
- From 9.14 seconds till 10.28 seconds, the sender retransmits all outstanding segments again because it assumes that all outstanding segments are lost and thus enters the slow start phase [2.3.1]. At second 10.28 the sender has retransmitted the 8 outstanding segments again.
- At 9.72 seconds the sender gets the next ACK, which is the ACK for segment 8. This process continues till seconds 11.4, when an ACK for segment 14 arrives at the sender.
- There is another big problem: the receiver generates a DUPACK for every segment received more than once.

- At 11.37 seconds the receiver generates the first DUPACK, and we know that 8 segments will arrive at all. At 13.33 seconds the last retransmitted segment has arrived at the sender and 8 DUPACKS are generated.
- This may trigger a spurious fast retransmit [3.1.3].
- At 13.92 seconds all problems are resolved and the sender can continue sending segments in normal manner.

It is clear that there are many problems when a spurious timeout occurs, and TCP is not able to handle this. Therefore the new algorithm [4] is created.

Listing 4.1: Default TCP behavior without any spurious timeout detection feature

```
0.100000 Sender:: send segment with seqno: 0
0.410667 Receiver:: received segment with seqno: 0

0.721333 Sender received ack for seqno 0
0.721333 Sender:: send segment with seqno: 1
0.721333 Sender:: send segment with seqno: 2
1.301333 Receiver:: received segment with seqno: 1
1.581333 Receiver:: received segment with seqno: 2

1.612000 Sender received ack for seqno 1
1.612000 Sender:: send segment with seqno: 3
1.612000 Sender:: send segment with seqno: 4

1.892000 Sender received ack for seqno 2
1.892000 Sender:: send segment with seqno: 5
1.892000 Sender:: send segment with seqno: 6
2.192000 Receiver:: received segment with seqno: 3
2.472000 Receiver:: received segment with seqno: 4

2.502667 Sender received ack for seqno 3
2.502667 Sender:: send segment with seqno: 7
2.502667 Sender:: send segment with seqno: 8
2.752000 Receiver:: received segment with seqno: 5

2.782667 Sender received ack for seqno 4
2.782667 Sender:: send segment with seqno: 9
2.782667 Sender:: send segment with seqno: 10

3: Intercept

3.032000 Receiver:: received segment with seqno: 6

3.062667 Sender received ack for seqno 5
3.062667 Sender:: send segment with seqno: 11
3.062667 Sender:: send segment with seqno: 12
3.312000 Receiver:: received segment with seqno: 7

3.342667 Sender received ack for seqno 6
3.342667 Sender:: send segment with seqno: 13
3.342667 Sender:: send segment with seqno: 14
```

```
5.922667 Timeout occured

5.922667 Slow start performed

5.922667 Sender:: send segment with seqno: 7

9.142908 Sender received ack for seqno 7
9.142908 Sender:: send segment with seqno: 8
9.142908 Sender:: send segment with seqno: 9
9.412241 Receiver:: received segment with seqno: 8
9.692241 Receiver:: received segment with seqno: 9

9.722908 Sender received ack for seqno 8
9.722908 Sender:: send segment with seqno: 10
9.722908 Sender:: send segment with seqno: 11
9.972241 Receiver:: received segment with seqno: 10

10.002908 Sender received ack for seqno 9
10.002908 Sender:: send segment with seqno: 12
10.002908 Sender:: send segment with seqno: 13
10.252241 Receiver:: received segment with seqno: 11

10.282908 Sender received ack for seqno 10
10.282908 Sender:: send segment with seqno: 14
10.532241 Receiver:: received segment with seqno: 12

10.562908 Sender received ack for seqno 11
10.562908 Sender:: send segment with seqno: 15
10.812241 Receiver:: received segment with seqno: 13

10.842908 Sender received ack for seqno 12
10.842908 Sender:: send segment with seqno: 16
11.092241 Receiver:: received segment with seqno: 14

11.122908 Sender received ack for seqno 13
11.122908 Sender:: send segment with seqno: 17


11.372241 Receiver:: received duplicate packet 7


11.402908 Sender received ack for seqno 14
11.402908 Sender:: send segment with seqno: 18
11.402908 Sender:: send segment with seqno: 19


11.652241 Receiver:: received duplicate packet 8
11.932241 Receiver:: received duplicate packet 9
12.212241 Receiver:: received duplicate packet 10
12.492241 Receiver:: received duplicate packet 11
12.772241 Receiver:: received duplicate packet 12
13.052241 Receiver:: received duplicate packet 13
13.332241 Receiver:: received duplicate packet 14
```

```
13.922908 Sender received ack for seqno 15
13.922908 Sender:: send segment with seqno: 20
14.172241 Receiver:: received segment with seqno: 17

14.202908 Sender received ack for seqno 16
14.202908 Sender:: send segment with seqno: 21
14.452241 Receiver:: received segment with seqno: 18

14.482908 Sender received ack for seqno 17
14.482908 Sender:: send segment with seqno: 22
14.732241 Receiver:: received segment with seqno: 19

14.762908 Sender received ack for seqno 18
14.762908 Sender:: send segment with seqno: 23
15.012241 Receiver:: received segment with seqno: 20

15.042908 Sender received ack for seqno 19
15.042908 Sender:: send segment with seqno: 24
15.042908 Sender:: send segment with seqno: 25
15.292241 Receiver:: received segment with seqno: 21

15.322908 Sender received ack for seqno 20
15.322908 Sender:: send segment with seqno: 26
15.572241 Receiver:: received segment with seqno: 22

15.602908 Sender received ack for seqno 21
15.602908 Sender:: send segment with seqno: 27
15.852241 Receiver:: received segment with seqno: 23

15.882908 Sender received ecnsp_ack for seqno 22
15.882908 Sender:: send segment with seqno: 28
16.132241 Receiver:: received segment with seqno: 24
```

Another major problem is the distortion of TCP's Error Recovery Strategies. As we know, after a timeout based retransmission TCP's congestion control takes place. In this case we have a timeout based retransmission and therefore slow start and congestion avoidance (the combination of both algorithms is implemented) is performed unnecessarily [2.3.3]. Figure [4.9] shows the cwnd of the sender. This should show how the sender reduces its load unnecessarily.

- The senders congestion window grows normally till second 3.
- Here the delay spike begins
- At 5.92 seconds the spurious timeout occurs. The sender is forced to perform slow start, and sets its cwnd to 1 segment.
- Slow start increases cwnd exponentially each time an ACK arrives at the sender, till cwnd reaches half of its value.
- At second 10.0 that value of cwnd has reached. Now the sender enters congestion avoidance.
- Here it increases its load linearly. segsize*segsize/cwnd is the exact value of the linear increase.
- At 18.9 seconds the original value of cwnd has reached.

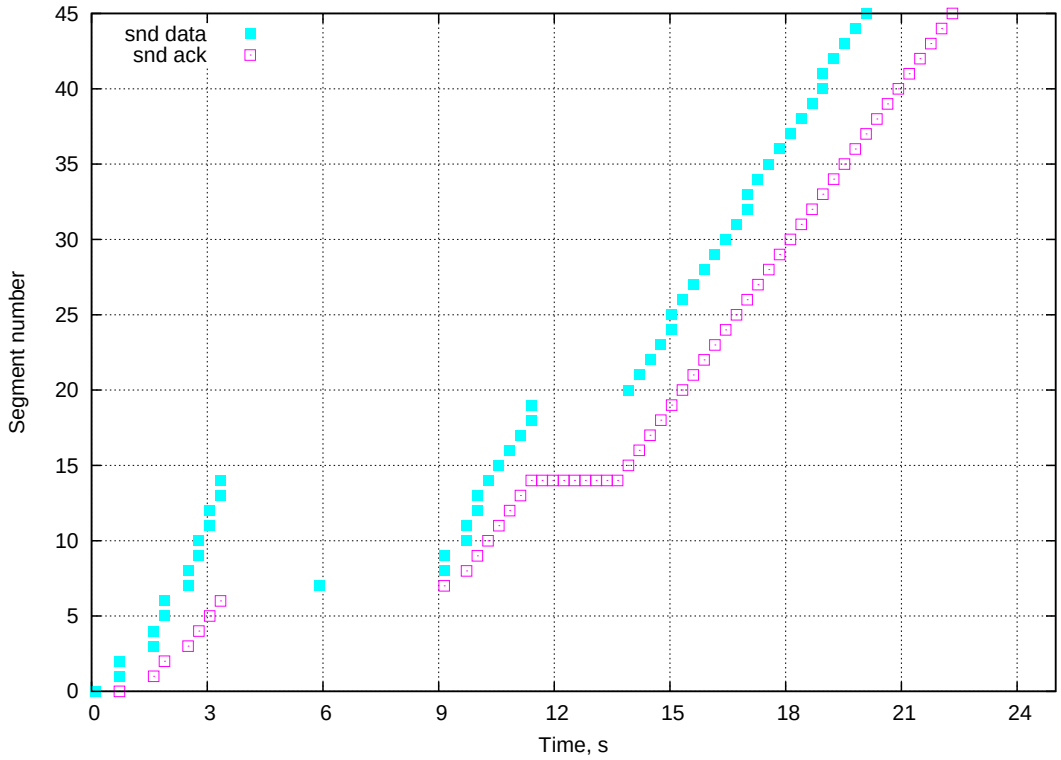

Figure 4.8: Spurious timeout in TCP

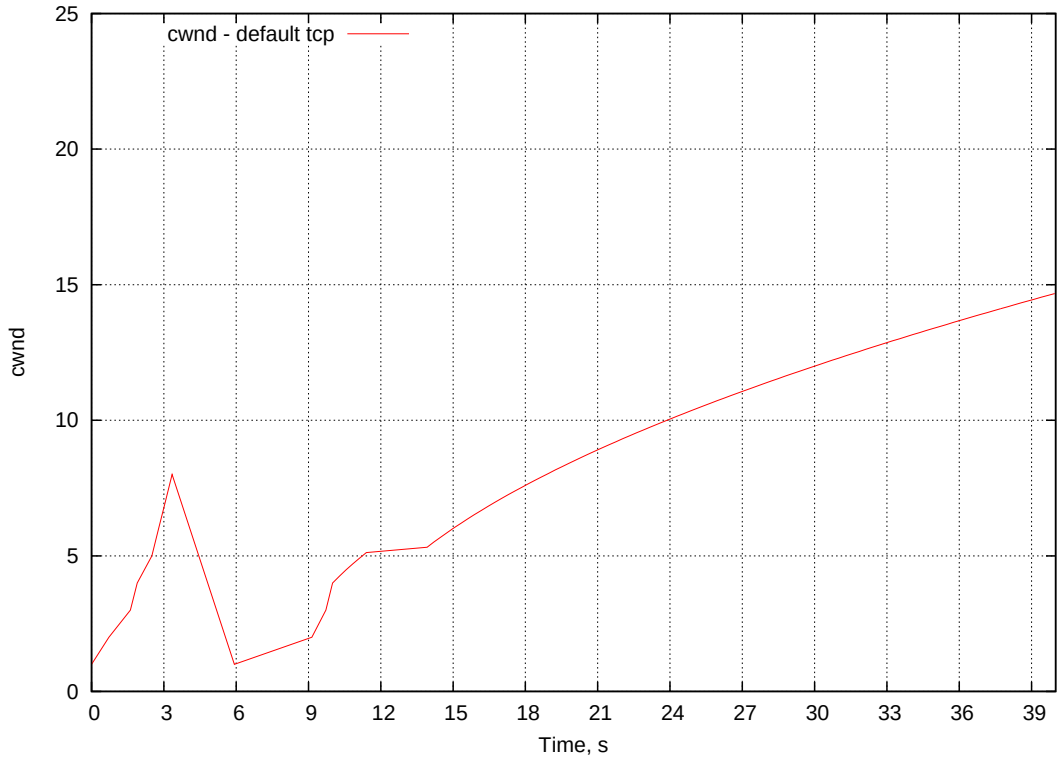

Figure 4.9: cwnd of the sender without any spurious detection feature

It is clear that if there is no need for slow start and congestion avoidance. The TCP sender would not reduce its load unnecessarily. This increases the whole TCP throughput.

4.5.2 Spurious fast retransmits in TCP

As decribed in the previous section the TCP error strategy other than the timeout based retransmission is DUPACK based retransmission. The idea is to use duplicate ACKs as an indication of packet loss. TCP may generate an immediate acknowledgment (a duplicate ACK) when an out- of-order segment is received. This duplicate ACK should not be delayed. The purpose of this duplicate ACK is to let the other end know that a segment was received out of order, and to tell it which sequence number is expected. Since TCP does not know whether a duplicate ACK is caused by a lost segment or just by a reordering of segments, it waits for a small number of duplicate ACKs to be received. It is assumed that if there is just a reordering of the segments, there will be only one or two duplicate ACKs before the reordered segment is processed, which will then generate a new ACK. If three or more duplicate ACKs are received in a row, it is a strong indication that a segment has been lost. TCP then performs a retransmission of what appears to be the missing segment, without waiting for a retransmission timer to expire. After fast retransmit, congestion avoidance, and not slow start is performed. This behavior is called fast recovery [2.3.4]. It is an improvement that allows high throughput under moderate congestion, especially for large windows. As we know after a fast retransmit congestion avoidance [2.3.2] is performed. When a fast retransmit occurred spurious, the cwnd window is reduced to half and is increased linearly.

Figure [4.10] shows a plot of a spurious timeout and a spurious fast retransmit with a default TCP implementation. At second 3 in the connection a delay spike begins. After that a spurious timeout occurs. The scenario of a spurious timeout is described in more detail in section [4.5.1].

Without any spurious timeout detection feature TCP is not able to detect such a spurious timeout and unnecessary reduces its load. Later in the connection, a single segment gets lost. After receiving 3 DUPACKs, a fast retransmit is performed. A default TCP implementation is not able to determine if the fast retransmit is spurious or not. It could be possible that the assumed lost segment is not lost, and the ACK is on the fly. If this happens the performed fast retransmit is spurious and affects the TCP performance, because the sender halves its load by entering the congestion avoidance phase.

Figure [4.11] shows the corresponding congestion window.

4.5.3 Spurious timeout reaction with ECNSP

Eliminating the retransmission ambiguity is the main goal of the new algorithm and requires extra information in ACKs, so that the sender is able to distinguish an ACK for the original transmission of a segment from that of a retransmission. The combination of ECN and using a one bit random nonce decoded as ECN codepoint provides this extra information [4].

The listing below shows how the new algorithm works.
In this simulation an ecnsp_param of 0 is used. It means that the sender has to check the nonce of the first ACK arriving after the timeout only. The chance to detect a spurious timeout is about 50%. This might be very low, but this variant is also the fastest (we do not wait for more ACKs arriving at the sender).

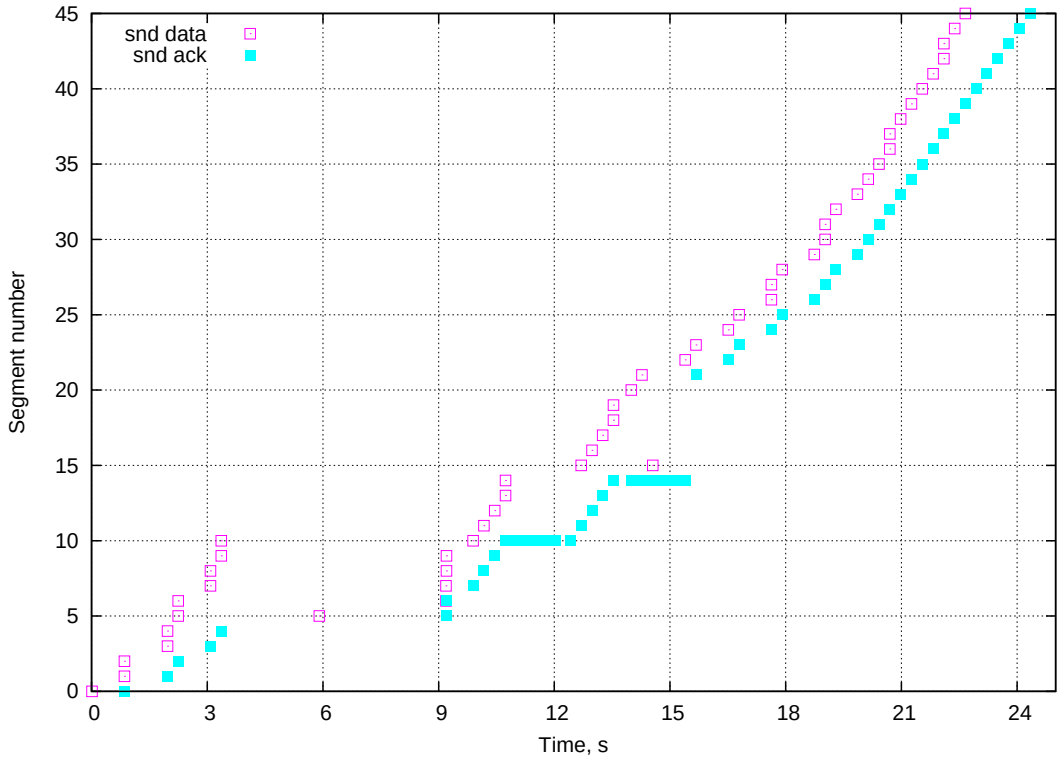

Figure 4.10: Spurious fast retransmit in TCP

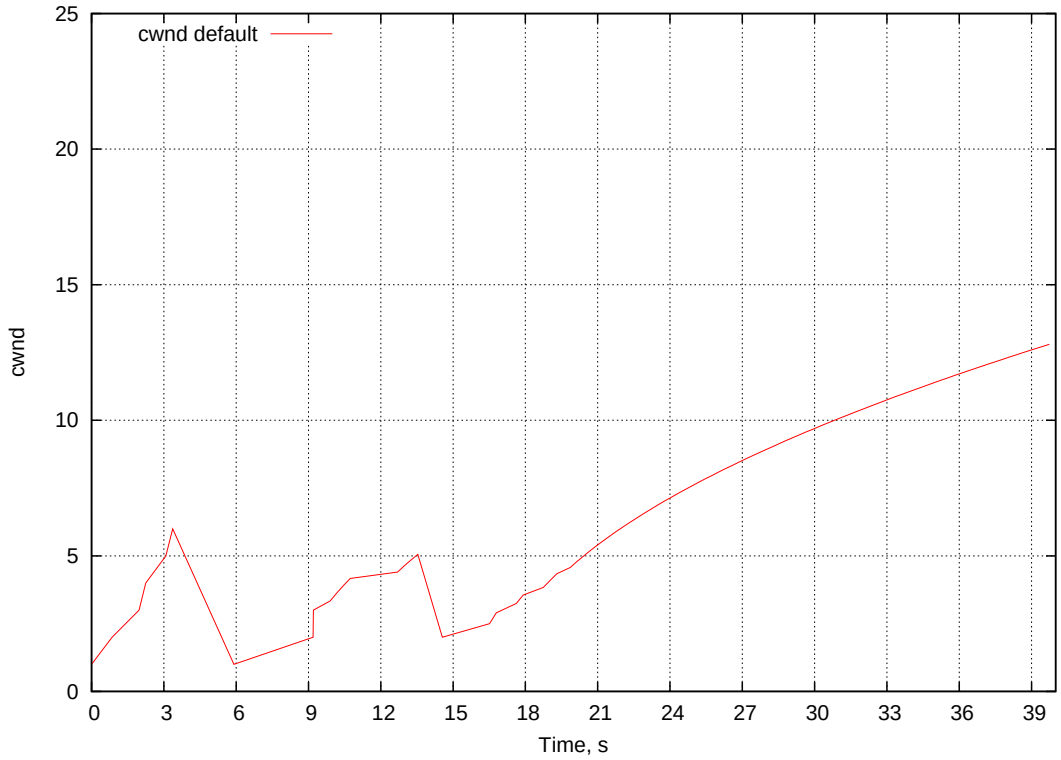

Figure 4.11: cwnd of the sender without any spurious fast retransmit detection feature

- The sender attaches a random nonce (0 or 1) to each segment (decoded as ECN codepoint)
- At second 3 in the connection a delay spike begins
- The sender continuous sending till 3.34 seconds. Meanwhile the ACKs keep on coming.
- At 5.92 seconds the timeout occurs.
- At this time the sender has sent 14 segments and has last received an ACK with the sequence number 6. It means that there are 8 unacknowledged segments in the network.
- At the same time the sender starts retransmitting the unacknowledged segments (in this case only segment with the sequence number 7), because the sender has to assume that all outstanding segments are lost.
- At 9.14 seconds in the connection an ACK arrives at the sender. A question arises whether this ACK is an ACK for the original transmission of segment 7 or this is an ACK for the retransmission? As mentioned in [4] the sender does not attach any nonce to retransmitted segments, and the receiver acknowledges such segments with nonce value 0. The arrived segment has the nonce value 1, so the sender is able to determine that this ACK is an ACK for the original transmission of segment 7, and the timeout was spurious.
- The sender can now continue sending with the last unsent segment before the delay spike has happened.
- It can also restore the congestion window states cwnd and ssthresh to the last known values.
- From 9.72 seconds on, all ACKs for the original segments, starting with the segment 8, arrive at the sender, which means that no duplicate segments had been sent spuriously.

Listing 4.2: Spurious timeout detection with the new algorithm

```
Sender:: ECN Nonce is set for syn packet to 1 for seqno: 0

0.721333 Sender received ecnsp_ack for seqno 0 with nonce 0
0.721333 Sender:: send segment with seqno: 1 and nonce: 0
0.721333 Sender:: send segment with seqno: 2 and nonce: 1
1.301333 Receiver:: received segment with seqno: 1 and ecn_nonce 0
1.581333 Receiver:: received segment with seqno: 2 and ecn_nonce 1

1.612000 Sender received ecnsp_ack for seqno 1 with nonce 0
1.612000 Sender:: send segment with seqno: 3 and nonce: 1
1.612000 Sender:: send segment with seqno: 4 and nonce: 1

1.892000 Sender received ecnsp_ack for seqno 2 with nonce 1
1.892000 Sender:: send segment with seqno: 5 and nonce: 1
1.892000 Sender:: send segment with seqno: 6 and nonce: 0
2.192000 Receiver:: received segment with seqno: 3 and ecn_nonce 1
2.472000 Receiver:: received segment with seqno: 4 and ecn_nonce 1

2.502667 Sender received ecnsp_ack for seqno 3 with nonce 1
2.502667 Sender:: send segment with seqno: 7 and nonce: 1
2.502667 Sender:: send segment with seqno: 8 and nonce: 1
2.752000 Receiver:: received segment with seqno: 5 and ecn_nonce 1
```

```
2.782667 Sender received ecnsp_ack for seqno 4 with nonce 1
2.782667 Sender:: send segment with seqno: 9 and nonce: 1
2.782667 Sender:: send segment with seqno: 10 and nonce: 0

3.000000: Intercept

3.032000 Receiver:: received segment with seqno: 6 and ecn_nonce 0

3.062667 Sender received ecnsp_ack for seqno 5 with nonce 1
3.062667 Sender:: send segment with seqno: 11 and nonce: 1
3.062667 Sender:: send segment with seqno: 12 and nonce: 0
3.312000 Receiver:: received segment with seqno: 7 and ecn_nonce 0

3.342667 Sender received ecnsp_ack for seqno 6 with nonce 0
3.342667 Sender:: send segment with seqno: 13 and nonce: 1
3.342667 Sender:: send segment with seqno: 14 and nonce: 1

5.922667 timeout occured

Sender:: last segment send: 14 last received segment: 6

5.922667 EcnspFullTcpAgent::timeout_action - first retransmission  7

Retransmission:: new nonce for packet 7 is: 0
5.922667 Sender:: send segment with seqno: 7 and nonce: 0

9.142908 Sender received ecnsp_ack for seqno 7 with nonce 1

9.142908 Spurious timeout detected for seqno 7

Sender:: continue sending with seqno 15
Sender:: ecnsp_cc is set to 1

Ecnsp::CC: new cwnd=8, new ssthresh=20

9.142908 Sender:: send segment with seqno: 15 and nonce: 0
9.142908 Sender:: send segment with seqno: 16 and nonce: 0
9.412241 Receiver:: received segment with seqno: 8 and ecn_nonce 1
9.692241 Receiver:: received segment with seqno: 9 and ecn_nonce 1

9.722908 Sender received ecnsp_ack for seqno 8 with nonce 1
9.722908 Sender:: send segment with seqno: 17 and nonce: 0
9.722908 Sender:: send segment with seqno: 18 and nonce: 0
9.972241 Receiver:: received segment with seqno: 10 and ecn_nonce 0

10.002908 Sender received ecnsp_ack for seqno 9 with nonce 1
10.002908 Sender:: send segment with seqno: 19 and nonce: 1
10.002908 Sender:: send segment with seqno: 20 and nonce: 0
10.252241 Receiver:: received segment with seqno: 11 and ecn_nonce 1

10.282908 Sender received ecnsp_ack for seqno 10 with nonce 0
10.282908 Sender:: send segment with seqno: 21 and nonce: 1
10.282908 Sender:: send segment with seqno: 22 and nonce: 1
10.532241 Receiver:: received segment with seqno: 12 and ecn_nonce 0
```

```
10.562908 Sender received ecnsp_ack for seqno 11 with nonce 1
10.562908 Sender:: send segment with seqno: 23 and nonce: 0
10.562908 Sender:: send segment with seqno: 24 and nonce: 0
10.812241 Receiver:: received segment with seqno: 13 and ecn_nonce 1

10.842908 Sender received ecnsp_ack for seqno 12 with nonce 0
10.842908 Sender:: send segment with seqno: 25 and nonce: 0
10.842908 Sender:: send segment with seqno: 26 and nonce: 1
11.092241 Receiver:: received segment with seqno: 14 and ecn_nonce 1

11.122908 Sender received ecnsp_ack for seqno 13 with nonce 1
11.122908 Sender:: send segment with seqno: 27 and nonce: 1
11.122908 Sender:: send segment with seqno: 28 and nonce: 1
11.372241 Receiver:: received segment with seqno: 7 and ecn_nonce 0

11.402908 Sender received ecnsp_ack for seqno 14 with nonce 1
11.402908 Sender:: send segment with seqno: 29 and nonce: 1
11.402908 Sender:: send segment with seqno: 30 and nonce: 0
11.652241 Receiver:: received segment with seqno: 15 and ecn_nonce 0
11.932241 Receiver:: received segment with seqno: 16 and ecn_nonce 0

11.962908 Sender received ecnsp_ack for seqno 15 with nonce 0
11.962908 Sender:: send segment with seqno: 31 and nonce: 0
11.962908 Sender:: send segment with seqno: 32 and nonce: 0
12.212241 Receiver:: received segment with seqno: 17 and ecn_nonce 0

12.242908 Sender received ecnsp_ack for seqno 16 with nonce 0
12.242908 Sender:: send segment with seqno: 33 and nonce: 1
12.242908 Sender:: send segment with seqno: 34 and nonce: 1
12.492241 Receiver:: received segment with seqno: 18 and ecn_nonce 0

12.522908 Sender received ecnsp_ack for seqno 17 with nonce 0
12.522908 Sender:: send segment with seqno: 35 and nonce: 1
12.522908 Sender:: send segment with seqno: 36 and nonce: 0
12.772241 Receiver:: received segment with seqno: 19 and ecn_nonce 1
```

Figure [4.12] shows the enhancement of the new algorithm. Another big advantage of the new algorithm is that there is no spurious reduction of the load at the sender. The sender is able to restore cwnd and ssthresh to the last known values, which increases the TCP throughput.

Figure [4.13] shows this enhancement.

- The senders congestion window grows normally till second 3.
- Here the delay spike begins.
- At 5.92 seconds the spurious timeout occurs, and the sender is forced to perform slow start, and sets its cwnd to 1 segment.
- At 9.14 seconds the next ACK arrives at the sender, but now it is able to determine that the timeout was spurious and can restore the cwnd and sstresh states to the last known values before the timeout occurred.
- This behavior is a big advantage, because the sender does not need to go in the congestion avoidance phase, can restore the congestion control states immediately.

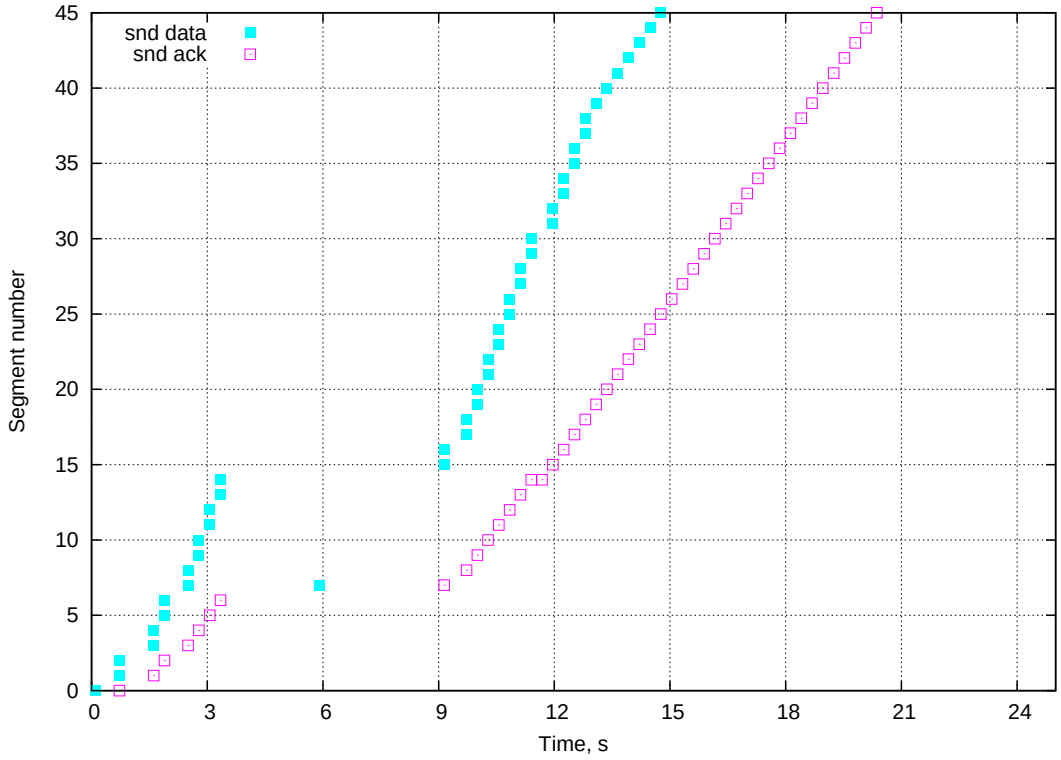

Figure 4.12: TCP sender response to a spurious timeout using the new algorithm with ecnsp_param=0

- At 12.8 seconds the cwnd value has reached the ssthresh value
- From this point the cwnd value is increased linearly.

As described in section [4] the chance to detect a spurious timeout with ecn_param of 0 is 50%. But there is a possibility to increase this. The next simulation shows the new algorithm using an ecn_param of 1, which means that the sender checks 2 segments after a timeout. Now, the chance to detect a spurious timeout is 75%. As shown in figure [4.14] the sender has to wait until 2 ACKs arrive at the sender, after a timeout occurs. When 2 ACKs arrive, it can compare the nonce values and determine if the timeout was spurious or not. If you compare figure [4.12] and figure [4.14] you can see, that it is no advantage, if the sender has to wait for more than one segment after the timeout. Then, there is a possibility that an original ACK arrives at the sender. Then the ACK for the retransmission arrives, and the algorithm would check the wrong nonce value. There is no guaranty that 2 original ACKs in series arrive at the sender. But if it is so, the chance to detect a spurious timeout is increased to 75%. As we can see in figure [4.15] the sender can not determine immediately if the timeout was spurious (see also figure [4.13]) or not. It has to wait until an additive ACK arrives at the sender. If we compare this with figure [4.13] the sender has to wait 0.37 seconds longer. This might be not very long, but it would be better to react immediately.

Figure [4.16] shows a comparison of some ecnsp_param variants and F-RTO [3.2.3]. It is clear that the use of an ecnsp_param=0 is the fastest variant of the ECNSP Algorithm [4], but also the variant with the highest chance for a receiver with a bath faith to guess the correct nonce value(s). Probably the **best choice** would be to use an ecnsp_param of 3 (checking the nonce values of 4 ACKs in series). When the TCP window size is large enough an ecnsp_param of 4,5 or 6 is also thinkable.

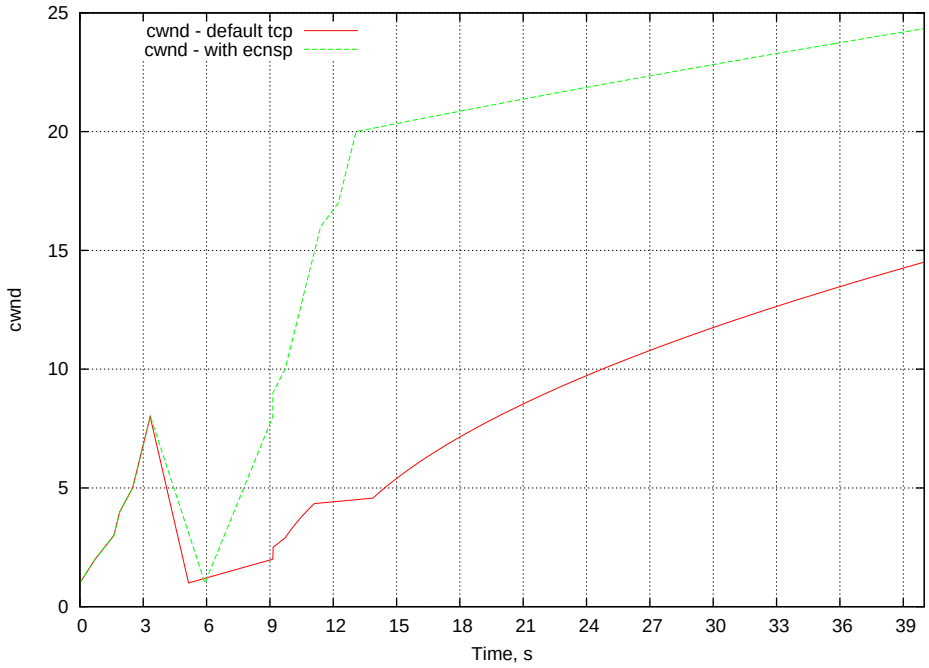

Figure 4.13: TCP sender response to a spurious timeout using the new algorithm with ecnsp_param=0

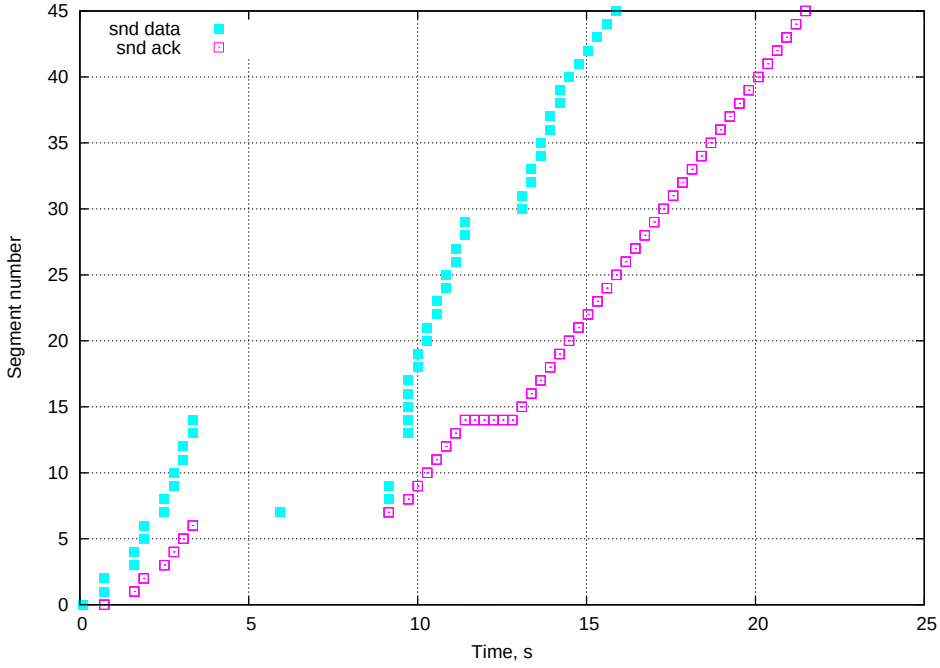

Figure 4.14: TCP sender response to a spurious timeout using the new algorithm and ecnsp_param=1

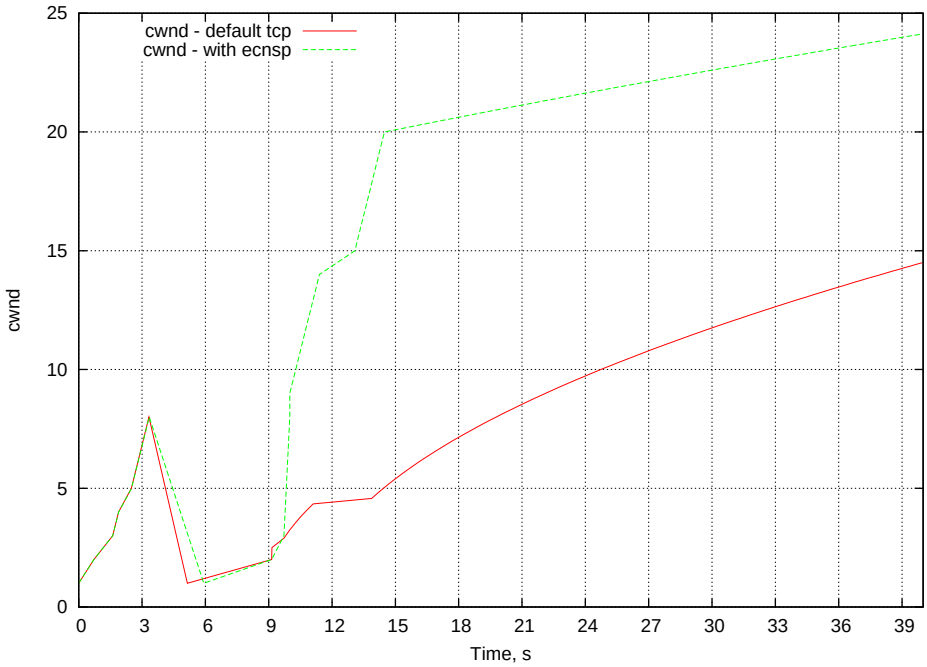

Figure 4.15: TCP sender response to a spurious timeout using the new algorithm and ecnsp_param=2

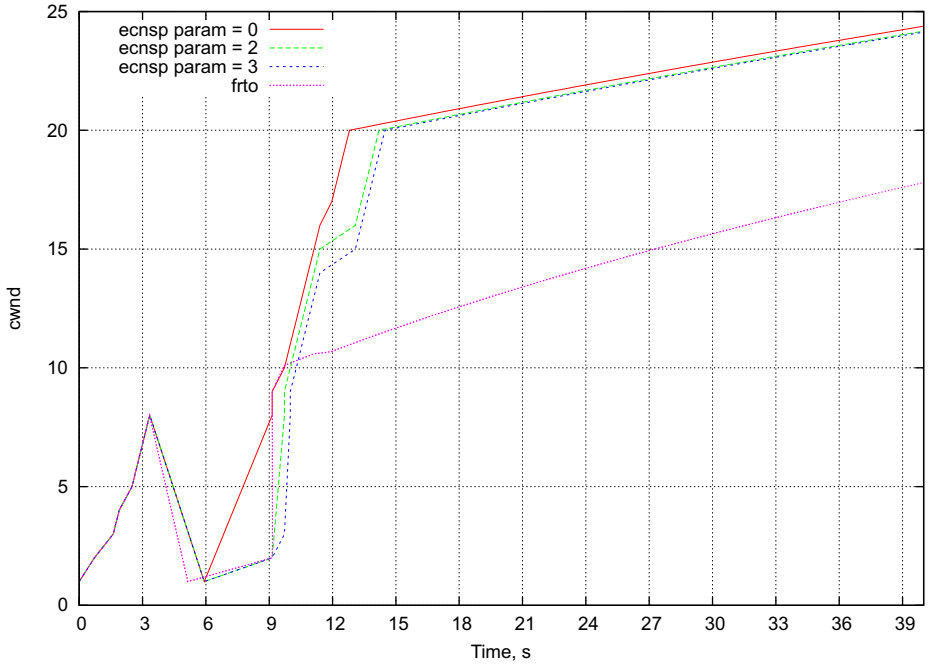

Figure 4.16: Comparison of some ecnsp_param variants and F-RTO

4.5.4 Restoring the congestion control state

After detection of a spurious timeout there are several strategies to restore TCP's congestion control state. Three options are given below.

- variant 1: full undo: cwnd = last known cwnd; ssthresh = last known ssthresh
- variant 2: ssthresh = last know ssthresh; if ssthresh $<$ 2: ssthresh = 2; cwnd = 1
- variant 3: ssthresh = last know ssthresh; if ssthresh $<$ 2: ssthresh =2;
  cwnd = last known cwnd / 2; if cwnd $<$ 2: cwnd = 1

The first option, complete restoration, is to set the slow start threshold and the congestion window to the values before the timeout. The second, partial restoration, is done normally by TCP. It sets the slow start threshold to the half of the old congestion window. The third option is also a little bit different from the second option. The figure [4.17] shows that the first option is the best.

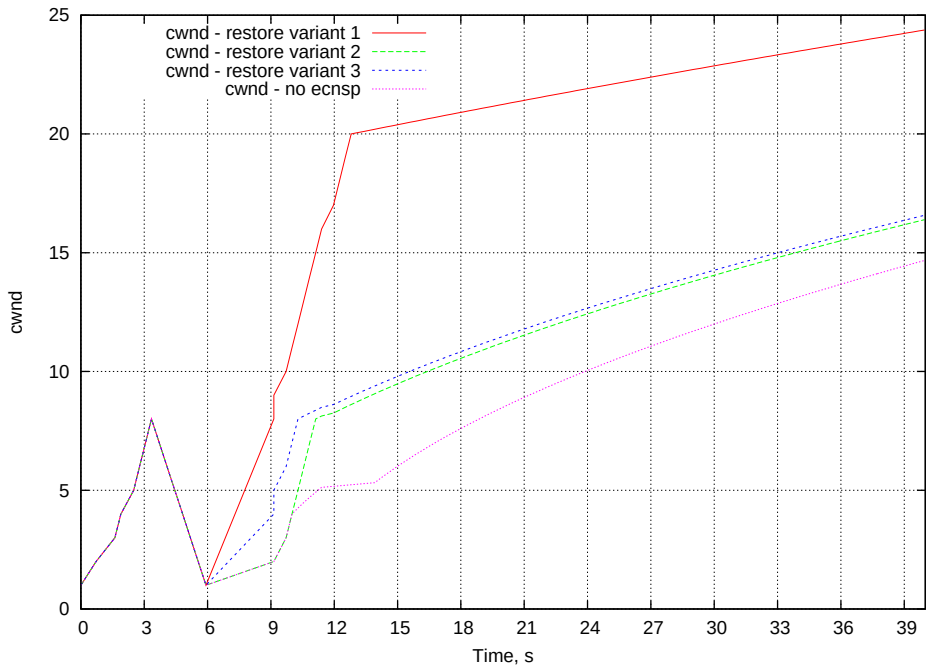

Figure 4.17: Restoring the congestion control state

4.5.5 Spurious fast retransmit reaction with ECNSP

As described in this section, TCP is not able to determine by default if a spurious fast retransmit is spurious or not. To avoid this effect the new algorithm [4] with an **ecn_param of 0** can be used to detect spurious fast retransmits.

Figure [4.18] shows this scenario. At first a spurious timeout and later in the connection a spurious fast retransmit detection with ECNSP is shown.

- At second 3 in the connection a delay spike begins
- At 5.92 seconds the spurious timeout occurs, and the sender is forced to perform slow start, and sets its cwnd to 1 segment.
- At 9.14 seconds the next ACK arrives at the sender, but now it is able to determine that the timeout was spurious and can restore the cwnd and ssthresh state to the last known values before the timeout occurred.
- This behavior is a big advantage, because the sender does not need to go in the congestion avoidance phase can restore the congestion control states immediately.
- Later in the connection the next problem occurs. A single segment seems lost. The sender receives 3 DUPACKs and performes a fast retransmit. So it sends the lost segment again, at 14.2 seconds.
- The sender now retrieves all outstanding ACKs. Now an ACK for the retransmitted segments arrives at the sender. With ECNSP the sender is now able to determine if this ACK is the acknowledgement for the original (spurious lost segment) or the acknowledgement for the retransmitted segment.
- It is the ACK for the original segment, because the sender uses ECNSP with the ecnsp_param=0, and the nonce differs from that of the retransmitted segment.
- So the sender can restore the congestion control states immediately.

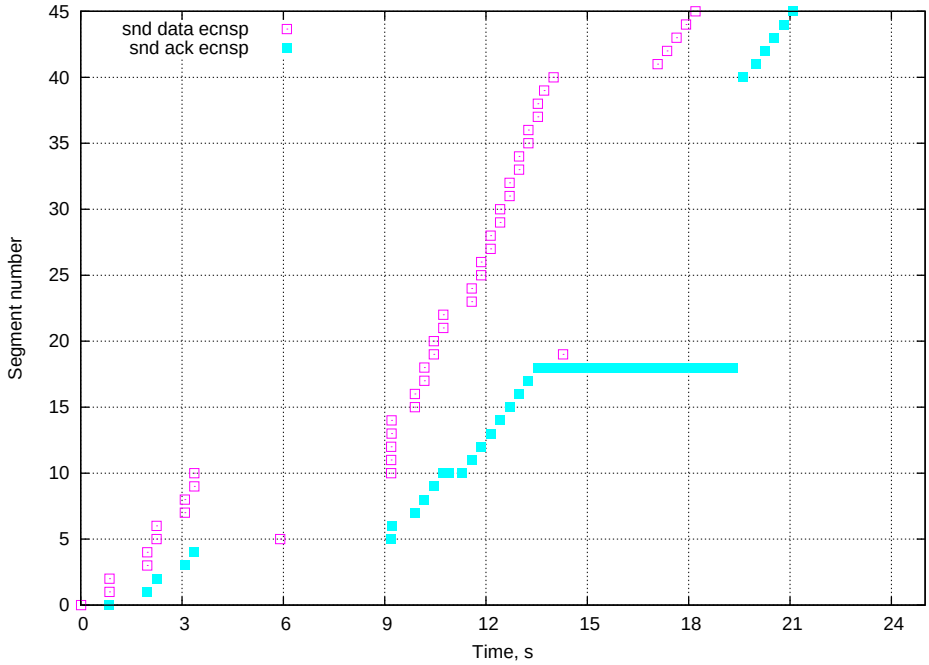

Figure 4.18: Spurious fast retransmit reaction with ECNSP

Figure [4.18] shows the corresponding congestion window. The sender can restore the congestion control states, after detecting the spurious fast retransmit at 20.1 seconds. The chance to detect the spurious fast retransmit is 50%, because only one nonce is checked (ecnsp_param=0).

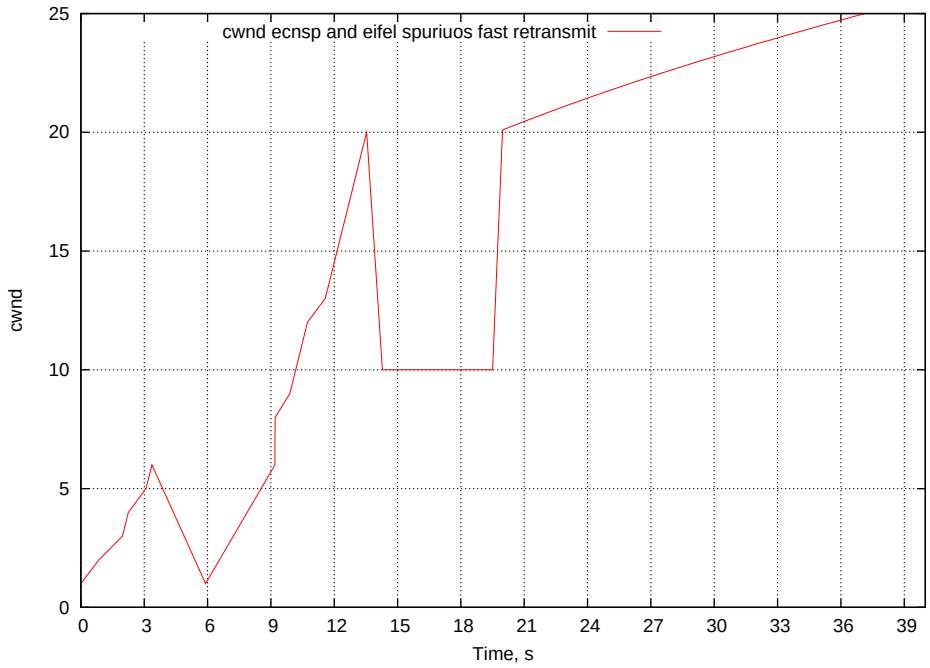

Figure 4.19: cwnd of a spurious fast retransmit reaction with ECNSP

4.5.6 Multiple segment loss from one window

The new algorithm [4] is able to react amazing to spurious loss events, when it happens for one segment in the same window, and we have a ecnsp_param of 1, which means that the sender only checks one segment after a spurious timeout. If we consider the effect of spurious fast retransmits, the ecnsp_param can only be 1. With such a configuration the reaction to a spurious loss event is as same fast as Eifel [3.2.1].
But when we have multiple loss of segments of the same window, the ecnsp_param should always be 1. That means that the chance to detect a spurious loss event is exactly 50% for each lost segment. One possible solution would be to use FACK (MM96) for faster recovery when more than one segment is spurious lost from the same window. FACK is not standardized by IETF (IET) due to concerns with operations in presence of packet re-ordering. Linux TCP uses FACK by default but disables when packet reordering is detected.

4.5.7 Comparison with other algorithms

In this section we will compare the new algorithm with already existing algorithms, when a spurious timeout or a spurious loss event occurs. We will find out advantages and disadvantages of the respective algorithms. The table below shows whether an algorithms detect spurious timeouts and spurious loss events or not.

Algorithm	Detection of spurious timeouts	Detection of spurious fast retransmits
ECNSP	yes	yes
EIFEL	yes	yes
F-RTO	yes	no
D-SACK	yes	yes

ECNSP:

- Advantages:
  - There is no overhead.
  - It detects spurious timeouts and spurious fast retransmits immediately - like EIFEL.
- Disadvantages:
  - The chance to detect depends on ecnsp_param. (50%,75%,87% chance)

EIFEL:

- Advantages:
  - Detection always works.
  - It detects spurious timeouts and spurious fast retransmits immediately.
- Disadvantages:
  - because of timestamps, big extra overhead in each segment needed

F-RTO:

- Advantages:
  - Timestamps and ecn capability are not needed
- Disadvantages:
  - detect only spurious timeouts

D-SACK:

- Advantages:
  - Timestamps and ecn capability are not needed
- Disadvantages:
  - detection of spurious loss timeouts and spurious fast retransmits takes much longer than EIFEL or ECNSP

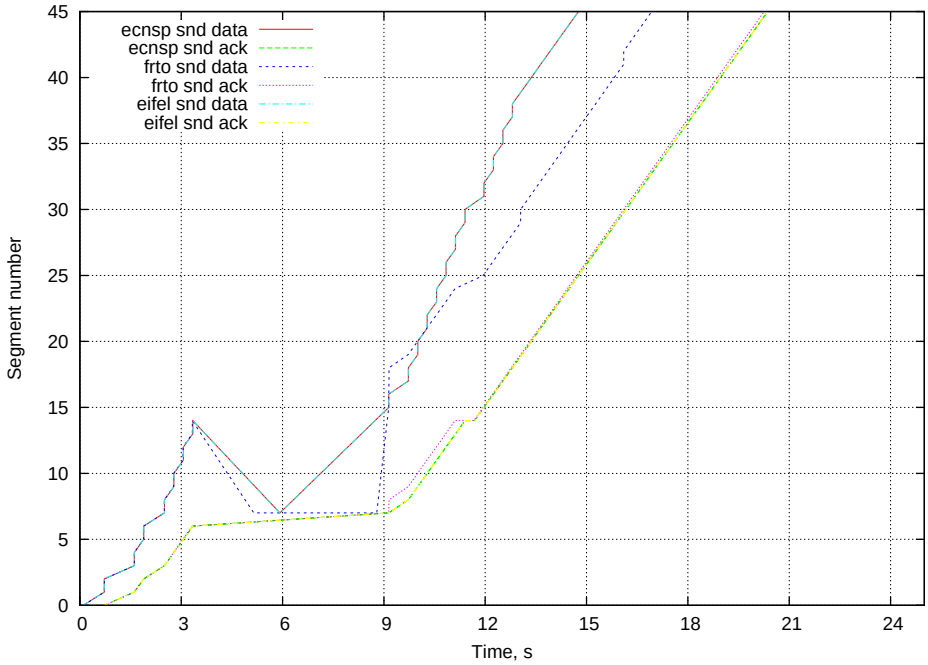

Figure 4.20: Reaction to a spurious timeout

Figure [4.20] shows a simulation of a spurious timeout with ECNSP (ecnsp_param=0), EIFEL and F-RTO (enhanced version). If we compare EIFEL and ECNSP, we can see that both algorithms react alike. The big disadvantage of EIFEL is the huge overhead of 12 bytes in each segment and ACK.

Comparison of the three algorithms considering the restoration of the congestion control state shows that they all perform much better than a TCP implementation without a feature to detect spurious timeouts or spurious loss event in general. All of them can restore the the congestion window and the slow start threshold very quick and efficiently (only cwnd is shown).

Figure [4.22] shows a simulation of a spurious fast retransmit with ECNSP (ecnsp_param=0) and EIFEL. (F-RTO is not able to detect spurious fast retransmits) If we compare EIFEL and ECNSP, we can see that both algorithms react alike. The big disadvantage of EIFEL is the huge overhead of 12 bytes in each segment and ACK. Figure [4.23] shows the corresponding congestion window of this scenario.

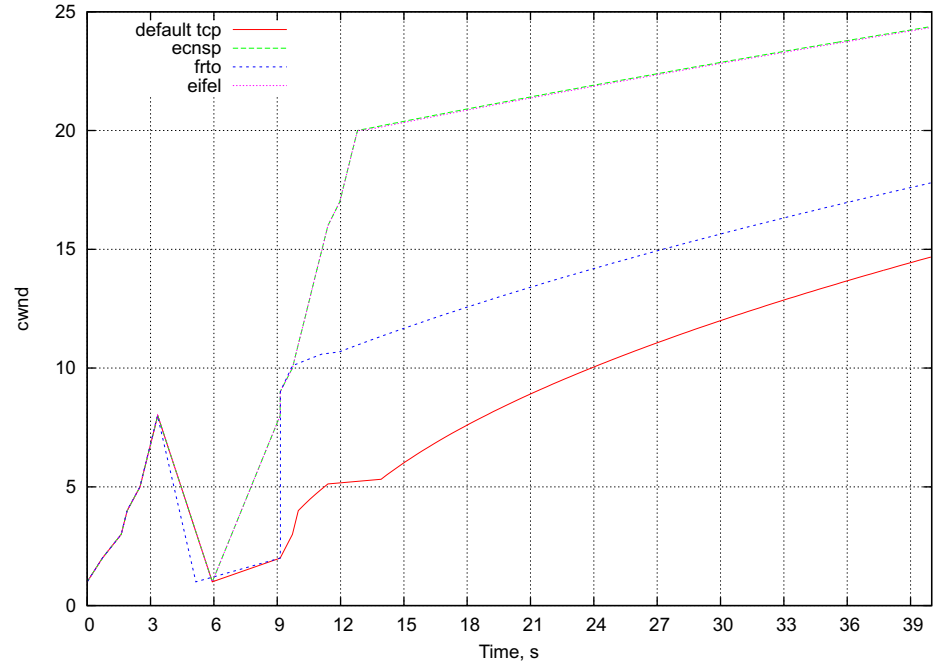

Figure 4.21: Restoration of the congestion control state

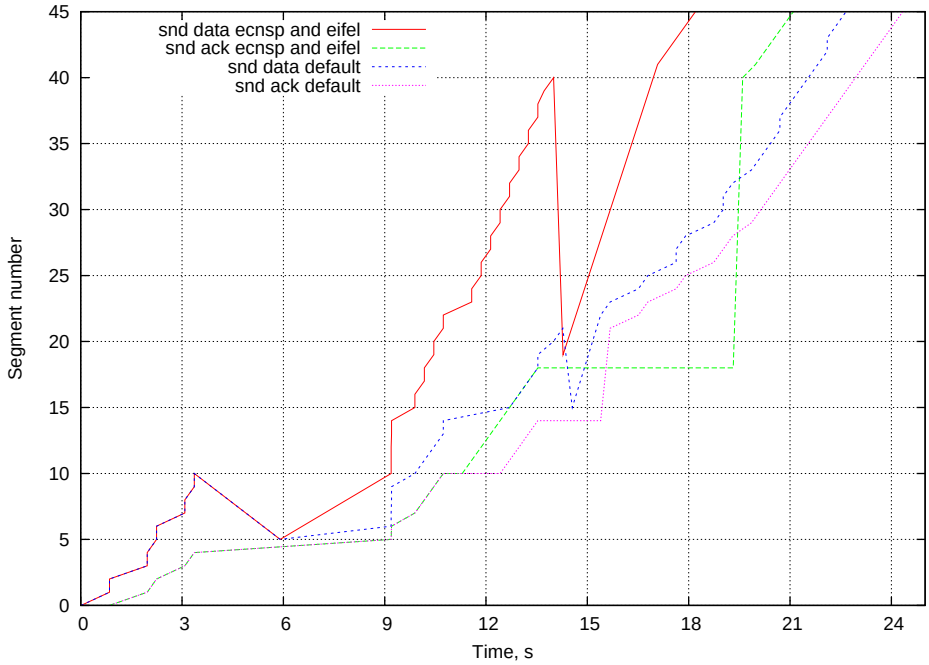

Figure 4.22: Reaction to a spurious fast retransmit

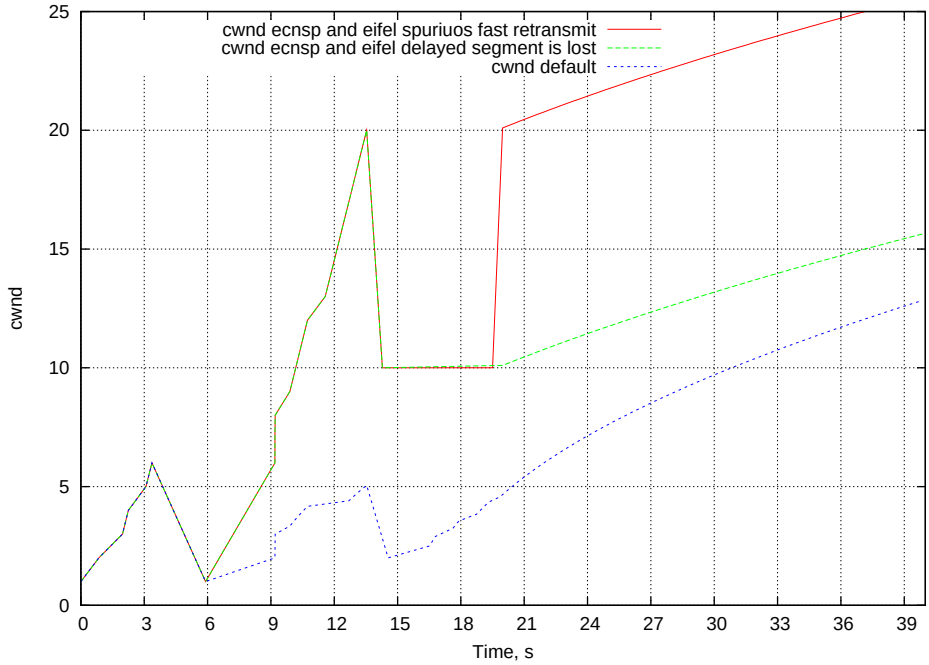

Figure 4.23: Reaction to a spurious fast retransmit

4.5.8 Security

A very important aspect of all kinds of protocols is security. Two security aspects which are relevant when using ECNSP [4] are worth mentioning.

- receiver could guess the correct nonce value(s)
- receiver could have bad faith

When using ECNSP the extra information for the sender is coded as ECN codepoint, which can have a random value of 0 or 1. The fact that the value is random is very security relevant. The following table shows the chance of a receiver with bad faith to guess the correct nonce(s).

ecnsp_param	chance to guess
0	50%
1	25%
2	12,5%
3	6.25%
4	3.20%
5	1.60%

If we use **ecn_param=0** the chance to guess the correct nonce is 50%, which might be a little bit too high. When we take **ecnsp_param=1** the chance to guess the correct nonces (there are 2 ACKS to check) is now 25%. This is much better. The other variants, **ecnsp_param =2 or 3** can make the chance to guess the correct nonce values lower and are probably the **best choices**, because this variants are also very fast too (see figure [4.16]). When we use **ecnsp_param 4 or 5** the advantages of using ECNSP would be decreased.

Chapter 5

Enhancements

5.1 Suggestion one

One possible improvement of ECNSP would be another use of the proposed ECN nonce [2.6.4]. To encode the one bit nonce the following ECN codepoints are used:

NONCE	RFC2481	PROPOSED
00	ECN-incapable	ECN-incapable
10	No congestion	No congestion, Nonce=0 – ECT(0)
01	Undefined	No congestion, Nonce=1 – ECT(1)
11	Congestion	Congestion, Nonces cleared

Here is another possibility:

Nonce	Variant
000	ECN-incapable
010	No congestion, Nonce=0
010	No congestion, Nonce=1
011	Retransmitted segment(s)
111	Congestion, Nonces cleared

This new declaration would bring the following advantages:

- No ecnsp_param is needed. The sender does not need to way for more than one ACK after a timeout, and can therefore immediately determine if the spurious loss event was spurious or not.
- The retransmission ambiguity would be eliminated.
- It would always work including the cases of spurious fast retransmits.
- No security risks – the random nonce would be used

Disadvantages:

- It is not proposed by IETF, and hence it is a completely different specification of ECN nonce. [2.6.4]
- ECN capable sender, receiver and router are needed.

5.2 Suggestion two

Another possible improvement of the whole spurious loss event difficulty could be the use of EIFEL in a little bit different form than proposed in [3.2.1]. The use of timestamps brings a big disadvantage: There is a overhead of 12 byte in each segment and each ACK. That is heavy weight. The advantage of using the timestamp option is that this scheme is already a proposed standard and that it is widely deployed (All00b).

The possibility would be to support compression of TCP options, to decrease the overhead of 12 bytes in each segment and ACK.

Chapter 6

Relevant changes in NS2

This chapter lists the most important changes made to NS2 2.29 (NS206) to enable ECNSP [4]. There are no TCL or Phython scripts I have used to generate the simulation results. The following listings contain the most important changes only.

6.1 Delay spike generation - spurious timeout generation

Generation of delay spikes are important to simulate the effect of spurious timeouts. To generate them I have used Anrei Gurtov's (Gur02) implementation of the hiccup tool. This tool is for triggering delay spikes of arbitrary length (e.g. a few seconds) for all queue types in NS2. Compared to hiccup this code has two benefits: it places delays after the queue and can be used in both directions simultaneously. A remaining drawback is that it allows the packet currently "in the air" to complete transmission after a delay spike begins.

Listing 6.1: delay patch

```
File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/queue/queue.cc

Queue::~Queue() {
}

Queue::Queue() : Connector(), blocked_(0), unblock_on_resume_(1),
                 qh_(*this),
+                pq_(0), no_resume(0) ,
                 last_change_(0), /* temporarily NULL */
                 old_util_(0), period_begin_(0), cur_util_(0),
                 buf_slot_(0),
                 util_buf_(NULL)
{
        bind("limit_", &qlim_);
        bind("util_weight_", &util_weight_);
        bind_bool("blocked_", &blocked_);
        bind_bool("unblock_on_resume_", &unblock_on_resume_);
        bind("util_check_intv_", &util_check_intv_);
        bind("util_records_", &util_records_);

        if (util_records_ > 0) {
```

```
                util_buf_ = new double[util_records_];
                if (util_buf_ == NULL) {
                        printf("Error allocating util_bufs!");
                        util_records_ = 0;
                }
                for (int i = 0; i < util_records_; i++) {
                        util_buf_[i] = 0;
                }
        }
        printf("Queue constructor called \n");
}

...


void Queue::resume()
{
        double now = Scheduler::instance().clock();
+       Packet* p; /*delay spike*/
+       if (no_resume) return; //AG
+       p = deque();


        if (p != 0) {
                target_->recv(p, &qh_);
        } else {
                if (unblock_on_resume_) {
                        utilUpdate(last_change_, now, blocked_);
                        last_change_ = now;
                        blocked_ = 0;
                }
                else {
                        utilUpdate(last_change_, now, blocked_);
                        last_change_ = now;
                        blocked_ = 1;
                }
        }
}

...

+ //AG it was difficult to put this anywhere else since resume() calls
+ //outside of Queue class generate a Scheduler error
+ int Queue::command(int argc, const char*const* argv) {
+       if (argc == 2 && !strcmp(argv[1],"block")) {
+                       no_resume=true;
+                       return (TCL_OK);
+               }
+               if (argc == 2 && !strcmp(argv[1],"unblock")) {
+                       no_resume=false;
+                       resume();
+                       return (TCL_OK);
+               }
+       return Connector::command(argc, argv);
+ }
```

```
...


File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/queue/queue.h

...

protected:
        Queue();
        void reset();
        int qlim_;              /* maximum allowed pkts in queue */
        int blocked_;           /* blocked now? */
        int unblock_on_resume_; /* unblock q on idle? */
        QueueHandler qh_;
        PacketQueue *pq_;       /* pointer to actual packet queue
                                 * (maintained by the individual
                                 * disciplines
                                 * like DropTail and RED). */
        double true_ave_;       /* true long-term average queue size */
        double total_time_;     /* total time average queue size
                                        compute for */

+       int no_resume;          /* for generating delay spikes */
+       int command(int argc, const char*const* argv);

        void utilUpdate(double int_begin, double int_end, int link_state)
            ;
        double last_change_;  /* time at which state changed/utilization
                                        measured */
        double old_util_;     /* current utilization */

...
```

6.1.1 Deadline drop

This code provides setting of TTL field to now+min(RTO, object lifetime) for UDP, TFRC, and TCP. A new queue module DropTail/DropExpired deletes stale packets from the queue. This code is also taken from (Gur02).

Listing 6.2: deadline drop patch

```
File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/Makefile

...

        diffserv/dsEdge.o diffserv/dsCore.o \
        diffserv/dsPolicy.o diffserv/ew.o diffserv/dewp.o \
        queue/red-pd.o queue/pi.o queue/vq.o queue/rem.o \
-       queue/gk.o \
+       queue/gk.o queue/drop-expired.o queue/lib-atp.o \
        pushback/rate-limit.o pushback/rate-limit-strategy.o \
        pushback/ident-tree.o pushback/agg-spec.o \
```

```
12          pushback/logging-data-struct.o \

...

File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/Makefile.in

...

          diffserv/dsEdge.o diffserv/dsCore.o \
          diffserv/dsPolicy.o diffserv/ew.o diffserv/dewp.o \
22        queue/red-pd.o queue/pi.o queue/vq.o queue/rem.o \
-         queue/gk.o \
+         queue/gk.o queue/drop-expired.o queue/lib-atp.o \
          pushback/rate-limit.o pushback/rate-limit-strategy.o \
          pushback/ident-tree.o pushback/agg-spec.o \
          pushback/logging-data-struct.o \

...

File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/apps/udp.cc
32

...

          #include "address.h"
          #include "ip.h"

+         #include "../queue/lib-atp.h"

          static class UdpAgentClass : public TclClass {

42 ...

                              target_->recv(p);
                    }
                    idle();
                    }
          void UdpAgent::recv(Packet* pkt, Handler*)
          {
+                   atp_log_ttl(pkt);
                    if (app_) {
52                  //If an application is attached,pass the data to the app
                              hdr_cmn* h = hdr_cmn::access(pkt);
                              app_->process_data(h->size(), pkt->userdata())

...


File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/tcp/tfrc-sink.cc

...

62

          #include "formula-with-inverse.h"
          #include "flags.h"

+         #include "../queue/lib-atp.h"
```

```
        static class TfrcSinkClass : public TclClass {

...

                last_report_sent = now;
                rcvd_since_last_report = 0;
                losses_since_last_report = 0;
+               atp_set_ttl(pkt, Scheduler::instance().clock()+rtt_+3);
                send(pkt, 0);
        }
}

...

File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/tcp/tfrc.cc

...

#include "flags.h"

+       #include "../queue/lib-atp.h"

        int hdr_tfrc::offset_;
        int hdr_tfrc_ack::offset_;

...

                tfrch->round_id=round_id;
                ndatapack_++;
+               atp_set_ttl(p, Scheduler::instance().clock() + t_rtxcur_)
    ;
                ndatabytes_ += size_;
                if (useHeaders_ == true) {
                        hdr_cmn::access(p)->size() += headersize_;
                }
                last_pkt_time_ = now;
                send(p, 0);


File:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/trace/trace.cc

...

        int Trace::get_seqno(Packet* p)
        {
                hdr_cmn *th = hdr_cmn::access(p);
                hdr_tcp *tcph = hdr_tcp::access(p);
                hdr_rtp *rh = hdr_rtp::access(p);
                hdr_rap *raph = hdr_rap::access(p);
+               hdr_tfrc *tfrch = hdr_tfrc::access(p);
                hdr_tfrc_ack *tfrch_ack = hdr_tfrc_ack::access(p);

                packet_t t = th->ptype();
                int seqno;
```

```
122                 /* UDP's now have seqno's too */
                    if (t == PT_RTP || t == PT_CBR ||
                            t == PT_UDP || t == PT_EXP ||
                        t == PT_PARETO)
                            seqno = rh->seqno();
                    else if (t == PT_RAP_DATA || t == PT_RAP_ACK)
                           seqno = raph->seqno();
                    else if (t == PT_TCP || t == PT_ACK ||
                            t == PT_HTTP || t == PT_FTP ||
                      t == PT_TELNET || t == PT_XCP)
132                         seqno = tcph->seqno();
                    else if (t == PT_TFRC)
                            seqno = tfrch->seqno;
+                   else if (t == PT_TFRC_ACK)
+                          seqno = tfrch_ack->seqno;
                    else
                            seqno = -1;
                    return seqno;
            }
```

6.1.2 ECNSP patch

The following listing provides the most important changes I have made to NS2 2.29 (NS206).

Listing 6.3: deadline drop patch

```
FILE:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/common/flags.h

#ifndef ns_flags_h
#define ns_flags_h
#include "config.h"
6 #include "packet.h"

struct hdr_flags {
        unsigned char ecn_;             /* transport receiver notifying
                                         *  transport sender of ECN
                                         *  (ECN Echo bit in TCP header) */
        unsigned char ecn_to_echo_;     /* ecn to be echoed back in the *
                                         * opposite direction *
                                         * (CE bit in IP header) */

16      unsigned char eln_;         /* explicit loss notification (snoop) */
        unsigned char fs_;          /* tcp fast start (work in progress --
           venkat) */
        unsigned char no_ts_;       /* don't use the tstamp of this pkt for
           rtt */
        unsigned char pri_;         /* unused */
        unsigned char ecn_capable_; /* an ecn-capable tranport *
                                     * (ECT bit in IP header) */
        unsigned char cong_action_; /* Congestion Action.  Transport
                                     * sender notifying transport
                                     * receiver of responses to
                                         congestion.
```

```
                                        * (CWR bit in TCP header) */
26
+       unsigned int ecn_nonce_; /* nonce bit sent from sender (
                                        cleared by a marking router) */
        unsigned char qs_;       /* Packet is from Quick-Start window, i.e
            .
                                 * a window following an approved QS
                                    request.
                                 */

        unsigned char& ect()     { return ecn_capable_; }
                                        /* (ECT bit in IP header) */
        unsigned char& ecnecho() { return ecn_; }
36                                      /* (ECN Echo bit in TCP header) */
        unsigned char& ce() { return ecn_to_echo_; }
                                        /* (CE bit in IP header) */
        unsigned char& cong_action() { return cong_action_; }
                                        /* (CWR bit in TCP header-old name)
                                            */
        unsigned char& cwr() { return cong_action_; }
                                        /* (CWR bit in TCP header) */

        unsigned char& qs() { return qs_; } /* Quick-Start packet */
        unsigned int& nonce() {return ecn_nonce_; }
46
        static int offset_;
        inline static int& offset() { return offset_; }
        inline static hdr_flags* access(const Packet* p) {
                return (hdr_flags*) p->access(offset_);
        }
};
#endif


56 FILE:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/tcp/tcp.h

...

        /*ECNSP */
        int ecnsp_;              /* use the ecnsp algorithm or not */
        int ecnsp_cc_;           /* 1-restore CC fully, 0-not all, etc...
            see tcp.cc*/
        int ecnsp_rto_;          /* 0-standard rto, 1-reseed after timeout
            , 2-backoff */
        //int ecnsp_param_;      /* determines how many acks after a
                                    timeout should be checked*/
66
        int usingECNNonce;

        int ecnsp_first_rexmit_;   /* holds nonce of first retransmission
             */
        int ecnsp_last_cwnd_;          /* to save the current cwnd */
        int ecnsp_last_ssthresh_;        /* same for ssthresh */
        int ecnsp_rexmit_seq_;           /* sequence number of
            retransmitted
```

```
                                        segment */
        int ecnsp_rexmit_no_;            /* number of retransmissions */
        void ecnsp_timeout(); /*timeout and new random nonce*/
76      void ecnsp_ack(Packet*);
        void reset_ecnsp();


        /* End of parameters for backwards compatility. */

        /* Parameters for alternate congestion control mechanisms. */
        double k_parameter_;     /* k parameter in binomial controls */
        double l_parameter_;     /* l parameter in binomial controls */
        int precision_reduce_;  /* non-integer reduction of cwnd */
86      int maxburst_;          /* max # packets can send back-2-back */
        int aggressive_maxburst_;       /* Send on a non-valid ack? */
        /* End of parameters for alternate congestion control mechanisms.
            */

        FILE *plotfile_;

...



96      /* Used for ECN */
        int ecn_;               /* Explicit Congestion Notification */
        int cong_action_;       /* Congestion Action.  True to indicate
                                   that the sender responded to
                                      congestion. */
        int ecn_burst_;         /* True when the previous ACK packet
                                 *  carried ECN-Echo. */
        int ecn_backoff_;       /* True when retransmit timer should
           begin
                                    to be backed off.  */
        int ect_;               /* turn on ect bit now? */
        int SetCWRonRetransmit_;  /* True to allow setting CWR on */
106                               /*  retransmitted packets.   Affects */
                                  /*  performance for Reno with ECN.  */
        /* end of ECN */

...


FILE:: /ns-allinone-2.29.2/ns-allinone-2.29/ns-2.29/tcp/tcp.cc

include <stdlib.h>
116 #include <math.h>
#include <sys/types.h>
#include "ip.h"
#include "tcp.h"
#include "flags.h"
#include "random.h"
#include "basetrace.h"
#include "hdr_qs.h"
```

```
#define RAND_BIT_POS 0
#define arraysize 1000
int noncearray[arraysize];

/*tmp array at sender - stores retransmission nonces*/
int noncearray_retr[arraysize];

int noncearray_received[arraysize];


int sender_last_send=0;
int sender_last_received=0;
int sender_first_retransmitted=0;
char received_first_retransmitted_ack ='f';
int count_next_acks=0;

int hdr_tcp::offset_;

static class TCPHeaderClass : public PacketHeaderClass {
public:
        TCPHeaderClass() : PacketHeaderClass("PacketHeader/TCP",
                                             sizeof(hdr_tcp)) {
                bind_offset(&hdr_tcp::offset_);
        }
} class_tcphdr;


...

bind("ncwndcuts_", &ncwndcuts_);
bind("ncwndcuts1_", &ncwndcuts1_);
bind("ecnsp_", &ecnsp_);
bind("ecnsp_cc_", &ecnsp_cc_);
bind("ecnsp_rto_", &ecnsp_rto_);

...

delay_bind_init_one("ecnsp_");
delay_bind_init_one("ecnsp_cc_");
delay_bind_init_one("ecnsp_rto_");
#endif /* TCP_DELAY_BIND_ALL */

...

if (delay_bind_bool(varName, localName, "ecnsp_", &ecnsp_, tracer))
    return TCL_OK;
if (delay_bind(varName, localName, "ecnsp_cc_", &ecnsp_cc_, tracer))
    return TCL_OK;
if (delay_bind(varName, localName, "ecnsp_rto_", &ecnsp_rto_, tracer))
    return TCL_OK;

...

```

```
void TcpAgent::output(int seqno, int reason)
{
        int force_set_rtx_timer = 0;
        Packet* p = allocpkt();
        hdr_tcp *tcph = hdr_tcp::access(p);
        hdr_flags* hf = hdr_flags::access(p);
        hdr_ip *iph = hdr_ip::access(p);



        int databytes = hdr_cmn::access(p)->size();
        tcph->seqno() = seqno;

        double n = Scheduler::instance().clock();
        printf("%f Sender:: send segment with seqno: %d \n",n,seqno);

        tcph->ts() = Scheduler::instance().clock();
        int is_retransmit = (seqno < maxseq_);

        printf("nonce for packet %d is: %d \n",seqno,hf->nonce());

...

        tcph->ts_echo() = ts_peer_;
        tcph->reason() = reason;
        tcph->last_rtt() = int(int(t_rtt_)*tcp_tick_*1000);

        if (ecn_) { /*ecn = true */
                int i;
                double now = Scheduler::instance().clock();
                /*set ECT bit in the IP header to 1*/
                /*ECN capable transport*/
                //printf("Sender set ECT bit in the IP header");
                hf->ect() = 1;

                /* this is hack to get around the SYN packet */
                if( seqno != 0 ) {

                /*set randomnonce for each packet*/
                hf->ecn_nonce_ = randomnonce();

                i=hf->ecn_nonce_;
                this->usingECNNonce = 1;
                noncearray[seqno]=i;
                sender_last_send=seqno;
                printf("%f Sender:: send segment with seqno: %d and nonce
                    : %d\n",now,seqno,i);
                }

        }

...

        /* Check if this is the initial SYN packet. */
        if (seqno == 0) {
```

```
        if (syn_) {
                databytes = 0;
                curseq_ += 1;
                hdr_cmn::access(p)->size() = tcpip_base_hdr_size_;
        }
        if (ecn_) {
                int i;
                /*set randomnonce for each packet*/
                hf->ecn_nonce_ = randomnonce();
                i=hf->ecn_nonce_;
                this->usingECNNonce = 1;
                noncearray[seqno]=i;
                sender_last_send=seqno;
                printf("Sender:: ECN Nonce is set for syn packet to %d
                    for seqno: %d\n",i,seqno);

                /*setting EE (ECN echo) in the TCP header to 1*/
                hf->ecnecho() = 1;
                hf->ect() = 0; /*set ECN capable bit to 0?*/

...



void TcpAgent::ecnsp_timeout() {

        if (!ecnsp_) return;
        double now = Scheduler::instance().clock();

        printf("\nlast send: %d last received: %d\n",
        sender_last_send,sender_last_received);
        printf("\nsegment count to retransmit: %d\n\n", sender_last_send-
            sender_last_received);

        int oldest_seg = (highest_ack_ < 0) ? 0 : int(highest_ack_);
        printf("\noldest unacknowledged segment is: %d\n",oldest_seg);
        if (ecnsp_rexmit_no_ == 0){

        printf("%f EcnspFullTcpAgent::timeout_action - first
            retransmission  %d\n",now,oldest_seg+1);
        //oldest_seg++;
        ecnsp_rexmit_seq_ = oldest_seg;
        ecnsp_rexmit_no_ = 1;
        ecnsp_first_rexmit_ = highest_ack_;
        ecnsp_last_cwnd_ = cwnd_;
        ecnsp_last_ssthresh_ = ssthresh_;

        //save seqno of the first retransmitted segment
        sender_first_retransmitted=ecnsp_rexmit_seq_+1;
        printf("\nsender first retransmitted: %d\n",
        sender_first_retransmitted);

        printf("Retransmission:: new nonce for packet %d is: %d\n",
            ecnsp_rexmit_seq_,noncearray_retr[ecnsp_rexmit_seq_]);
        }
```

```
        else
        {
        if (oldest_seg == ecnsp_rexmit_seq_){
        printf("EcnspFullTcpAgent::timeout_action - Retransmission No: %d
            seqno: %d\n",ecnsp_rexmit_no_+1,oldest_seg);
286
        ecnsp_rexmit_no_++;

        }

        }
}

...

296 /*
 * For every ACK check if 1) timeout occured 2) it was good or bad
     timeout
 * If Ecnsp is enabled, respond to timeouts
 */
void TcpAgent::ecnsp_ack(Packet *p) {

        hdr_tcp *otcp = hdr_tcp::access(p);
        hdr_flags* hf = hdr_flags::access(p);

        double now = Scheduler::instance().clock();
306     int ecnsp_cc_=0;

        printf("\n%f Sender received ecnsp_ack for seqno %d with nonce %d
            \n",now,otcp->seqno(),hf->ecn_nonce_);

        sender_last_received=otcp->seqno();
        noncearray_received[otcp->seqno()]=hf->ecn_nonce_;

        hdr_tcp *tcph = hdr_tcp::access(p);
        ecnsp_rexmit_seq_ = otcp->seqno();

316     //if((received_first_retransmitted_ack=='t')
        //&&((ecnsp_rexmit_seq_->sender_first_retransmitted)&&(
            sender_first_retransmitted>0)))

        //param =1 , 50% chance to detect
        if((otcp->seqno()==sender_first_retransmitted)&&(
            sender_first_retransmitted>0))
        {

        //count_next_acks++;
        //printf("\n\n%f Sender: count_next_acks %d seqno: %d \n",now,
            count_next_acks,otcp->seqno());
        //if(count_next_acks==1)
326     //{

        now = Scheduler::instance().clock();
        printf("\n\n%f Spurious timeout detected for seqno %d \n\n",
        now,sender_first_retransmitted);
```

```
        //t_seqno_ = sender_first_retransmitted+1;
        printf("\ncontinue sending with seqno %d \n",sender_last_send);
        t_seqno_ =15;

        if (ecnsp_rto_==3) minrto_++;

        /* ecnsp_param value*/
        ecnsp_cc_ = 1;
         switch (ecnsp_cc_) {
                case 1 : // full undo
                printf("ecnsp_cc is set to 1\n\n\n\n");
                cwnd_ =  ecnsp_last_cwnd_;
        ssthresh_ = ecnsp_last_ssthresh_;
        break;
        case 2 :
                printf("ecnsp_cc is set to 2\n\n\n\n");
                // cwnd/=2, ssthresh=cwnd
        cwnd_ = ecnsp_last_cwnd_/2;
        if (int(cwnd_)<2) cwnd_ = 1;
        ssthresh_ = cwnd_; //eifel_last_ssthresh_/2;
        if (int(ssthresh_) <2) ssthresh_ = 2;
        break;
                case 3:
                printf("ecnsp_cc is set to 3\n\n\n\n");
                // sstresh=cwnd, cwnd=1
        ssthresh_ = int(ecnsp_last_cwnd_);
        if (int(ssthresh_) <2) ssthresh_ = 2;
        cwnd_ = 1;
        break;
        case 4:
                printf("ecnsp_cc is set to 4\n\n\n\n");
                // sstresh=cwnd, cwnd/=2
        ssthresh_ = int(ecnsp_last_cwnd_);
        if (int(ssthresh_) <2) ssthresh_ = 2;
        cwnd_ = ecnsp_last_cwnd_/2;
        if (int(cwnd_)<2) cwnd_ = 1;
        break;
        case 5:
                printf("ecnsp_cc is set to 5\n\n\n\n");
                // sstresh=cwnd, cwnd=cwnd
        ssthresh_ = int(ecnsp_last_cwnd_);
         if (int(ssthresh_) <2) ssthresh_ = 2;
                cwnd_ = ecnsp_last_cwnd_;
                if (int(cwnd_)<2) cwnd_ = 1;
                break;


        default:
        //  nothing: We go on with standard TCP behavior
        cwnd_ = 1;
        ssthresh_ = ecnsp_last_ssthresh_/2;
        }

                //reset
```

```
386                 received_first_retransmitted_ack='f';
                    sender_first_retransmitted=0;


          printf("Ecnsp::CC: new cwnd=%d, new ssthresh=%d\n",
                    (int)cwnd_,(int)ssthresh_);
          printf("Ecnsp::CC: old RTO %.2f\n", rtt_timeout());

                    /* eifel_rto=2 is dealed with else where */

396                 if (ecnsp_rto_==1) {
                    // reseed RTO with new RTT sample (ra=rtt, va=rtt/2, t=
                        rtt+4*va)
          double now = Scheduler::instance().clock();
          t_srtt_=0; //this forces rtt_update to reset srtt and svar
          rtt_update(now - tcph->ts_echo());
          set_rtx_timer();
          printf("EcnSP::CC: new RTO %.2f\n", rtt_timeout());
                    }
                    //}
          //}
406       //else if (( ecnsp_rexmit_no_ > 0)&&(otcp->seqno()==
          sender_first_retransmitted)) {
          //printf("\n\n\n%f YES firts retransmitted ack is here: %d\n\n",
              now,otcp->seqno());
          //received_first_retransmitted_ack='t';

          }
          }

/* init eifel-related variables */
void TcpAgent::reset_eifel() {
416     eifel_ts_first_rexmit_ = 0;
        eifel_rexmit_seq_ = 0;
        eifel_rexmit_no_ = 0;
}
```

6.1.3 Spurious fast retransmit generation

To generate spurious timeouts I have used Andrei Grutov's (Gur02) tool as mentioned in section [6.1]. But this tool does not support any feature to simulate spurious fast retransmits. One possibility to do this, is a link breakdown and an alternative route with a heavy delay, so that the ACK arrives exactly at the time when the sender has reacted to the spurious fast retransmit.

6.2 Related Software

This section provides an overview of the used software.

6.2.1 Latex

Latex LaTeX (Lat) is a document markup language and document preparation system for the TeX typesetting program. It is widely used by mathematicians, scientists, philosophers, engineers, and scholars in academia and the commercial world, and by others as a primary or intermediate format (e.g. translating DocBook and other XML-based formats to PDF) because of the quality of typesetting achievable by TeX.

It offers programmable desktop publishing features and extensive facilities for automating most aspects of typesetting and desktop publishing, including numbering and cross-referencing, tables and figures, page layout and bibliographies.

6.2.2 Gnuplot

Gnuplot (Gnu) is a portable command-line driven interactive data and function plotting utility for UNIX, IBM OS/2, MS Windows, DOS, Macintosh, VMS, Atari and many other platforms.

6.2.3 WinEDT

WinEdt (**?**) is a powerful and versatile ASCII editor and shell for MS Windows with a strong predisposition towards the creation of [La]TeX documents...

WinEdt is widely used as a front-end for compilers and typesetting systems, such as TeX or HTML. WinEdt's highlighting schemes can be customized for different modes and its spell checking functionality supports multi-lingual setups, with dictionaries (word-lists) for many languages available for downloading from WinEdt's Community Site.

6.2.4 SmartDraw

SmartDraw (Sma) is a business graphics software from SmartDraw.com. SmartDraw is used to create business graphics such as flowcharts, organization charts, Gantt charts, timelines, floor plans and other diagrams.

6.2.5 NS2

Ns (NS206) is a discrete event simulator targeted at networking research. Ns provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks.

Ns was built in C++ and provides a simulation interface through OTcl, an object-oriented dialect of Tcl. The user describes a network topology by writing OTcl scripts, and then the main ns program simulates that topology with specified parameters. I have used version 2.29, the latest version of ns-2 is 2.31.

6.2.6 CorelDraw

Is a vector graphics editor developed and marketed by Corel Corporation of Ottawa, Canada. It is also the name of Corel's Graphics Suite. Its latest version, named X3, was released in January 2006.

Bibliography

[AG03] R. Ludwig A. Gurtov. Responding to spurious timeouts in tcp, 2003.

[AG05] R. Ludwig A. Gurtov. The eifel response algorithm for tcp, http://www.ietf.org/rfc/rfc4015.txt, 2005.

[All00a] M. Allman. A web server's view of the transport layer, 2000.

[All00b] M. Allman. A web server's view of the transport layer, 2000.

[Cla82] David D. Clark. Window and acknowledgement strategy in tcp, http://www.ietf.org/rfc/rfc813.txt, 1982.

[Flo00] Mahdavi... Floyd, ACIRI. An extension to the selective acknowledgement (sack) option for tcp, http://www.ietf.org/rfc/rfc2883.txt, 2000.

[Gnu] Gnuplot, http://www.gnuplot.info/.

[Gur02] Andrei Gurtov. Ns2 code by andrei gurtov, http://www.cs.helsinki.fi/u/gurtov/ns/index.html, 2002.

[IET] Ietf, http://www.ietf.org/.

[Jac90] V. Jacobson. Compressing tcp/ip headers for low-speed serial links, http://www.ietf.org/rfc/rfc1144.txt, 1990.

[Jac92] Borman Jacobson, Braden. Tcp extensions for high performance, http://www.ietf.org/rfc/rfc1323.txt, 1992.

[Lat] Latex, http://www.latex-project.org/ftp.html.

[Lud03] Meyer Ludwig. The eifel detection algorithm for tcp, http://www.ietf.org/rfc/rfc3522.txt, 2003.

[MA99] W. Stevens M. Allman, V. Paxson. Tcp congestion control, http://www.ietf.org/rfc/rfc2581.txt, 1999.

[MA00] V. Paxson M. Allman. Computing tcp's retransmission timer, http://www.ietf.org/rfc/rfc2988.txt, 2000.

[Mat96] Mahdavi Mathis, Floyd. Tcp selective acknowledgment options, http://www.ietf.org/rfc/rfc2018.txt, 1996.

[MD99] S. Pink M. Degermark, B. Nordgren. Ip header compression, http://www.ietf.org/rfc/rfc2507.txt, 1999.

[MM96] J. Mahdavi M. Mathis. Mahdavi, forward acknowledgment: Refining tcp congestion control, 1996.

[Nag84] J. Nagle. Congestion control in ip/tcp internetworks, http://www.ietf.org/rfc/rfc896.txt, 1984.

[NS03] D. Ely N. Spring, D. Wetherall. Robust explicit congestion notification (ecn),signaling with nonces, http://www.ietf.org/rfc/rfc3540.txt, 2003.

[NS206] Ns2, http://www.isi.edu/nsnam/ns/, 2006.

[oSC81] Information Sciences Institute University of Southern California. Transmission control protocol, http://www.ietf.org/rfc/rfc793.txt, 1981.

[PS05] M. Kojo P. Sarolahti. Forward rto-recovery (f-rto) , http://www.ietf.org/rfc/rfc4138.txt, 2005.

[Ram99] K. Ramakrishnan. A proposal to add explicit congestion notification (ecn) to ip, http://www.ietf.org/rfc/rfc2481.txt, 1999.

[Ram01] K. Ramakrishnan. The addition of explicit congestion notification (ecn) to ip, http://www.ietf.org/rfc/rfc3168.txt, 2001.

[rfc98] Recommendations on queue management and congestion avoidance in the internet, http://www.ietf.org/rfc/rfc2309.txt, 1998.

[Sma] Smartdraw, http://www.smartdraw.com/.

Index