"Week 4" starts here.

Other ~~sts~~ scheme:

Bipolar.

(Bipolar is not Polar)(!)

Also three levels:

$+X$ ($\rightarrow$ 1)

0 ($\equiv$ zero)

$-X$ ($\rightarrow$ 1)

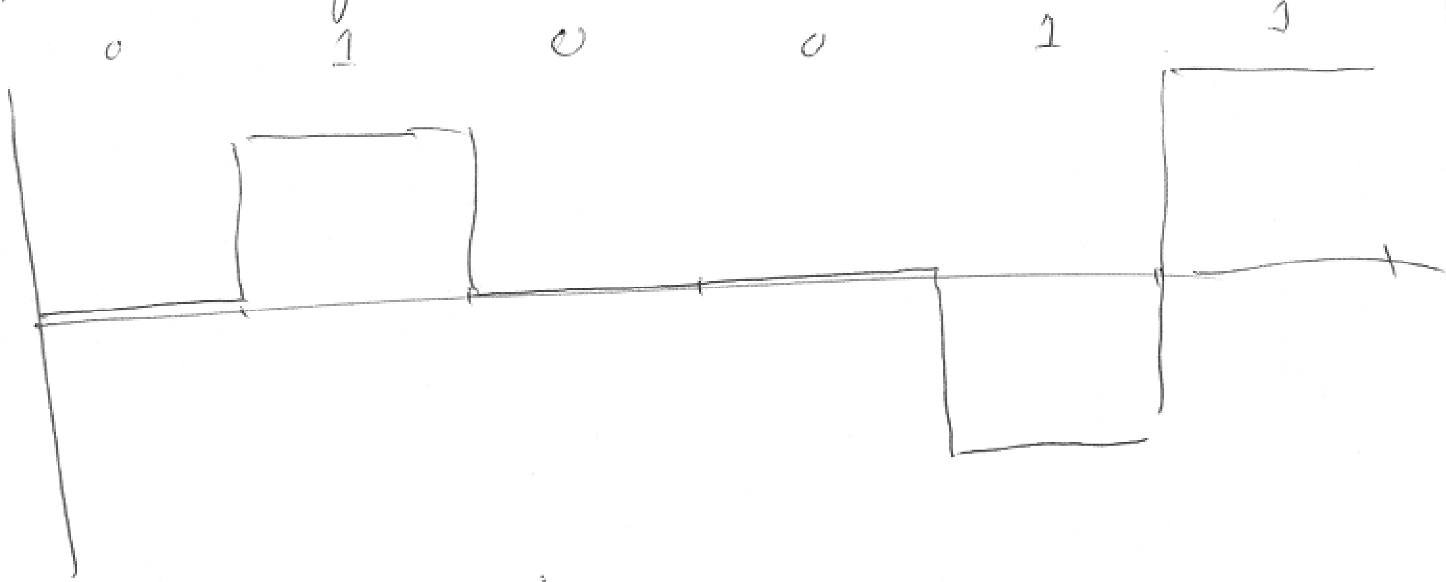
But 0 is not "intermediate state"

but used to represent zero.

1 is represented by $+X$ as well as $-X$.

Bipolar AMI.

Alternating Mark Inversion.



etc.

There are more versions of Bipolar.

Let's stop here.

To here 09/23/2003.

Next time:

$$(1-p)\sum_{k=1}^{\infty} k p^{k-1} = \frac{1}{1-p}$$

Start here 09/26/2003.

Last time we saw a number of ways to put a digital signal (bits) on a ~~coax~~ pair of conductors (co-axial cable, twisted wire).

The examples I gave all are what is called Baseband.

Since last week I found out: in 802.3, 10Base5, 10Base2, 1Base5 and 100BaseT all use Manchester encoding

There are other coding schemes: later.
(Broadband, so called PSK encoding).
(Also QAM etc.

But First something else!

Number representations. (I assume you know this).

(1) One's complement:

In 8-bit representation:

0000 0000	= zero =	+0
1111 1111	= -zero =	-0
0000 0001	=	1
1111 1110	=	-1
0000 0010	=	2
1111 1101	=	-2
⋮		
0111 1111	=	$1+2+4+8+\cdots+2^6 = 2^7-1 = 127$
1000 0000	=	-127

Similar for 16-bit representation
$\left(-(2^{15}-1) \text{ to } +(2^{15}-1)\right)$

32 bit $\left(-(2^{31}-1) \text{ to } +(2^{31}-1)\right)$

etc

X is "one's complement" of -X.

One's complement addition:

A: Add <u>three</u> bits:

three ~~all~~ zeros gives zero, no carry

2 zero, ~~one~~ 1 one gives 1, no carry

1 zero, 2 ones gives 0, carry 1

three ~~all~~ ones gives 1, carry 1.

B. Add <u>Two</u> numbers: use A.

Example:
10 bit representation!

~~-2^{9} to $+(2^{9}-1)$~~

$-(2^9-1)$ to $+(2^9-1)$

$$
\begin{array}{r}
1001001101 \\
0011011110 \\
\hline
1100101011
\end{array}
+
$$

C. If <u>last</u> (left most) bits generate a carry: add to first (right most) bits.

6 bit repr

$$
\begin{array}{r}
110101 \\
010001 \\
\hline
1\ \boxed{000110}
\end{array}
\rightarrow 000111
$$

$$1\ \boxed{000111} \rightarrow \boxed{001000}$$

Result <u>can</u> be Wrong

Result <u>can</u> be wrong. (8 bit repr.)

1000 0000	(−127)	
1000 0000	(−127)	+
0000 0001	(+1)	

should be −254 difference: $255 = 2^8 - 1$

~~01110~~

0100 0000	(+64)	
0100 0000	(+64	+
1000 0000	(−127)	

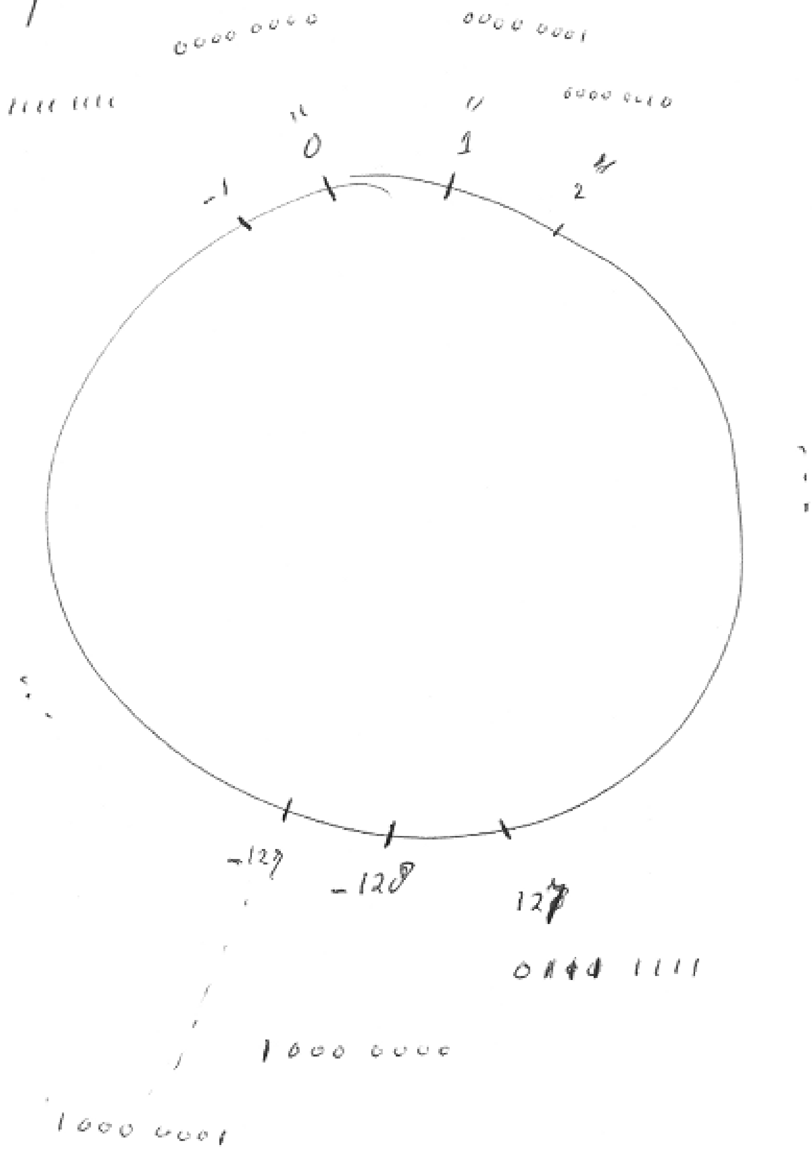
should be +128

difference: $255 = 2^8 - 1$

Two's complement.

8 bits : $2^8 = 256$ different numbers

<u>Think</u> of them in a circle:



Arithmetic modulo $2^8 = 256$

Digital → "decimal":

(1) Compute as if it is an unsigned int.

(2) If ~~value~~ $0 \le \text{value} \le 127$: that is the value

If $128 \le \text{value} \le 255$: subtract 256

(And similar for 16 bit representation, 32 bit representation, <u>etc</u>.).

Two's complement addition:
(Same $, only do <u>not</u> carry last bit.

Result <u>can</u> be wrong, but is right mod 2^n.
(8 bit representation: mod 2^8).

E.g.

$$
\begin{array}{rrl}
 & 0111\ 1111 & = +127 \\
 & 0000\ 0001 & = +1 \quad + \\
\hline
-128 & = 1000\ 0000 & ?\ 128
\end{array}
$$

* modulo 256 these are the same!

$$
\begin{array}{rrl}
 & 1000\ \ 0000 & -128 \\
 & 1000\ \ 0000 & -128 \quad + \\
\hline
0 = & 0000\ \ 0000 & \text{should be } -256.
\end{array}
$$

Mod 256 they are the same.

The Internet Checksum.

I'll do it for 16 bit representation.
You can do it for general n bit representation.

16 bits is done in the Internet.

Arbitrary sequence of bits: (say a packet, or a header, or...)

0100111 01010111

(1) Append zeros at end to get integer number times 16 bits.

(2) Divvy up in k "16 bit words".

|0100111 ...| ... | ... | ... | - - | .

(3) Add these 16 bit words using one's complement addition.

(4) Take one's complement of result.

(5) Append at end:
Now we have $(k+1)$ "16 bit words".

(6) Send these $(k+1)$ "16 bit words".
(with or without the extra zeros: agree first).

How does the receiver check?

(3) Check: (if necessary put zeros back in).

Add all (k+1) 16-bit words.

Result must be "−0" =

1111 1111 1111 1111

(No need to repeat the procedure).

"Proof": Do it yourself.
(Maybe: end of semester).

This checksum detects <u>all</u> errors in an <u>odd</u> number of bits, and <u>most</u> errors of an even number of bits. But not all. E.g.

$$\boxed{\begin{matrix} x & y & z & u \\ a & !y & b & c \end{matrix}} \rightarrow \boxed{\begin{matrix} x & !y & z & u \\ a & y & b & c \end{matrix}}$$

is <u>not</u> detected.

There are <u>better</u> error detecting methods. 802.3 uses "CRC", 32 bits (Cyclic ~~Red~~ Redundancy Check).
This is better.
Uses fancy math.
End of semester.

There are several CRCs.
Typically: Detect all errors in <u>odd</u> number of bits.
All errors localized to a stretch of 32 bits, ("error burst") (32? or 31?)
Most other errors.

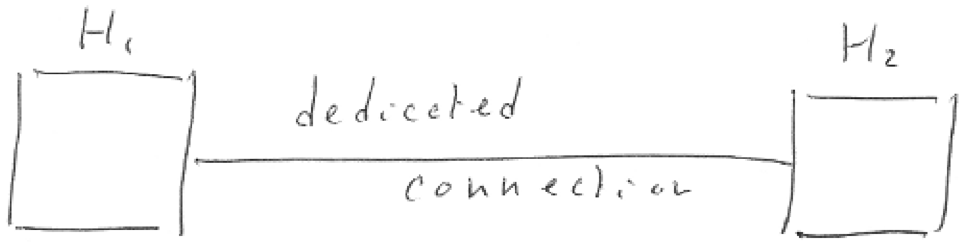
How are error detecting codes used?
At receiver:
IF the test Fails,
throw packet on the floor.

<u>possibly</u>: send "NACK".
(Usually <u>not</u> done).

ACK: acknowledgement.
"I got it".

NACK: negative acknowledgement.
"I did not get it".

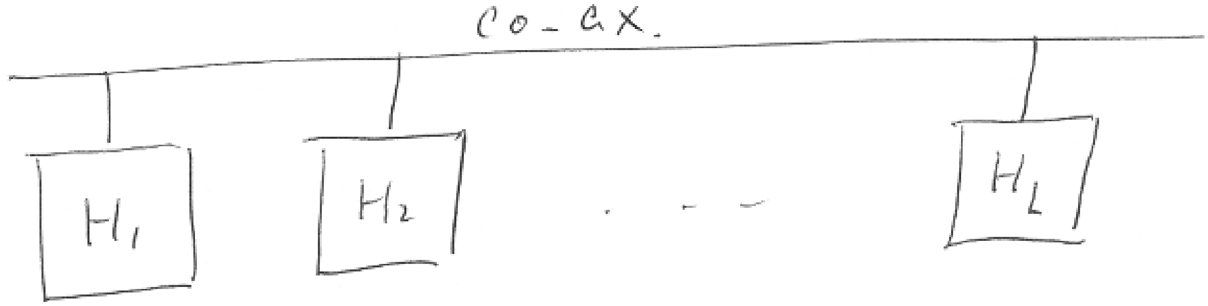
Why do <u>most</u> systems <u>not</u> use NACKs?

A. 

In case A, NACKs <u>can</u> be used.
"I got a packet, it failed $\left\{\begin{array}{l}\text{CRC}\\ \text{Checksum}\end{array}\right.$".

old Fashioned ethernet.

B:



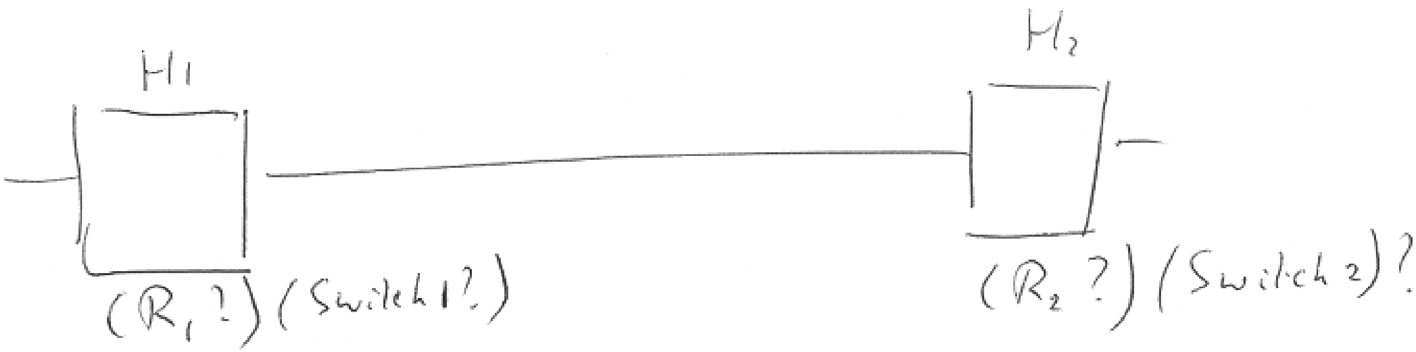
Host i receives packet.
Checks CRC.
Fails.
"Maybe the error is in the ~~source~~ dest address. Is this packet really for me?".
"Maybe the error is in the ~~#~~ source address. If I send a NACK, maybe I will confuse somebody".
So: in case B, drop packet,
do not send NACK.

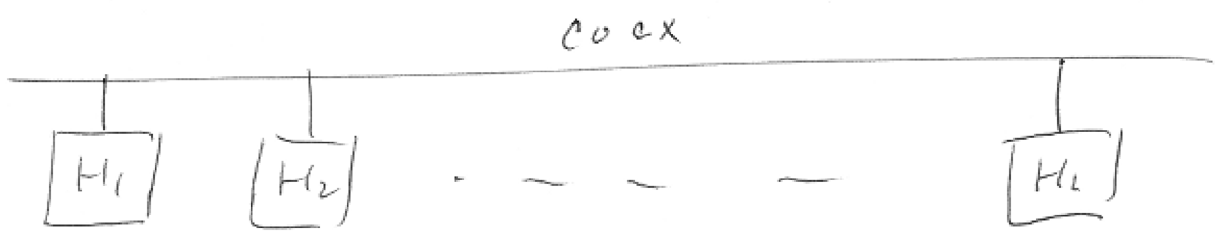
There are systems that use NACKs.
X.25 uses "NACK", there called "Reject".

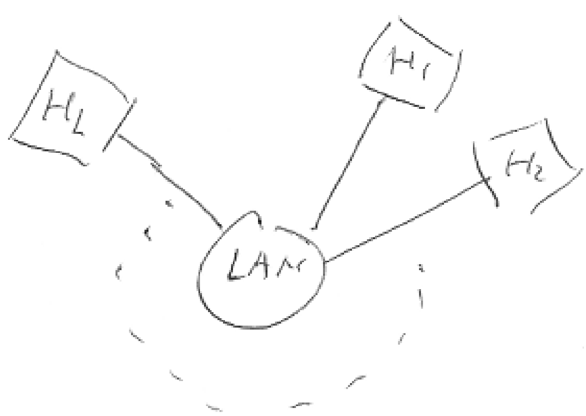
Elementary Data Link Protocols.

Tanenbaum p 200 etc.

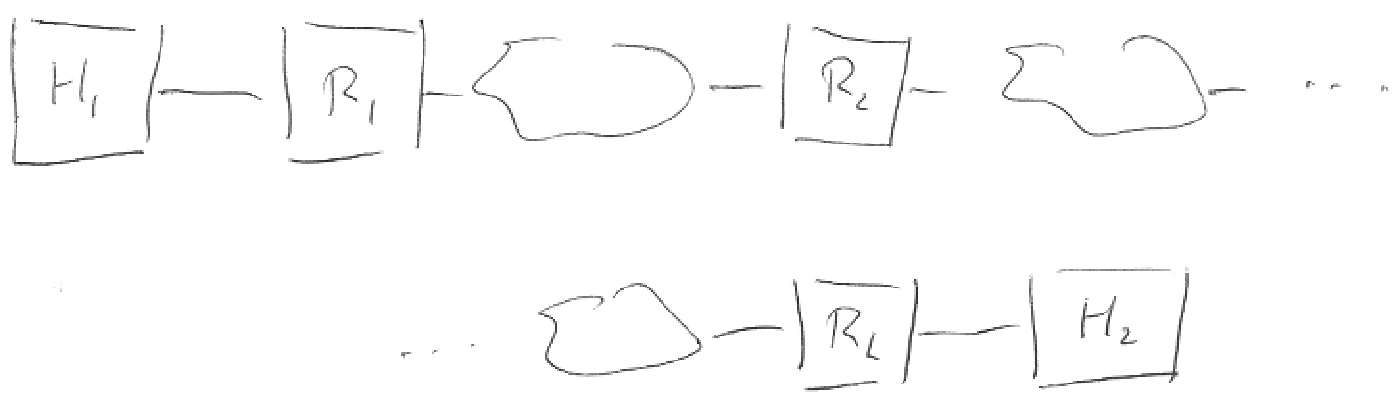
This story applies to
Data Link Protocols (Layer 2)



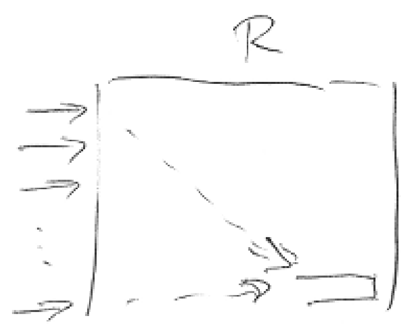




But it <u>also</u> applies to "Transport Protocols" (Layer 4)



Routers may drop packets due to congestion.



("Focussed overload").

Issue:

Frames (packets) can get damaged.
(CRC or Checksum Fails).

Frames (packets) can be <u>dropped</u>.
(disappear into nowhere).
e.g., congestion.

How do we build a
<u>Reliable</u> system?

ACKs, NACKs.

First: assume ~~#~~ ACKs as well as NACKs are used.

Simple system: "stop and wait".

Bad implementation:

At source:

1. Source sends frame (or packet).
2. Waits for ACK or NACK.
3. if ~~Nac~~ NACK: Retransmit, back to 2

   if ACK: back to 1

At destination:

1. Wait for frame.
2. if good: send ACK

   if bad: send NACK.

Why is this implementation

BAD ?

IF even a packet disappears
without trace:
both ~~the~~ sides wait Forever.
(dead-lock).

Solution: T.O.
Time - Out

Simple "stop and wait",

<u>Improved</u> implementation.
(still not good!).

1. At time t, source sends Frame.
Sets time-out for $t + TO$.
(TO: parameter to be chosen).

2. Source waits for ACK, NACK, or time-out, whichever happens first.

3. At time-out: (at time τ)
Re-transmit Packet. (Frame)
Set time-out for $\tau + TO'$
(Not necessarily $TO' = TO$). Goto (2)

4. At NACK: (at time τ)
Re-transmit packet (Frame)
Set time-out for $\tau + TO''$ Goto (2)

5. IF ACK: (at time τ)
Transmit new packet.
Set time-out for $\tau + TO$ "Goto 1".
Goto 2.

At dest:

(1) Wait for frame.

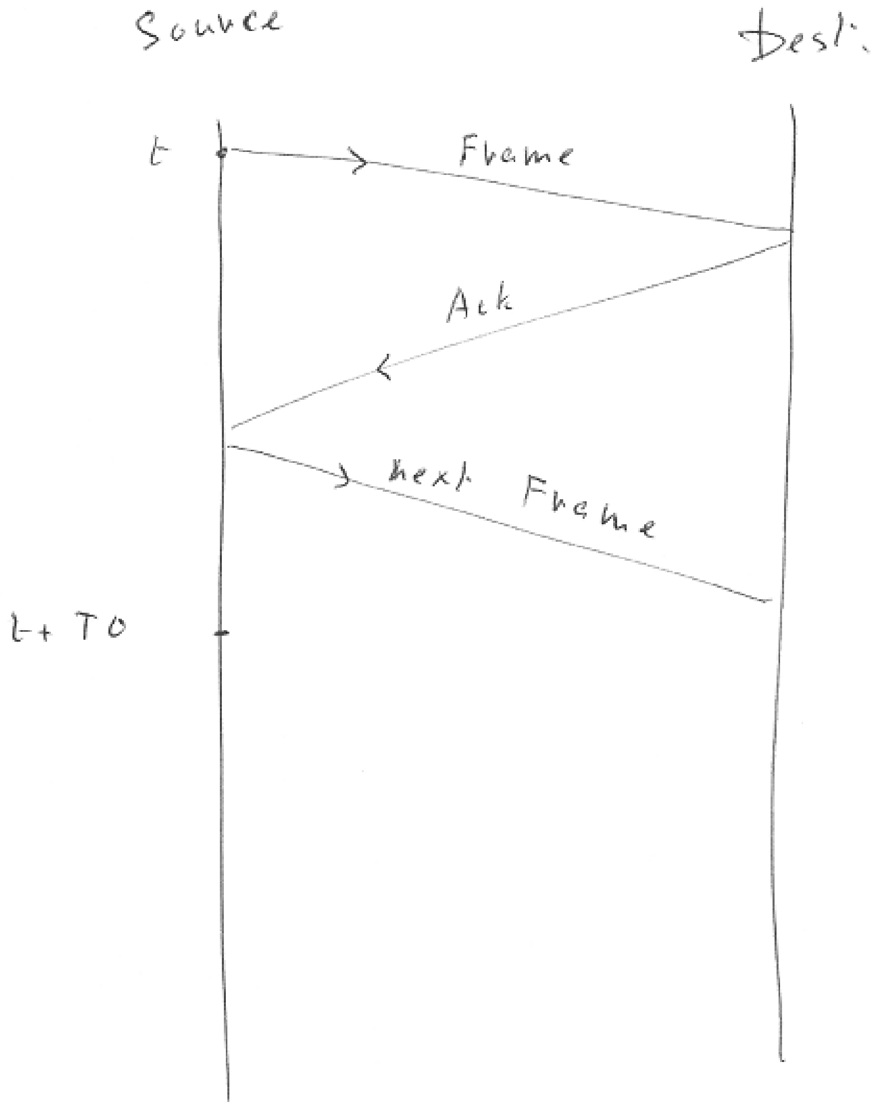
(2) if good: send ~~Ack~~ ACK
if bad: send NACK
Goto (1)

Is <u>this</u> system sound?

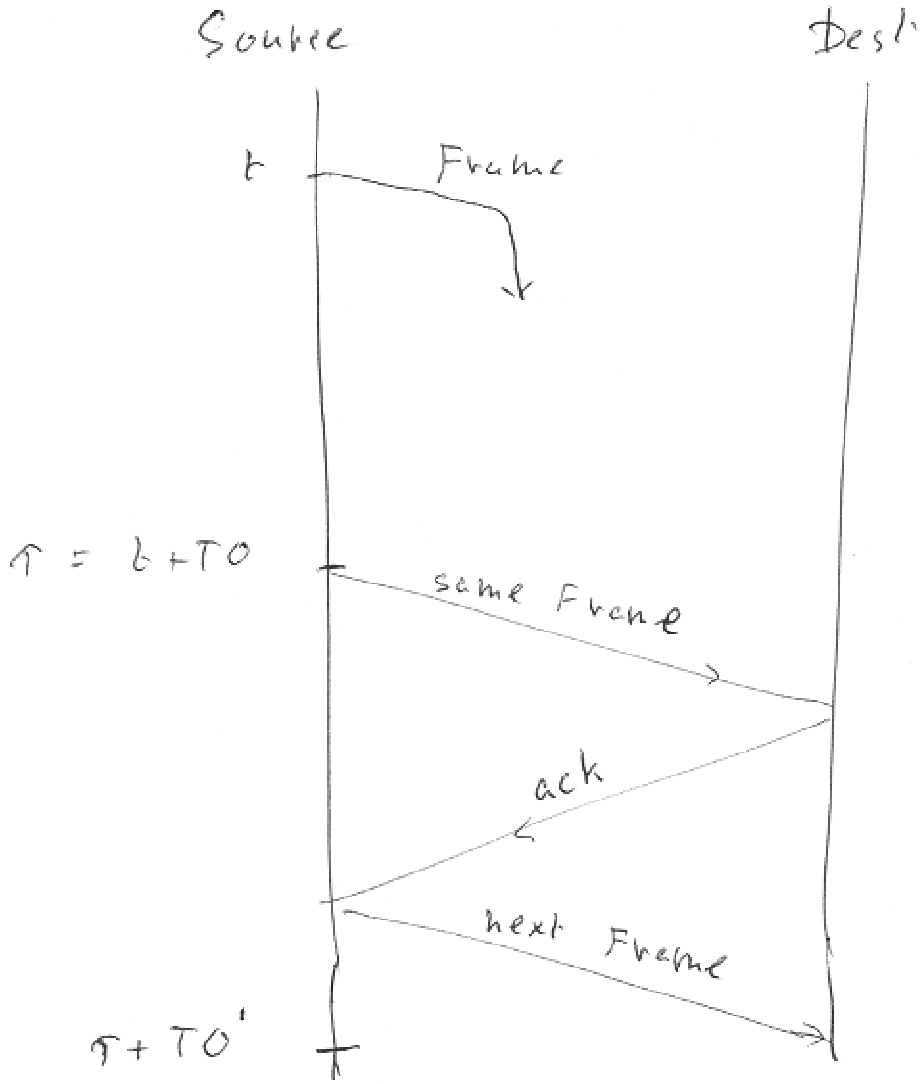
<u>No</u>!

At least: not if a "good" packet can get delayed.

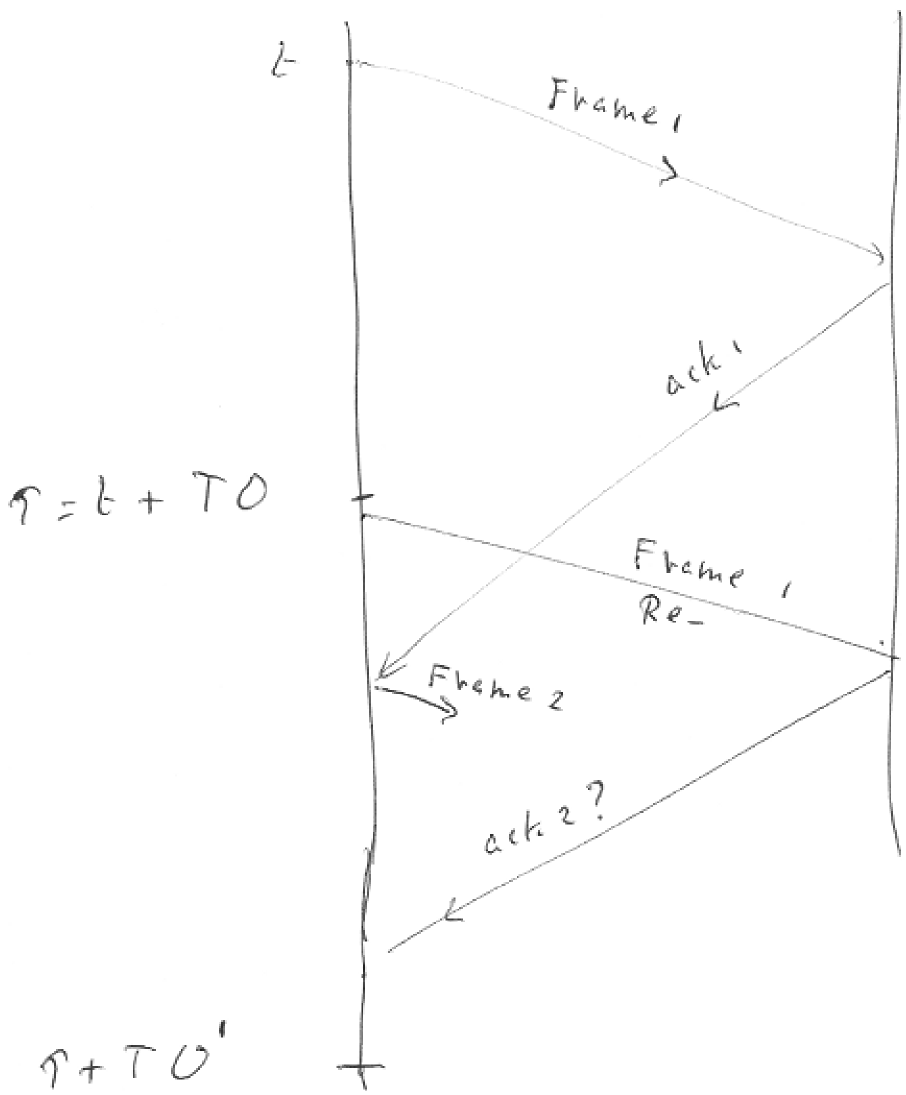
94



Ok!



Ok!



Delayed Data Frame (or Ack).

BAD.

Not OK.

The destination gets Frame 1 twice but thinks it got frames 1 and 2 : Error.

E.g. Email: a few tens or hundreds of characters will be repeated.

Data File: worse!

Solution :

Data Frames : number the Frames, (sequence number).
Include the sequence number in the Frame.

Acknowledgements:
Include sequence number of Frame being acknowledged.

NACK: ~~better not include sequence number: could be wrong!~~