Chapter 3 - The Datalink Layer

Possible Service Classes:

Unacked connectionless or connection oriented.
- Undetected loss of packets can occur

Acked connectionless (stateless)
- Possible duplication of packets can occur

Acked connection oriented (stateful)
- Virtual error free channel
- How to recover from loss of state??

Datalink functions:

- Framing
    - Identify packet start and
    - Packet end
- Error Control
    - Error correction or
    - Error detection
- Flow Control
    - Fast sender vs. slow receiver
- Link management (for connection oriented link layers)
    - Initialization
    - Termination
    - Recovery
    - Multi-station management

Framing

(At least) 4 basic approaches have been used

Physical layer ``violations''

Use special physical layer encodings.
Used in Token Ring LANs.
Disadvantage
    Requires a physical layer encoding with unused capacity.

Character count

Packet
    Starts with a character count
    Contains embedded char count.

Disadvantage
    Character count corruption.

Obsolete: formerly used in DEC's DDCMP

Start and end delimiter bytes (or byte pairs) with character stuffing.

DLE STX ....the..data .... DLE ETX
0x12 0x02....................................... 0x12 0x02

Problem

What if "the data" contains DLE ETX.

Solution

Sender:

Whenever DLE occurs in data duplicate it.

Receiver:

Convert all DLE pairs to single DLE.

Disadvantage

Code ASCII / EBCDIC dependencies

8 bit byte dependencies

Obsolete: Formerly used in IBM's BISYNCH

Bit stuffing

Special flag 0x01111110 indicates start of packet *and* end of packet.

Sender USART IC(universal synchronous and asynchronous receiver/transmitter) automagically generates the flag when directed by device driver to transmit .

During data transmission any time 5 1's follow a zero USART injects a 0.

Receiver USART eats any zero that follows 5 1's.

True data stream:	0101111011111011111011111110
Stuffed stream:	010111101111100111110101111110110

This method is used in SDLC, HDLC, etc.

Disadvantages:

Corrupted start flag

Loss of packet and end flag assumed to be start flag of *next* packet

Corrupted end flag

Possible super long packet and/or start of next packet confused with end of current packet.

Solution: Send continuous stream of 1's in the idle state

Loss of start flags leads to presence 0's where 1's should be

Loss of end flag lead to absence of 0's where required.

Error Control

Error correction
Error detection

Error Correcting Codes

Consider *fixed size* packets called *code words.*
n = number of bits in a codeword

Given two n-bit code words A and B the Hamming Distance between A and B is

H(A, B) = # number of bit position in which codes don't match
A = 01101110
B = 11101001

H(A, B) = 4

The Hamming distance of a code is Min {H(A, B): A, B in the code}

For an *n* bit codeword there are 2^n possible codewords.

If all of these may be transmitted, then the Hamming distance of the code is 1.

If the Hamming distance of a code is 1, then neither error detection or correction is possible.

In order to use Hamming codes to correct or detect errors it is necessary to partition the code set into *legal* and *illegal* code words.

To *detect* 1 bit errors a code must have a hamming distance of at least 2

To *correct* 1 bit errors a code must have a hamming distance of at least 3

Example

Consider a 3 bit code which has 8 codewords:

We can create a distance 3 code by identifying exactly one pair of codewords which are distance 3 apart as legal.

000	100	Legal:	000, 111
001	101	xor	001, 110
010	110	xor	010, 101, etc.
011	111		

With each legal code word you must associate the three illegal codewords that are distance 1 from the legal word

000 ---- 001, 010, 100
111 --- 110, 101, 011

Thus if you receive a codeword with a single bit error you can determine which legal codeword was intended. (Its the illegal word at distance 1).

If you receive a codeword with a multiple bit error *you will not be able to detect any error* and *you will accept the wrong message*.

How many bits in general are required for ECC

n	= bits in codeword
m	= data bits
r	= ECC bits

n	$= m + r$
2^m	= Number of legal messages.

For each legal message $(n + 1)$ codewords must be reserved

1 for the message itself

n for each way a single bit might be inverted.

2^n	= Total number of codewords

Thus

$$(n + 1)\, 2^m <= 2^n$$

$$(n + 1) \quad <= 2^{(n-m)} = 2^r$$

$$r >= \log_2(n + 1)$$

Hamming showed that this bound could always be achieved if $n + 1$ was a power of 2.

For $n = 3$, $m = 1$ and $r = 2$ as seen above

For $n = 7$, $m = 4$ and $r = 3$.

How are the legal and illegal codewords identified?

Solution

Power of 2 bits are the check bits

They act as parity bits for the set of bits in whose binary representation they appear with value 1.

Example: For a 15 bit codeword

0	0000	4	0100	8	1000	12	1100
1	0001	5	0101	9	1001	13	1101
2	0010	6	0110	10	1010	14	1110
3	0011	7	0111	11	1011	15	1111

Check bit	Checks
1	1, 3, 5, 7, 9, 11, 13, 15
2	2, 3, 6, 7, 10, 11, 14, 15
4	4, 5, 6 , 7, 12, 13, 14, 15
8	8, 9, 10, 11, 12, 13, 14, 15

Computing the check bits

For a 7 bit codeword. There are 4 data bits and 3 ECC bits. Suppose the data bits are 1001 and that the number of 1 bits in a parity check group is required to be odd.

Val			1		0	1	1
	___	___	___	___	___	___	___
POS	1	2	3	4	5	6	7

Check bit	Data bits	Position	Check bit value
1	1 0 1	3 5 7	1
2	1 1 1	3 6 7	0
3	0 1 1	5 6 7	1

Transmitted:

Val	1	0	1	1	0	0	1
	___	___	___	___	___	___	___
POS	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Error correction algorithm:

The index of the bit in error is the sum of the indices of the parity bits in error.
Suppose the following code word is received.

Val	1	0	1	1	0	0	1
POS	1	2	3	4	5	6	7

Check bit	Grp bits	Position	Result
1	1 1 0 1	1 3 5 7	OK
2	0 1 0 1	2 3 6 7	BAD
4	1 0 0 1	4 5 6 7	BAD

Hence bit 6 (2 + 4) is the bit in error.

The size of each parity checked group is (n + 1) / 2
The intersection of any two parity check groups has size (n + 1) / 4
The intersection of any three paritry check groups has size (n + 1) / 8
:
The intersection of all the parity check groups is the last bit in the code word and thus has size 1.

Detecting error bursts:

An s bit burst means that the distance from the 1st bad bit in the codeword to the last bad bit in the codeword is s (and NOT that there are s bad bits.)

To correct s bit bursts lay out the message as an array of s rows and t columns

Apply the usual Hamming code to each *row*

$(\log_2(t + 1)$ ECC bits per row ... for a total of $s \log_2(t + 1)$ ECC bits

Transmit message in column order...

Any error burst of size s or less will have a bad bit in at most *one* column

==> the row containing the error can be corrected

Error Detecting Codes:

Parity:

Force the number of 1 bits in the codeword to be odd or even.
Can only detect odd number of bits in error
All even number of bit errors get through.

Checksum:

Treat the code word as a series of bytes, short words, or long words

Sender
- Add up the words in the message
- Append the sum to the message.

Receiver
- Add up the words as message is received
- Compare computed sum with received sum.

Advantage
- Fast and efficient
- Detects bursts up to word length

Disadvantage
- Not particularly robust
- Detects all odd # bits in error
- Detects some even # bits in error
- Even # bit errors in the same bit position get through undetected.

Cyclic redundancy codes or polynomial codes:

A (not necessarily fixed length) message is viewed as the coefficients of a polynomial over the finite field {0, 1} (Z_2 = the integers *mod* 2)

An *n* bit code word corresponds to a degree *n-1* polynomial in GF(2^n)

$1010011 = X^6 + X^4 + X + 1$

Addition and subtraction in this domain are both done via XOR

```
X^6 +        X^4 +              + X + 1
       X^5 +  X^4 + X^3 -  X^2 + X - 1
--------------------------------------------------
X^6 + X^5          + X^3 +  X^2
```

$$\begin{array}{l} X^6 + X^4 + X + 1 \\ \quad X^5 + X^4 + X^3 - X^2 + X - 1 \\ \hline X^6 + X^5 + X^3 + X^2 \end{array}$$

Mathematical framework -

The Reed-Solomon codes are built upon the Galois field GF(2^m) in which the polynomials constituted a field with multiplication was carried out modulo an irreducible generator polynomial g(x).

The cyclic codes are built on the ring of polynomials over the field Z_2. In this domain, polynomial division is defined, but polynomials do not have multiplicative inverses.

Computation of CRCs

Let M(X) = a message of *m* bits (M(X) has degree m - 1).

To compute an *r* bit CRC, a *generator* polynomial G(X) having degree *r* is used. Since the generator has *r + 1* terms the remainder will have at most *r* terms.

Let X^r * M(X) = message polynomial with r low order zero bits appended.

Let R(X) = MOD(X^r * M(X), G(X)) (has degree < than G(X))

Let Q(X) = Quotient(X^r * M(X), G(X))

Then $X^r * M(X) = G(X) * Q(X) + R(X)$

(multiplying the divisor times the quotient and adding back the remainder yields the original dividend).

or

$X^r * M(X) - R(X) = X^r * M(X) + R(X) = G(X) * Q(X)$

(if you subtract the remainder from the original dividend you obtain a value that is evenly divisible by the original divisor G(X).)

and

MOD(X^r * M(X) + R(X), G(X)) = 0

Implementation of CRC

The sender transmits $X^r * M(X) + R(X)$ which should be what the receiver receives.

The receiver *divides* the incoming message by $G(X)$ and if the remainder is 0 the message is accepted.

Questions:

What kinds of errors can get through
What kinds of $G(X)$ make good generators.
How does one do polynomial long division in real time.

Characterization of errors not detected by CRC's

Error = Some bits are inverted during transmission
Bit inversion is equivalent to XORing a bit with a 1 bit.
Let E(X) be a polynomial that has 1 bits in the positions in which the errors occurred.

Receiver actually receives $X^r * M(X) + R(X) + E(X) = T(X) + E(X)$

Def: A polynomial G(X) is said to *divide* a polynomial P(X) if and only if there exists a polynomial Q(X) such that G(X) Q(X) = P(X). (This definition is where the term *zero divisor* originates. A polynomial is a zero divisor if it satisfies this equation when P(X) = 0.)

Theorem: The message will be accepted in error <==> G(X) evenly divides E(X).
Proof:
===> Suppose message is accepted in error

Then G(X) div E(X) + T(X) and
there exists W(X) such that G(X) W(X) = E(X) + T(X) (1)
recall T(X) is constructed so that G(X) div T(X)
thus there exists P(X) such that G(X)P(X) = T(X)
replacing T(X) in (1) with G(X)P(X) yields:
G(X)W(X) - G(X)P(X) = E(X)
G(X)(W(X) - P(X)) = E(X)
or G(X) div E(X)

<=== Suppose G(X) div E(X)

Then there exists S(X) such that G(X) S(X) = E(X)
But G(X) also divides T(X) by definition..
Thus there exists P(X) such that G(X)P(X) = T(X)
Adding them up gives:
G(X)S(X) + G(X)P(X) = E(X) + T(X)
or
G(X)(S(X) + P(X)) = E(X) + T(X) ==>
G(X) divides E(X) + T(X) and so the packet is accepted in error.

Therefore, *a good generator should evenly divide as few polynomials as possible.*

Characteristics of generators and relation to errors not caught:

Single bit errors

$E(X) = X^k$ where k is the bit position of the error.

If G(X) has more than 1 term G(X) can't divide E(X) and the error will be caught.

Double bit errors

$E(X) = X^k + X^m = X^m \; (X^{k - m} + 1)$

If G(X) has more than one term it can't divide the first factor.

If G(X) doesn't divide $X^n + 1$ for n less than the message length then it can't divide the second factor either.

Note that this argument (taken from the Tanenbaum book) appears to be saying that if
E(x) = U(x) V(x) and G(x) divides neither U(x) nor V(x) then
G(x) doesn't divide E(x) either.

This is not true of integers
10 div 70 and 70 = 2 x 35 but 10 doesn't div 2 or 35

It is also not true for for polynomials

$G(x) = (X^2 + X + 1)(X + 1) = X^3 + 1$

$E(x) = (X^3 + 1)X = X^4 + X = (X^3 + X^2 + X)(X + 1)$

Then G(x) div E(x) but G(X) does not divide either factor

Even though not true in the general case the result is true in the specific case in which
$E(x) = X^m \quad (X^k + 1)$

Suppose $G(x)$ not div $(X^k + 1)$

Then $(X^k + 1) = Q(x)\ G(x) + R(x)$ where $R(x)$ ne 0 and
$\deg(R(x)) < \deg(G(x)$

Now suppose $G(x)$ div $E(x) = X^m (Q(x)\ G(x) + R(x))$

Since $G(x)$ div $X^m (Q(x)\ G(x)$, $G(x)$ must also div $X^m R(x)$ by theorem on the last page regarding $G(x)$ div $T(x) + E(x)$

Thus there must exist $P(x)$ such that $G(x)\ P(x) = X^m R(x)$

$X^m R(x) = X^{m+j-1} + \ldots + X^m$ where $j = \deg(R(x)) < r = \deg(G(x))$

Since $G(x)$ has a constant term, the smallest term in $P(x)$ must be X^m
Since $G(x)$ has degree r the product $G(x)\ P(x)$ must have an X^{m+r} term.
This contradicts $X^m R(x) = X^{m+j-1} + \ldots + X^m \ldots$ w/ $j < r$ Q.E.D.

Odd number of bits in error 3, 5, 7, 9, ... 999, etc.

If G(X) has x + 1 as a factor the G(X) can't divide E(X).

Proof:

Assertion:

If G(X) divides E(X) then any factor of G(X) must also divide E(X).

Proof of assertion:

Suppose G(X) = F(X) H(X).

G(X) divides E(X) <=> there is a Q(X) such that G(X)Q(X) = E(X)

or

F(X)H(X)Q(X) = E(X)

Thus F(X) divides E(X) with quotient H(X)Q(X) and H(X) divides E(X) with quotient F(X)Q(X)

Now suppose X + 1 divides E(X).

This implies there exists Z(X) such that (X + 1) Z(X) = E(X)

Therefore

E(1) = (1 + 1) * Z(1) = 0

But E(X) has an odd number of terms so E(1) = 1 ==> <==

Error burst.. Suppose errors are limited to a k bit burst:

$E(X) = X^m (X^{k - 1} + ... + 1)$ where m is the displacement of the last bit of the burst from the end of the message.

If G(X) has a constant term it can't divide X^m

If (k - 1) < degree(G), G(X) can't divide the other piece either.

Here we have the same situation where we appear to be saying that if G(x) doesn't divide either factor then it cannot divide the product. As before, for this special case if G(X) divides neither factor then it also doesn't divide the product.

Random multibit errors:

P(error gets through) = $1 / 2^r$

Computing CRC's by long division - G(X) = x^4 + x + 1

```
                   1 1 0 0 0 0 1 0 1 0
           _______________________________
1 0 0 1 1 | 1 1 0 1 0 1 1 0 1 1 0 0 0 0
            1 0 0 1 1
           ---------
              1 0 0 1 1
              1 0 0 1 1
              ----------
                0 0 0 0 1
                0 0 0 0 0
                 ---------
                  0 0 0 1 0
                  0 0 0 0 0
                   ---------
                    0 0 1 0 1
                    0 0 0 0 0
                     --------
                      0 1 0 1 1
                      0 0 0 0 0
                       ---------
                        1 0 1 1 0
                        1 0 0 1 1
                         ---------
                          0 1 0 1 0
                          0 0 0 0 0
                           ---------
                            1 0 1 0 0
                            1 0 0 1 1
                             ----------
                              0 1 1 1 0
                              0 0 0 0 0
                               -------
                                1 1 1 0
```

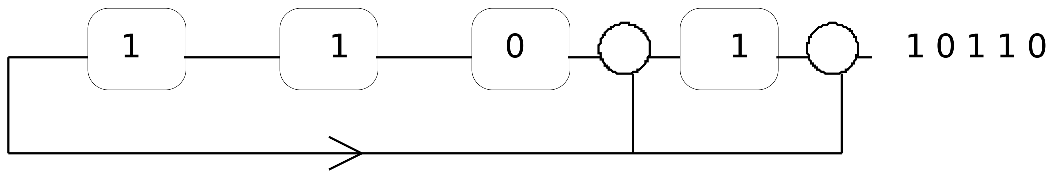
This looks and is very computationally difficult.

But it provides motivation for the shift register approach

- At each step
  - Shift partial dividend left
  - If high order bit was 1 XOR partial dividend with generator

A shift register implementation of long division

For the generator $X^4 + X + 1$ used in the book the following shift register is appropriate(circles represent XOR gates.

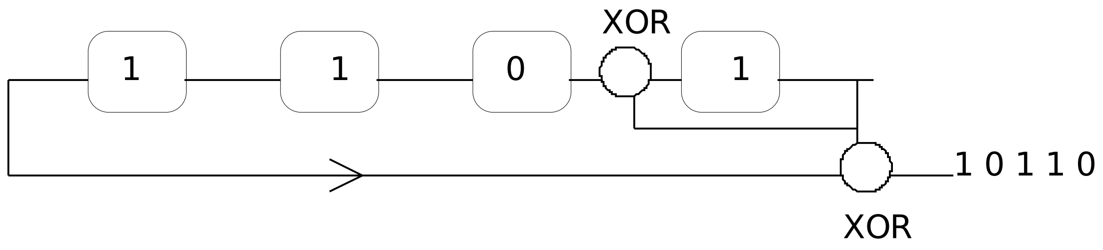


Division done by the shift register method (left column shows shift register contents).

```
1 1 0 1     (Next input bit)
                 0
1 0 0 1
                 1
0 0 0 0
                 1
0 0 0 1
                 0
0 0 1 0
                 1
0 1 0 1
                 1
1 0 1 1
                 0
0 1 0 1
                 0
1 0 1 0
                 0
0 1 1 1
                 0
1 1 1 0
```

This is fast and easy to build... but you don't get the desired answer out until... 4 bit times *after* the actual end of the message!

The fix is the pre-multiplying XOR shift register



Division done by the pre-multiplying shift register..

Note that in this version we can't simply start with the first 4 bits preloaded..

Also note that only the final answers agree.. none of the intermediate steps.

```
0 0 0 0
         1
0 0 1 1
         1
0 1 0 1
         0
1 0 1 0
         1
0 1 0 0
         0
1 0 0 0
         1
0 0 0 0
         1
0 0 1 1
         0
0 1 1 0
         1
1 1 1 1
         1
1 1 1 0
```

General shift register characteristics

number of stages = degree of the generator
right most stage corresponds to constant term in generator
left most stage corresponds to degree of generator - 1 term.
stages in the shift reg have an XOR to their right <==> corresponding term in the generator is non zero.

Real world CRC's

CRC-16	=	$X^{16} + X^{15} + X^2 + 1$
CRC-CCITT	=	$X^{16} + X^{12} + X^5 + 1$

Problems:

How is the shift register initialized ?
Which order are the bits of a byte injected ?
In what byte / bit order is the CRC itself transmitted ?

Sliding window protocols (a.k.a. Automatic Repeat Request (ARQ)):

Objectives:

Error control - make unreliable physical layer appear reliable

Flow control - prevent fast sender from overwhelming slow receiver

Evolution:

Protocol 1:

Sender sends at will

No acks used at all

Possible problems

Any error will be undetected

Overrun may occur if packets are sent faster than they can be consumed

Protocol 2:

Sender sends a packet and waits for an ACK or a timeout.

(0) => Frame number is not actually transmitted!

```
F(0) ------------------> Received correctly

      <-----------------   ACK
           (lost)
timeout
F(0) -------------------> Duplicated
```

Protocol fails on lost ACK.

Protocol 3:

Possible fix.. Explicitly send a 1 bit sequence #

```
F-0(a) --------------------->    Received correctly

          <------------------       ACK
              (lost)
timeout

F-0(a)  ------------------>       Discarded (since F-1 was expected)

       <-----------------          ACK

F-1(a)  ------------------>       Accepted

                            / -------- ACK (delayed)
timeout                    /
                          /
F-1(a) ------------------->  Discarded duplicate        Thought to be for the second
                       /                                copy of F-1(a) but actually
                      / ◄                / ---ACK (delayed)    for the first copy.
             <----------                /
                                       /
F-0(b) -------X---------->            /         Thought to be for F-0(b) but
              (lost )                /          actually for the second copy
                <-------------- / ◄                    of F-1(a).

F-1(b) ---------------->  Discarded duplicate  ◄  Thought to be yet another
                                                     duplicate of F-1(a)
       <-------------------    ACK
                      ▲
F-0(c) ----------------->  Accepted (oops -- protocol failed)
              ▲
   Thought to be F-0(b)                     Thought to be for F-1(b) but
                                            receiver is still looking for F-
                                                        0(b)
```

What if bad frames aren't acked??? The situation is even worse. The protocol instantly deadlocks on a lost ACK.

```
F-0(a) --------------------->    Received correctly


        <-----------------       ACK
           (lost)
timeout

F-0(a) ---------------------->    Discarded (but not ACKED)

timeout

F-0(a) ---------------------->    Discarded (but not ACKED)

timeout

etc.
```

Binary synchronous (Bisync) Protocol

Frames not numbered but ACK's are. ACK's conventionally specify the sequence number of the *next frame expected* not the last frame received correctly. This is done so that the initial state can be SEQ=0 and ACK=0

Sender

- sends and waits for an ACK or a timeout
- sends ENQ on timeout

Receiver

- accepts any with correct CRC
- sends NAK on receipt of damaged packet
- replies to ENQ with last ACK

```
F-(0) --------------------->   Recvd OK

    <---------------------    ACK - 1
              (lost)
timeout
ENQ -------------------->

      <-------------------    ACK - 1

F-(1)   -------------------->
             (lost)

ENQ  ------------------->

       <-------------------  ACK - 1

F-(1)  --------------------->    OK

     <--------------------- ACK - 0
```

Disadvantages of bisynch:

- Needlessly complex
- *No pipelining possible.*

True Sliding window protocols

Simple, full duplex protocols with pipelining.

Frames carry both

Seq a frame sequence number and

Ack which is *the next sequence number expected from the other end*.

If I receive frame zero, I send ACK=1.

Standalone ACKs carry only

Ack which is the next sequence number expected from the other end.

State variables required at *both ends*:

For sending

nxtsend - Frame number of the next frame to send

lstack - Last ACK received correctly. (Sequence number for oldest unacked frame if one exists. Otherwise *nxtsend* == *lstack*. It is initialized to 0.

sndwnd - Maximum number of unacked frames -- limited by # bits in seq #.

For receiving

nxtrecv - Frame number for next frame expected from the other end.

rcvwnd - The number of out of order frames the receiver can buffer – also limited by bits in the sequence number.

For link layer protocols values in blue are dynamic while those in black are parameters of the protocol that don't change during operation.

Sequence number space (simplex model)

```
   sent and ack rcvd     sent not acked      can send      can't send yet
  |----------------------|----------------------|---------------|---------------->
  0                  lstack                 nxtsend    lstack + sndwnd


     rcvd and acked              can recv        not yet eligible
  |-------------------------------|---------------|------------------->
  0                             nxtrcv     nxtrcv + rcvwnd
```

Send logic:

Send frame using *nxtsend* as a *seq* number
Increment *nxtsend*
Timestamp the buffer
When *nxtsend* - *lstack* == *sndwnd*, sender must stop and wait for ACK.

Receive logic:

```
/* Incoming frame processing – frames with bad CRC are disregarded */

If frameseq == nxtrcv
        increment nxtrcv and forward frame to network layer

Send ACK [nxtrecv]   (regardless of frameseq)

/* Ack processing */

/* Performed only upon undamaged frames */
        if (lstack <= ack in frame <= nxtsnd)  // valid ack ??
        {
            /*  Test for non-duplicate ACK */
                while (lstack != ack in frame)
                {
                        free buffer[lstack]
                        lstack += 1
                }
        }
```

Protocol 3 failure is eliminated!

```
F-0-0(a) ------------------>    Received correctly

        <------------------       A-1
            (lost)
timeout

F-0-0(a) ------------------>        Discarded (since F-1 was expected)

       <---------------------     A-1

F-1-0(a) ----------------->       Accepted

                           / --------    A-0 (delayed)
timeout                   /
                         /
F-1-0(a)---------------/-->     Discarded duplicate
                      /
                     /       /------      A-0 (delayed)
           <--------/       /
                           /
F-0-0(b)--------- -------/---->
             (lost)       /
                         /
             <----------/

F-0-0(b) ----------------->  Accepted
```

Frame 0 was retransmitted since ACK says that 0 is the next frame expected.

Summary

The sliding window protocols:

- Provide a foolproof mechanism for identifying lost packets and initiating retransmission.
- Rely upon slow delivery of acks to prevent a fast sender from overwhelming slow receiver.
- The long-term ack delivery rate governs the rate at which the sender can send.
- If the receiver can only consume 10 packets per second, it must generate only 10 acks per second.

A proper flow control mechanism should be decoupled from the ACK mechanism

Window Size Considerations

Sender window (*sndwnd*) = {transmitted but unacknowledged frames}

Receiver window (*rcvwnd*) = {frames that the receiver is willing to accept and, if out of order, buffer for later delivery.. therefore a receiver window of size >= 1 is useful only when errors occur.}

Relationship between sizes is:

1 <= *rcvwnd* <= *sndwnd*

If *sndwnd = 1* then the sender can never have more than 1 outstanding unacked packet so there is no way that the receiver can receive out of order frames! This protocol is called *stop and wait*.

If *sndwnd = 15* then the receiver might wish to buffer 14 out of order frames, but it also might not want to buffer any.

Sliding Window Protocol Categories:

Stop and Wait

Sender sends 1 packet and waits until ACK is received before sending another. *sndwnd* = *rcvwnd* = 1.

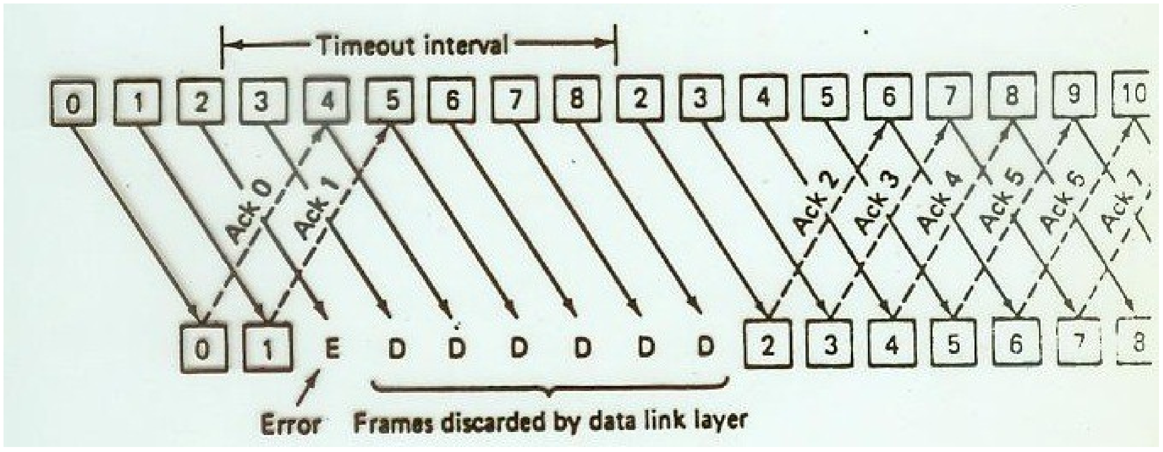
For the sake of efficiency Stop and wait should *not* be used *unless round trip latency << packet transmission time* as would be the case on a modest speed small LAN

Go Back N

- Sender may continuously send while the number of unacked packets < *sndwnd.*
- If number of unacked packets = *sndwnd*, sender must stop and wait for an ACK.
- If the packet received is not the packet expected, the receiver discards it.

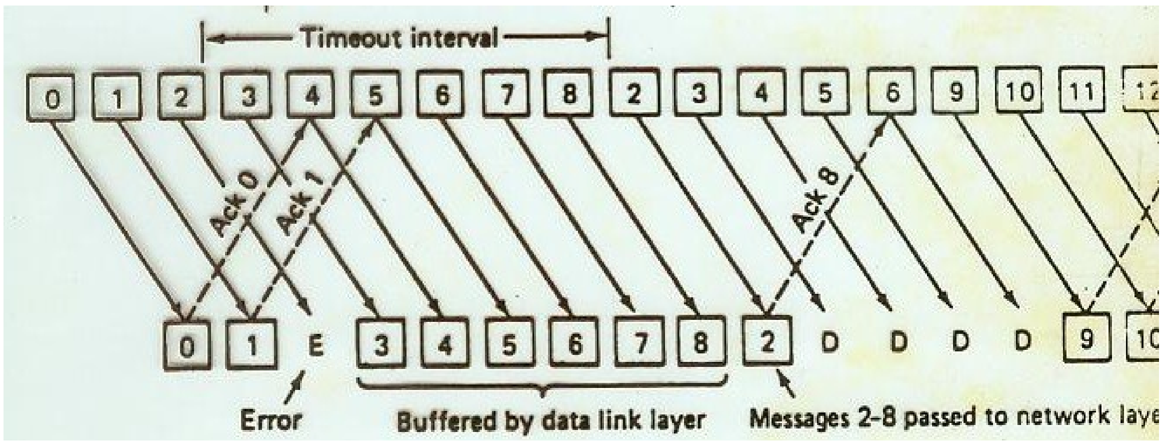
- A sender timeout causes the retransmission of entire block of *sndwnd* packets.
- This strategy works well for channels with large round trip latency but *low error rates*.

- Each time an error occur, the amount of wasted channel time is equal to the size of the *timeout* interval



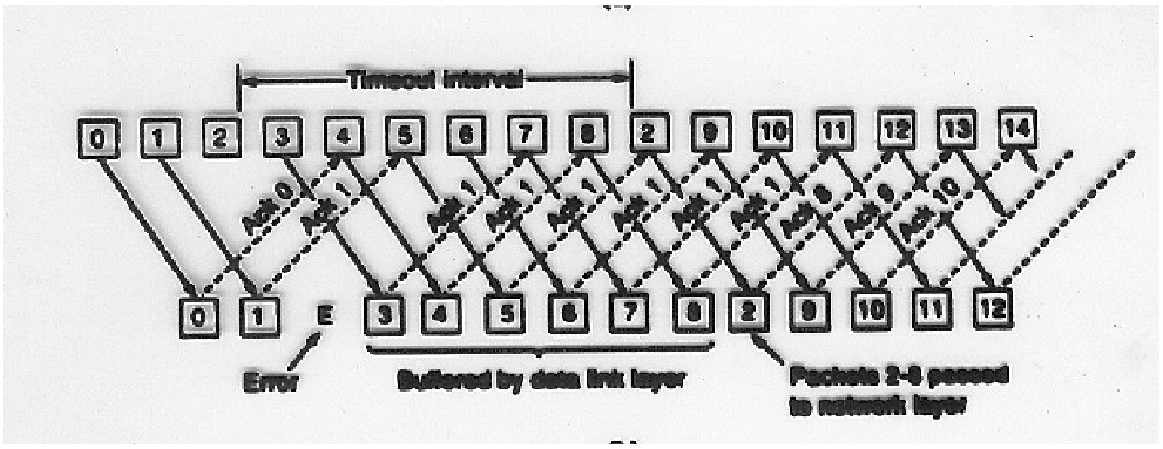
Receiver Buffering

- The sender uses the same logic as in *Go Back N.*
- If the packet received is not the packet expected, the receiver buffers it if it lies in the receiver window where *rcvwnd* can be as large and *sndwnd.*
- Sender timeout initiates retransmission of an entire block of *sndwnd* packets.
- Receiver ACK may make retransmission of entire block unnecessary.
- However, the deferred discards of frames 3 - 6 show that this protocol still wastes the two-way propagation delay in channel times.
- Since the lower bound on the timeout interval is the two way propagation delay this version of the protocol adds receiver complexity for no significant benefit and should be avoided.


Timeout interval
0 1 2 3 4 5 6 7 8 2 3 4 5 6 9 10 11 12
Ack 0
Ack 1
Ack 8
0 1 E 3 4 5 6 7 8 2 D D D D 9 10
Error
Buffered by data link layer
Messages 2-8 passed to network laye

Receiver buffering with out of order retransmission

- The Sender may send until *sndwnd* packets have been transmitted but then must wait for an ACK.
- If the packet received is not the packet expected, the receiver buffers it if it lies in the receiver window.
- If a packet times out the sender retransmits only that packet and then resumes transmitting where it left off (The sender must run independent timers for every outstanding packet... not just the one numbered *lstack.)*
- Receiver ACK may eliminate unnecessary retransmissions altogether.



Selective reject

- Sender may send until *sndwnd* packets have been transmitted but then must wait for an ACK.
- If the packet received is not the packet expected, the receiver buffers it if it lies in the receiver window.
- If the receiver detects a damaged or missing packet, it sends an explicit NAK.
- Receipt of NAK initiates retransmission of only the packet NAK'ed.
- Sender timeout may initiate retransmission of entire block of *sndwnd* packets.
- ACK for retransmitted packet hopefully makes retransmission of entire block unnecessary.

Analysis of the cost of errors:

Stop and Wait:

sndwnd=1 rcvwnd=1

When a packet is lost, sender delays for the timeout interval.

Lost time = time out interval

Lost packet times = timeout interval / packet time

Go back N:

sndwnd=N rcvwnd=1

Receiver discards damaged frame and all its successors.

Sender eventually times out and restarts.

Lost time = time out interval

Lost packet times = timeout interval / packet time

Receiver buffering:

sndwnd=rcvwnd=N

Receiver discards damaged frames but buffers its successors

Sender eventually times out and restarts.

As soon as receiver receives damaged frame it ACKs all buffered frames

Sender doesn't receive the ACK until two way prop delay elapses.

Lost time = two way propagation delay (barely) < time out interval.

Selective Reject/ Receiver buffer with OoO retransmission.

sndwnd=rcvwnd=N

Receiver explicitly NAKs damaged or missing frames and continues to buffer

Sender retransmits single frames after NAK.. but falls back to timeout and restart

Lost time = 1 packet time

Sequence number size and Sender Window Size limitations.

In the real world sequence numbers are of finite size and eventually wrap around. Failure to account for this can produce a *defective protocol!*

Suppose an *m* bit sequence number is used in a *Go Back N* protocol.

Suppose *m* = *2*. The set of possible sequence numbers is {0, 1, 2, 3}
Suppose the sender sends all four

```
F-0-0 ------------------>
F-1-0 ------------------>
F-2-0 ------------------>
F-3-0 ------------------>
      <------------------  A-0
```

Does this mean all got through or all got lost?? It is impossible to tell.

Thus the maximum value of *sndwnd* must be 3 with a 2 bit sequence number.

```
F-0-0 ------------------>
F-1-0 ------------------>
F-2-0 ------------------>
      <------------------  A-0
```

Now an A-0 unambiguously says that all three frames were lost.

In general the maximum safe size for *sndwnd in a Go Back N protocol with an m-bit sequence number is* is *2^m – 1*.

Sequence number size and receiver buffering

Now suppose *sndwnd = 3 rcvwnd = 2*

F-0-0 ---------------------->	Accepted SW = {0},	RW = {1, 2}
F-1-0 ---------------------->	Accepted SW = {0, 1},	RW = {2, 3}
F-2-0 ---------------------->	Accepted SW = {0, 1, 2},	RW = {3, 0}
blocked		
win full		
<----------------------	A-3	
(lost)		
timeout		
F-0-0 ---------------------->	Buffered though actually a duplicate (Thinks F-3 must have been lost)	
<----------------------	A-3	
F-3-0 ----------------------->	Accept 3 and buffered 0 RW={1,2}	
F-0-0 ----------------------->	Discarded as a duplicate outside window although actually the correct version of F-0.	

This problem is caused by the fact that trailing edge of the sender window and the leading edge of the and receiver windows overlapped in the sequence number space.

For sequence numbers in the overlapped area it is impossible for the receiver to know if a packet is a restransmission or new data

This cannot occur if $SWS + RWS <= 2^m$

Now suppose SWS = 2 and RWS = 2

```
F-0-0 ------------------>  Accepted  SW = {0},          RW = {1, 2}
F-1-0 ------------------>  Accepted  SW = {0, 1},       RW = {2, 3}
Blocked
win full
      <------------------  A-2
          (lost)
timeout
F-0-0 -------------------> Discarded since RW = {2,3}
      <------------------- A-2
F-2-0 -------------------> Accepted SW = {2}            RW = {3, 0}
```

Bits required for sequence numbers:

Go back N: ceiling($\log_2(N+1)$)

SRep/SRej: ceiling($\log_2(2N)$)

Relationship of sender window size and performance

Inadequate sender window size can have an *extremely adverse* impact on network throughput.

B =	bit rate of channel in bits / sec
L =	length of a packet in bits
R =	round trip propagation delay
L/B =	time to transmit a packet

Efficiency of stop and wait:

Efficiency = (time transmitting frame) / (total time required pert frame)=
= (packet-time(L/B) / (packet time + round trip latency)

= (L / B) / (L / B + R)
= L / (L + RB)

Example

B =	100,000 bits / sec
L =	4000 bits
R =	2 * 270 = 540 ms

Eff = 4000 / (4000 + 0.54 * 100,000) = 4000 / 58000 = 0.069

To obtain an efficiency of 1.0 it is necessary to transmit for the entire two way propagation delay without having to stop and wait for an ack.

For this to occur

*sndwnd * packet_time >= round_trip_time*

or

sndwnd >= round_trip_time / packet_time

Example:

Packet time = 0.04 seconds
sndwnd >= 0.54 / 0.04 = *ceil(13.5)* = 14

A process based model for a datalink layer implementation:

Processes
- Global
  - Router (= network layer)

- Per Link Processes
  - Read link
  - Queue manager
  - Timeout
  - Write link

Datalink and network layer share a pool of buffers
Buffers live in the following states:

```
Free
  |
--- A read completes and the read process needs a new input buffer ---
  |
Read pending
  |
--- A read completes and the read process passes the buffer to network layer
  |
Network layer input queue
  |
--- Network layer makes routing decision
  |
Link station output queue
  |
--- Queue manager attaches buffer to output window
  |
Link station output window
  |
--- A read completes and the ack processing determines buffer has been acked
  |
Free
```

Buffer Management Data Structures:

```
/* Buffer queue header an instance of this structure */
/* is used to manage each queue of buffers.          */

struct bqhtype
{
   int mtxsemid;            /* Id of mutex semaphore  */
   int avlsemid;            /* Buffer available semid */
   struct bqetype *bfirst;  /* Addr. of first buffer  */
   struct bqetype *blast;   /* Addr. of last buffer   */
};

/* Buffer queue element.. These are structures are    */
/* elements of the list defined above.                */

struct  bqetype
{
   struct bqetype *bnext;  /* Next buffer on list */
   struct blutype bdata;   /* Buffer data.        */
};
```

Buffer Queue Management Routines

```
/* enqueue.c */

/* Append a buffer to a buffer queue */

enqueue(qhdr, bufadr)
struct bqhtype *qhdr;    /* Address of target queue header  */
struct bqetype *bufadr;  /* Address of buffer queue element */
{

/* Obtain exclusive control of the queue header */

   semwait(nwad->semsetid, nwad->qhdmutex);

/* Add the buffer queue element to the end of the queue */

   bufadr->bnext = 0;
   if (qhdr->blast == 0)
   {
      qhdr->bfirst = bufadr;
      qhdr->blast = bufadr;
   }
   else
   {
      qhdr->blast->bnext = bufadr;
      qhdr->blast = bufadr;
   }

/* Signal buf avail and unlock the queue hdr */

   semsig(nwad->semsetid, qhdr->avlsemid);
   semsig(nwad->semsetid, nwad->qhdmutex);

}
```

```
/* dequeue.c */

/* Remove a buffer from a buffer queue */

struct bqetype *dequeue(qhdr)
struct bqhtype *qhdr;

{
   struct bqetype *bufadr;
   int sval;

/* Wait for buffer available on the queue */

   semwait(nwad->semsetid, qhdr->avlsemid);

/* Now get exclusive control of the queue header */

   semwait(nwad->semsetid, nwad->qhdmutex);

   bufadr = qhdr->bfirst;
   if (bufadr== 0)
   {
      error("Bad buffer address");
   }
   if (qhdr->bfirst == qhdr->blast)
   {
      qhdr->bfirst = 0;
      qhdr->blast = 0;
   }
   else
   {
      qhdr->bfirst = bufadr->bnext;
   }

   semsig(nwad->semsetid, nwad->qhdmutex);

   return(bufadr);
}
```

Link Management structures

```
/* Link station control structure.. The is one of these */
/* structures for each link at the IMP..                */

struct lsctype
{

/* The first three values are indexes into the */
/* semaphore table.                            */

   int wpcount;             /* # win slots to write sem*/
   int lscmutex;            /* Mutex for link station  */
   int numslots;            /* # win slots avail semid */

/* These values are indexes into the buffer address    */
/* and buffer time stamp arrays.                       */

   int nxtsend;             /* Next frame to send      */
   int nxtrecv;             /* Next frame expected.    */
   int lstack;              /* Next ack expected.      */
   int nxtqueue;            /* Next frame to queue.    */

/* These arrays represent the sender window.           */

   struct bqetype
      *bufads[MAX_SEQ + 1]; /* Buffer addresses .      */
   int
      buftimes[MAX_SEQ + 1];/* Time buffer sent.       */
```

Link Read Process

```
/* This process reads an input link and processes the incoming
packet */

readlink(linkid)
int linkid;

{
   struct bqetype *bufad;
   struct lsctype *lscad;

/* Get the address of link control block associated */
/* with this link.                                  */

   lscad = nwad->linktab + linkid;
/* Essentially this is a DO FOREVER */

   while (nwad->more)
   {

      bufad = dequeue(&nwad->freeq);   /* Get a buffer */

   /* Perform a blocking read to fill the buffer */

      cread(linkid, &bufad->bdata, sizeof(struct blutype));

      ackcheck(linkid, &bufad->bdata.lhdr);

   /* See if the sequence number is valid */

      if (bufad->bdata.lhdr.seqno == lscad->nxtrecv)

      /* If so then put the buffer
          on the network layer input list and increment the
          next frame  expected sequence number.  */
      {
         enqueue(&nwad->inputq, bufad);
         lscad->nxtrecv = inc(lscad->nxtrecv);
      }
      else  /* out of order packet */

      /* If not, just return the buffer to the free list */

      {
         enqueue(&nwad->freeq, bufad);
         fprintf(nodelog, "Bad sequence number \n");
         fflush(nodelog);
      }
   }
}
```

```
/* ackcheck.c */

/* This function does ack processing */

ackcheck(linkid, lhdr)
int linkid;
struct lhtype *lhdr;

{
   struct lsctype *lscad;

   lscad = nwad->linktab + linkid; /* Get lscb address */

/* Need mutex with the timeout procedure ..
   bugfix due to Mukul Kotwani, Oct 1999           */

   semwait(nwad->semsetid, lscad->lscmutex);

/* The ack number in the packet header contains the frame
   number of the next frame expected and acks all packets with a
   logically  smaller sequence number.        */

   if (is_between(lscad->lstack, lhdr->ackno, lscad->nxtsnd))
   {
        while (lscad->lstack != lhdr->ackno)
        {

        /* When a packet has been acked, the buffer may be
          returned to the free queue. Its addr was placed in a
          table with an entry for each number in the seq # space
          by the message queue manager routine.         */

           enqueue(&nwad->freeq, lscad->bufads[lscad->lstack]);
           lscad->lstack = inc(lscad->lstack);

        /* Signal the number of slots in the sender window sem */

           semsig(nwad->semsetid, lscad->numslots);
        }
   }
   semsig(nwad->semsetid, lscad->lscmutex);
}
```

Without the *is_between()* function, this protocol implementation can fail in a nasty way on delayed *acks.*

Consider the following scenario:

> lstack = 2
> nxtsend =12
> A time out occurs and nxtsend is set back to 2.
> Now a delayed ack 12 arrives.

The result will be that packets 2 .. 12 are in the sender window but the associated buffers are back on the free list!

It would also be necessary to move the *cwrite()* call within the *writelink()* procedure to within the protection of the *mutex.* Thanks to John MacPherson for pointing out the problem and suggesting these solutions.

Exercise: what use if any is the test:

```
lscad->lstack != lscad->nxtqueue)
```

Exercise: Devise a solution that does not require including the *cwrite()* within the *mutex.*

Output queue manager

```
/* queuemsg.c */

/* This routine removes messages from the output queue of a link
and assigns them to slots in the sender window.     */

queuemsg(linkid)
int linkid;

{
   struct bqetype *bufad;
   struct lsctype *lscad;
   int sval;
   int i;

   lscad = nwad->linktab + linkid;

   while (nwad->more)
   {

   /* Wait for an available slot in the senders window. */

      semwait(nwad->semsetid, lscad->numslots);

   /* Wait for an available message in the output queue */

      bufad = dequeue(&nwad->outputq[linkid]);

   /* Save the buffer address in the next slot in the link
      stations output buffer address table.              */

      lscad->bufads[lscad->nxtqueue] = bufad;
      lscad->nxtqueue = inc(lscad->nxtqueue);

   /* Signal the write pending counting semaphore to wake up
      the writer process.                              */

      semsig(nwad->semsetid, lscad->wpcount);

   }
}
```

Timeout Process

```
/* timeout.c */
timeout(numlinks)
int numlinks;
{
   int i;
   int temp;
   long tval;
   struct lsctype *lscad;

   while (nwad->more)
   {

   /* Sleep for the duration of the timeout interval */
      sleep(TIME_OUT);
      tval = time(0);    /* Get the current time */

   /* Check the output queue of each link */

      for (i = 1; i <= numlinks; i++)
      {
         lscad = nwad->linktab + i;
```

```
        /* See if there are any unacked frames */

           if (lscad->lstack != lscad->nxtsend)
           {
           /* See if the oldest frame has timed out. */

              if ((tval - lscad->buftimes[lscad->lstack]) >=
                      TIME_OUT)
              {

            /* Get control of the link control block and reset the
            next frame to send back to  the frame which has timed
            out. Signal the write pending semaphore for each frame
            which has been sent since the timed out frame.  */

                 semwait(nwad->semsetid, nwad->lscmutex);
                 temp = lscad->lstack;

                 while (temp != lscad->nxtsend)
                 {
                    semsig(nwad->semsetid, lscad->wpcount);
                    temp = inc(temp);
                 }
                 lscad->nxtsend = lscad->lstack;
                 semsig(nwad->semsetid,nwad->lscmutex);

              } /* Time out detected */
           } /* Outstanding frames exist */
        }  /* for current link station */
     }
  }
```

Write link process

```
/* writelink.c */

/* This routine is the output writer for a link */

writelink(linkid)
int linkid;

{
   struct bqetype *bufad;
   struct lsctype *lscad;
   int sval;
   int i;

/* Get address of link station control block. */

   lscad = nwad->linktab + linkid;

   while (nwad->more)
   {

   /* Wait on the write pending semaphore             */
   /* and then the link control block mutex semaphore */

      semwait(nwad->semsetid, lscad->wpcount);
      semwait(nwad->semsetid, nwad->lscmutex);

   /* Get the buffer address from the address table */
   /* and fill in the sequence # and ack number.    */

      bufad = lscad->bufads[lscad->nxtsend];
      bufad->bdata.lhdr.seqno = lscad->nxtsend;
      bufad->bdata.lhdr.ackno = lscad->nxtrecv;

   /* Set the time sent for the timeout and update */
   /* the next frame to send.                      */

      lscad->buftimes[lscad->nxtsend] = time(0);
      lscad->nxtsend = inc(lscad->nxtsend);

      semsig(nwad->semsetid, nwad->lscmutex);
      cwrite(linkid, &bufad->bdata, sizeof(struct blutype));
   }
}
```

Examples of the Data Link Layer

SDLC - Developed by IBM ~ 1970

A connection oriented / reliable data link protocol.

Modified and standardized by

ANSII	as	ADCCP
ISO	as	HDLC
CCITT	as	LAP later LAPB

SDLC Framing

Bit stuffing with flag byte 01111110

SDLC Frame Format

	Flag	Address	Control	--------- Info -------	CRC	Flag
Bytes	1	1	1	N	2	1

Addressing in SDLC

Link types
- Point to point
- Multipoint links

Every link has 1 *Primary* station and 1 or more *Secondary* stations

Primary controls access to link via polling (OK for links with very low *bandwidth x delay* products)

o------------o------------o---------------o---------------o

P S_1 S_2 S_3 S_4

Address byte *A* always identifies the secondary station so..

A = receiver when primary sends
A = sender when secondary sends

Frame types: Identified by control byte as follows:

Control Byte	Frame Type	Use
0xxx xxxx	Information	Normal data
10xx xxxx	Supervisory	Acks, NAKs
11xx xxxx	Unnumbered	Init, term, recovery

Control byte format: by frame type

Information

# bits	1	3	1	3
use	0	seq #	P/F	ack#

Supervisory

#bits	2	2	1	3
use	10	type	P/F	ack#

Unnumbered

#bits	2	2	1	3
use	11	type	P/F	mod

Use of Poll / Final (P/F) bit

When Primary sends the meaning is always Poll or !Poll
When Secondary sends meaning is final or !Final

Value/Sender	*Primary*	*Secondary*
0	!Poll	!Final
1	Poll	Final

Primary can poll with information frames
Secondaries can send multiple frames in response to a poll

Supervisory frames:

Type 0 - Receiver ready

Standalone ack
Standalone poll

Type 1 - Reject
A NAK that confirms receipt of frames up through ACK# - 1

Type 2 - Receiver not ready (provides flow control)

Used to ACK and tell the other end to desist
When secondary sends RNR, primary will continue to poll for RR
When primary sends RNR, secondary will desist until polled.

RNR works satisfactorily for links with *shallow pipelining.* For deep pipelines, by the time RNR is received enough packets to overwhelm the receiver's buffer space may already be in the pipe!

Type 3 - Selective reject.
NAK's the frame identified in ACK#.
Not in SDLC or LAPB

The Unnumbered protocol

Used for initialization and recovery.

The protocol is necessarily stateless since its mission is to recover from loss of state!

No sequence numbers or ACK numbers are used
Thus there is no way to tell how many copies are delivered.

Semantics must be idempotent..

Set state variable X to Y
Not toggle state variable X

Initialization / Recovery

RIM(SP)	- Request for initialization
SIM (PS)	- Orders secondary to initialize itself.
SNRM (PS)	- Resets sequence #'s establishes don't speak until requested rules. Enter normal data transfer mode.
SABM -	- Like SNRM but no primary / secondary relationship.
SNRME (PS)	- Use 7 bit sequence numbers.
SABME	- Like SABM but with 7 bit sequence #s
UA	- Unsequenced ACK

The datalink layer in ATM

ATM provides an unreliable connection oriented service.

ATM doesn't conform to the OSI model especially well, and so its somewhat difficult to identify the data link layer per se. We address the *framing* function here.

The TC Transmission Convergence) sublayer is actually the upper sub-layer of the physical layer (the lower sublayer is the PMD (physical media dependent).

It is concerned with transmission and reception of 53 byte cells.

Each cell contains a 5 byte header
 4 bytes control and routing information
 1 byte header CRC with generator = x^8 + x^2 + x + 1

This particular CRC can also perform single bit ECC

Framing is also based on the HEC.

A 40 bit shift register and a three state model are used

The *Hunt* state

 As each bit is shifted in the CRC check is performed to see if the last 8 bits comprise a valid CRC for the preceding 32.
 If so a possible cell boundary has been found and the *presynch* state is entered

The *Presynch* state

 The next 424 bits are then processed and the test is repeated.
 Failure -> return to hunt
 N successive success -> enter the *Synch* state.

This whole procedure violates layered protocol principles!

The SSCOP (Service Specific Connection Oriented Protocol)

The SSCOP is a reliable connection oriented protocol that provides reliable transport services for Q.2931 signaling . Even though it is said to provide *transport* services, it's best viewed as a *link layer* protocol because connections are always single hop point to point

Services provided include:

Flow control of rate at which switch or end system receives signaling messages
Error detection and retransmission.
Connection establishment and resynchronization.
Polling and status exchange.

All frames are a multiple of 4 bytes in length.

The last 32 bit word is a trailer containing control info

High order byte bit layout

8-7	Pad Length
6-5	Reserved
4-1	PDU Type

3 Low order bytes
    N(S) (sequence number)

SD - Sequenced Data type = 0x08

Information
Trailer(NS)

POLL - PDU type = 0x0a

Poll packets are periodically generated by the sender to ask if any SD packets are missing
Each Poll carries a sequence number from a *different* sequence number space
That is, Polls and SD's are numbered serially and independently

Word 1

Bytes 2-4	N(PS)	Poll seq #

Word 2 (standard control trailer format)

Byte 1	PDU_TYPE	0x0a = poll
Bytes 2-4	N(S)	next packet sequence number to be sent

STAT - PDU type = 0x0b

The STAT PDU is used to respond to a status request (POLL PDU) received from a peer SSCOP entity. It contains information regarding the reception status of SD PDUs, credit information for the peer transmitter, and the sequence number of the POLL PDU to which it is in response.

The list elements identify blocks of missing and correctly received packets
Odd entries identify start of a block of missing SD packets
Even entries identify start of a block of SD packets received correctly

List element 1 (a SD PDU N(S)) <--- missing PDU
List element 2 (a SD PDU N(S)) <--- correctly received PDU
:
:
List element 2n (a SD PDU N(S)) <--- correctly received PDU

Following the list are three words:

Word 1

N(PS)	PS # to which this is a response

Word 2

N(MR)	maximum # willing to receive now

Word 3 (standard control trailer format)

Byte 1	PDU_TYPE 0x0a = poll
Bytes 2 - 4	N(R) sequence # of next packet expected.

USTAT - (unsolicited STAT)

Sent when receiver detects a gap in received sequence numbers

List element 1 (a SD PDU N(S))
List element 2 (a SD PDU N(S))
N(MR)
N(R)

The main claim to fame of the SSCOP is that *no* unnecessary retransmissions need ever be made.

However, the benefits of the "claim to fame" are offset to a degree by the overhead of POLL/STAT

The *stat* feature identifies precisely which packets were lost in transit.
The ability to match stats with the corresponding poll makes it possible to avoid unnecessary duplicate retransmissions.

A timeout mechanism aborts the connection if no STATs are received during a timeout interval.

Example of the SSCOP in operation

```
IO: ----------IO: TO NET (0.0.5): BGN PDU (8 bytes)
IO:  00 00 00 00 01 00 00 1e    (Code for BGN is 01 and N(MR) is 1e
IO: ----------
IO: FROM NET (0.0.5): BGAK PDU (8 bytes)
IO:  00 00 00 00 02 00 00 0a    (code for BGAK is 02 and N(MR) is 0a
IO: ----------
IO: TO NET (0.0.5): POLL PDU (8 bytes)
IO:  00 00 00 01 0a 00 00 00
IO: ----------
IO: FROM NET (0.0.5): STAT PDU (12 bytes)
IO:  00 00 00 01 00 00 00 0a 0b 00 00 00   (next expected frame is 0)
IO: ----------
IO: FROM NET (0.0.5): POLL PDU (8 bytes)
IO:  00 00 00 01 0a 00 00 00
IO: ----------
IO: TO NET (0.0.5): STAT PDU (12 bytes)
IO:  00 00 00 01 00 00 00 1e 0b 00 00 00
IO: ----------
IO: TO NET (0.0.5): SD PDU (120 bytes)
IO:  09 03 00 00 01 05 80 00 68 58 80 00 0b 05 8c 05 ec 81 05 ec
IO:  83 01 84 00 59 80 00 09 84 05 63 b7 85 05 63 b7 be 5e 80 00
IO:  03 10 80 80 5f 80 00 09 6b 40 80 80 00 a0 3e 00 01 70 80 00
IO:  15 82 47 02 03 04 05 06 07 08 09 00 00 03 01 00 01 02 03 04
IO:  05 02 6c 80 00 15 82 47 02 03 04 05 06 07 08 09 00 00 03 01
IO:  00 04 ac 6c 30 b3 00 5c e0 00 02 00 00 00 00 00 c8 00 00 00
IO: ----------
IO: FROM NET (0.0.5): SD PDU (24 bytes)
IO:  09 03 80 00 01 02 80 00 09 5a 80 00 05 88 00 00 00 2d 00 00
IO:  88 00 00 00
IO: ----------
IO: FROM NET (0.0.5): SD PDU (24 bytes)
IO:  09 03 80 00 01 07 80 00 09 5a 80 00 05 88 00 00 00 2d 00 00
IO:  88 00 00 01
IO: ----------
IO: TO NET (0.0.5): SD PDU (16 bytes)
IO:  09 03 00 00 01 0f 80 00 00 00 00 00 c8 00 00 01
IO: ----------
IO: FROM NET (0.0.5): SD PDU (136 bytes)
IO:  09 03 00 00 01 05 80 00 79 6c 80 00 16 02 81 47 02 03 04 05
IO:  06 07 08 09 00 00 03 01 00 01 02 03 04 05 02 70 80 00 15 82
IO:  47 02 03 04 05 06 07 08 09 00 00 03 01 00 04 ac 6c 30 b3 00
IO:  5f 80 00 09 6b 40 80 80 00 a0 3e 00 01 58 80 00 0b 05 8c 05
IO:  ec 81 00 00 83 01 84 00 54 80 00 03 00 00 3c 59 80 00 09 84
IO:  00 fe 50 85 00 00 00 be 5e 80 00 03 10 80 81 5c e0 00 02 00
IO:  00 5a 80 00 05 88 00 00 00 2e 00 00 88 00 00 02
IO: ----------
```

```
IO: TO NET (0.0.5): SD PDU (20 bytes)
IO:  09 03 80 00 01 07 80 00 07 54 80 00 03 00 00 3c 08 00 00 02
IO: ----------
IO: FROM NET (0.0.5): SD PDU (16 bytes)
IO:  09 03 00 00 01 0f 80 00 00 00 00 00 c8 00 00 03
IO: ----------
IO: TO NET (0.0.5): SD PDU (120 bytes)
IO:  09 03 00 00 02 05 80 00 68 58 80 00 0b 05 8c 05 ec 81 05 ec
IO:  83 01 84 00 59 80 00 09 84 05 63 b7 85 05 63 b7 be 5e 80 00
IO:  03 10 80 80 5f 80 00 09 6b 40 80 80 00 a0 3e 00 04 70 80 00
IO:  15 82 47 02 03 04 05 06 07 08 09 00 00 03 01 00 01 02 03 04
IO:  05 02 6c 80 00 15 82 47 02 03 04 05 06 07 08 09 00 00 03 01
IO:  00 04 ac 6c 30 b3 00 5c e0 00 02 00 00 00 00 00 c8 00 00 03
IO: ----------
IO: FROM NET (0.0.5): SD PDU (24 bytes)
IO:  09 03 80 00 02 02 80 00 09 5a 80 00 05 88 00 00 00 2f 00 00
IO:  88 00 00 04
IO: ----------
IO: FROM NET (0.0.5): SD PDU (24 bytes)
IO:  09 03 80 00 02 07 80 00 09 5a 80 00 05 88 00 00 00 2f 00 00
IO:  88 00 00 05
IO: ----------
IO: TO NET (0.0.5): SD PDU (16 bytes)
IO:  09 03 00 00 02 0f 80 00 00 00 00 00 c8 00 00 04
IO: ----------
IO: FROM NET (0.0.5): SD PDU (136 bytes)
IO:  09 03 00 00 02 05 80 00 79 6c 80 00 16 02 81 47 02 03 04 05
IO:  06 07 08 09 00 00 03 01 00 01 02 03 04 05 02 70 80 00 15 82
IO:  47 02 03 04 05 06 07 08 09 00 00 03 01 00 04 ac 6c 30 b3 00
IO:  5f 80 00 09 6b 40 80 80 00 a0 3e 00 04 58 80 00 0b 05 8c 05
IO:  ec 81 00 00 83 01 84 00 54 80 00 03 00 00 3c 59 80 00 09 84
IO:  00 fe 50 85 00 00 00 be 5e 80 00 03 10 80 81 5c e0 00 02 00
IO:  00 5a 80 00 05 88 00 00 00 30 00 00 88 00 00 06
IO: ----------
IO: TO NET (0.0.5): SD PDU (20 bytes)
IO:  09 03 80 00 02 07 80 00 07 54 80 00 03 00 00 3c 08 00 00 05
IO: ----------
IO: FROM NET (0.0.5): SD PDU (16 bytes)
IO:  09 03 00 00 02 0f 80 00 00 00 00 00 c8 00 00 07
IO: FROM NET (0.0.5): POLL PDU (8 bytes)
IO:  00 00 00 02 0a 00 00 08
IO: TO NET (0.0.5): STAT PDU (12 bytes)
IO:  00 00 00 02 00 00 00 26 0b 00 00 08
IO: ----------
IO: ----------
IO: TO NET (0.0.5): POLL PDU (8 bytes)
IO:  00 00 00 02 0a 00 00 06
IO: ----------
IO: FROM NET (0.0.5): STAT PDU (12 bytes)
IO:  00 00 00 02 00 00 00 10 0b 00 00 06
IO: ----------
```

An example of a timeout and restart caused by a bug in a device driver

Values in the left colum are seconds:microseconds

Normal polling interval is O(5~10) seconds

Note 22 second delay between 257 and 279.

The failing device driver buffered all packets received and then delivered them all in a burst when all buffers were full.

By that time the other end had given up.

```
983384252:620281 atmsigd:IO: TO NET (0.0.5): POLL PDU (8 bytes)
983384252:620313 atmsigd:IO:  00 00 13 96 0a 00 01 11
983384252:621504 atmsigd:IO: ----------
983384252:621520 atmsigd:IO: FROM NET (0.0.5): STAT PDU (12 bytes)
983384252:621547 atmsigd:IO:  00 00 13 96 00 00 01 1b 0b 00 01 11
983384257:121355 atmsigd:IO: ----------
983384257:121397 atmsigd:IO: FROM NET (0.0.5): POLL PDU (8 bytes)
983384257:121426 atmsigd:IO:  00 00 13 79 0a 00 01 1a
983384257:121453 atmsigd:IO: TO NET (0.0.5): STAT PDU (12 bytes)
983384257:121481 atmsigd:IO:  00 00 13 79 00 00 01 38 0b 00 01 1a
983384279:498819 atmsigd:IO: ----------
983384279:498860 atmsigd:IO: TO NET (0.0.5): POLL PDU (8 bytes)
983384279:498888 atmsigd:IO:  00 00 13 97 0a 00 01 11
983384279:498921 atmsigd:IO: ----------
983384279:498935 atmsigd:IO: FROM NET (0.0.5): POLL PDU (8 bytes)
983384279:499171 atmsigd:IO:  00 00 13 7a 0a 00 01 1a
983384279:499186 atmsigd:IO: TO NET (0.0.5): STAT PDU (12 bytes)
983384279:499213 atmsigd:IO:  00 00 13 7a 00 00 01 38 0b 00 01 1a
983384279:499236 atmsigd:IO: ----------
983384279:499248 atmsigd:IO: FROM NET (0.0.5): POLL PDU (8 bytes)
983384279:499466 atmsigd:IO:  00 00 13 7b 0a 00 01 1a
983384279:499477 atmsigd:IO: TO NET (0.0.5): STAT PDU (12 bytes)
983384279:499504 atmsigd:IO:  00 00 13 7b 00 00 01 38 0b 00 01 1a
983384279:499524 atmsigd:IO: ----------
983384279:499536 atmsigd:IO: FROM NET (0.0.5): POLL PDU (8 bytes)
983384279:499557 atmsigd:IO:  00 00 13 7c 0a 00 01 1a
983384279:499763 atmsigd:IO: TO NET (0.0.5): STAT PDU (12 bytes)
983384279:499790 atmsigd:IO:  00 00 13 7c 00 00 01 38 0b 00 01 1a
983384279:499810 atmsigd:IO: ----------
983384279:499822 atmsigd:IO: FROM NET (0.0.5): POLL PDU (8 bytes)
983384279:500039 atmsigd:IO:  00 00 13 7d 0a 00 01 1a
983384279:500050 atmsigd:IO: TO NET (0.0.5): STAT PDU (12 bytes)
983384279:500076 atmsigd:IO:  00 00 13 7d 00 00 01 38 0b 00 01 1a
983384279:500096 atmsigd:IO: ----------
983384279:500108 atmsigd:IO: FROM NET (0.0.5): END PDU (8 bytes)
983384279:500129 atmsigd:IO:  00 00 00 00 13 00 00 00
983384279:503922 atmsigd:IO: TO NET (0.0.5): ENDAK PDU (8 bytes)
983384279:504888 atmsigd:IO:  00 00 00 00 04 00 00 00
983384279:505988 atmsigd:SSCF: entering state 1/2
983384279:506563 atmsigd:SSCF: entering state 2/2
983384279:507068 atmsigd:IO: TO NET (0.0.5): BGN PDU (8 bytes)
983384279:507971 atmsigd:IO:  00 00 00 00 01 00 00 1e
983384279:509129 atmsigd:IO: ----------
```

```
983384279:509698 atmsigd:IO: FROM NET (0.0.5): BGN PDU (8 bytes)
983384279:511250 atmsigd:IO:  00 00 00 00 01 00 00 0a
983384279:511709 atmsigd:IO: TO NET (0.0.5): BGAK PDU (8 bytes)
983384279:513434 atmsigd:IO:  00 00 00 00 02 00 00 1e
983384279:515331 atmsigd:SSCF: entering state 4/10
983384279:525146 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x000058
983384279:526708 atmsigd:IO: TO NET (0.0.5): SD PDU (16 bytes)
983384279:528666 atmsigd:IO:  09 03 00 00 58 75 80 00 00 00 00 00 c8 00 00 00
983384279:532174 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x000057
983384279:533785 atmsigd:IO: TO NET (0.0.5): SD PDU (16 bytes)
983384279:535741 atmsigd:IO:  09 03 00 00 57 75 80 00 00 00 00 00 c8 00 00 01
983384279:539418 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x800009
983384279:540152 atmsigd:IO: TO NET (0.0.5): SD PDU (16 bytes)
983384279:543355 atmsigd:IO:  09 03 80 00 09 75 80 00 00 00 00 00 c8 00 00 02
983384279:546210 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x800008
983384279:548022 atmsigd:IO: TO NET (0.0.5): SD PDU (16 bytes)
983384279:550089 atmsigd:IO:  09 03 80 00 08 75 80 00 00 00 00 00 c8 00 00 03
983384279:554295 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x000004
983384279:554980 atmsigd:IO: TO NET (0.0.5): SD PDU (16 bytes)
983384279:556935 atmsigd:IO:  09 03 00 00 04 75 80 00 00 00 00 00 c8 00 00 04
983384279:560986 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x800002
(16 bytes)
983384279:561560 atmsigd:IO: TO NET (0.0.5): SD PDU (20 bytes)
983384279:564869 atmsigd:IO:  09 03 80 00 02 75 80 00 07 54 80 00 03 00 00 3c 08
00 00 05
983384279:567705 atmsigd:SIGD: TO SAAL (0.0.5): STATUS_ENQ (0x75) CR 0x000002
983384279:569617 atmsigd:IO: TO NET (0.0.5): SD PDU (16 bytes)
983384279:571701 atmsigd:IO:  09 03 00 00 02 75 80 00 00 00 00 00 c8 00 00 06
```