Computer Networks

Data Link Layer

Topics

- Introduction
- Errors
- Protocols
- Modeling
- Examples

Introduction

- Reliable, efficient communication between two adjacent machines
- Machine A puts bits on wire, B takes them off. Trivial, right? Wrong!
- Challenges:
  - Circuits make errors
  - Finite data rate
  - Propagation delay
- Protocols must deal!

Data Link Services

- Network layer has bits
- Says to data link layer:
  - "send these to this other network layer"
- Data link layer sends bits to other data link layer
- Other data link layer passes them up to network layer

Data Link Services

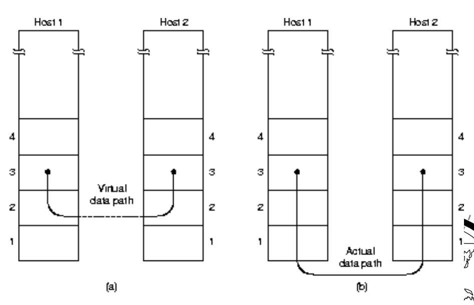


Fig. 3-1. (a) Virtual communication. (b) Actual communication.

Data Link Placement

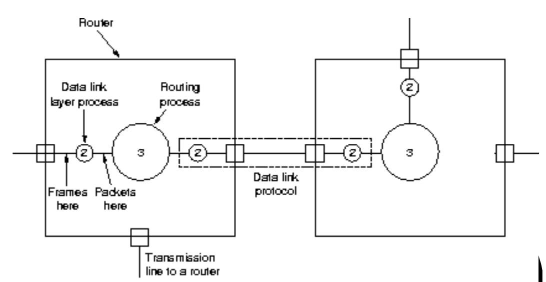


Fig. 3-2. Placement of the data link protocol.

Types of Services Possible

- Reliable Delivery
  - All frames arrive
  - Same order as generated by the sender
- Best Effort
  - No acknowledgements
  - Why would you want this service?
    - When loss infrequent, easy for upper layer to recover
    - "Better never than late"
- Acknowledged Delivery
  - Server acknowledges (or not), doesn't retransmit

Framing

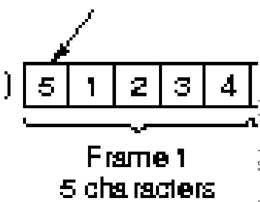
- Data link breaks physical layer stream of bits into *frames*

  `...01011010010100110101001O...`

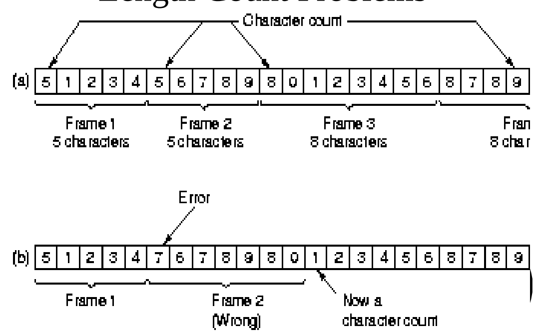
- How does receiver detect boundaries?
  - Length count
  - Special characters
  - Bit stuffing
  - Special encoding

Length count

- First field is length of frame
- Count until end
- Then, look for next frame
- Problems?



Length Count Problems



Special Characters

- Reserved characters for beginning and end
- Beginning:
  - DLE STX (Data-Link Escape, Start of TeXt)
- End:
  - DLE ETX (Data-Link Escape, End of TeXt)
- Problems?
- Solution?

Character Stuffing

- Replace DLE in data with DLE DLE (reverse)

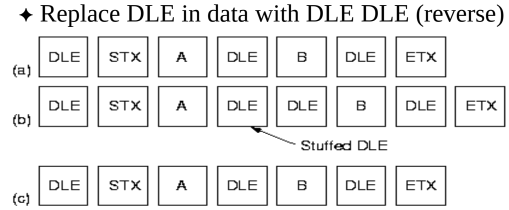


Fig. 3-4. (a) Data sent by the network layer. (b) Data after being character stuffed by the data link layer. (c) Data passed to the network layer on the receiving side.

- Not all architectures are character oriented!

Bit Stuffing

Frame delimiter: `01111110`

(a) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

(b) 0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0

Stuffed bits

(c) 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

- Garbaged frames ok, just keep scanning
- Problem? Wasted bandwidth/processing
  - How much in proj1?

Special Encoding

- Send a signal that does not have legal representation
  - low to high means a 1
  - high to low means a 0
  - high to high means frame end
  - IEEE 802.4
- Lastly, combination of above:
  - length plus frame boundary
  - IEEE 802.3

Topics

- Introduction ✓
- Framing ✓
- Errors ←
  - why
  - detecting
  - correction
- Protocols
- Modeling **?**
- Examples **?**

Review

- What is *framing*?
- What are the four ways the data link layer may do *framing*?
- What is *hamming distance*?

Errors

- Lines becoming digital
  - errors rare
- Copper the “last mile”
  - errors infrequent
- Wireless
  - errors common
- Errors are here for a while
- Plus, consecutive errors
  - bursts

Handling Errors

- Add redundancy to data
- Example:
  - “hello, world” is the data
  - “hzllo, world” received (detect? correct?)
  - “xello, world” received (detect? correct?)
  - “jello, world” received (detect? correct?)
  - what about similar analysis with “caterpillar”?
- Some: *error detection*
- More: *error correction*

What is an Error?

- Frame has *m* data bits, *r* redundancy bits
  - n = (m+r) bit *codeword*
- Given two codewords, compute distance:
  - `10001001`
  - `10110001`
  - XOR, 3 bits difference
  - *Hamming Distance*
- "So what?"

Code Hamming Distance

- Two codewords are *d* bits apart,
  - then *d* errors are required to convert one to other
- *Code Hamming Distance* min distance between any two legal codewords

Hamming Distance Example

- Consider 8-bit code with 4 codewords:

  `00000000` `00001111` `11110000` `11111111`

- What is the *Hamming distance*?
- What is the min bits needed to encode?
  - What are n, m, and r?
- What if `00001110` arrives?
- What if `00001100` arrives?

Parity Bit

- Single bit is appended to each data chunk
  - makes the total number of 1 bits even/odd
- Example: for even parity
  - `1000000(1)`
  - `1111101(0)`
  - `0000000(1)`
- What is the *Hamming distance*?
- What bit errors can it detect?
- What bit errors can it correct?

Ham On

- Consider a 10-bit code with 4 codewords:
  00000 00000 00000 11111 11111 00000 11111 11111
- *Hamming distance*?
- Correct how many bit errors?
  - 10111 00010 received, becomes 11111 00000 corrected
  - 11111 00000 sent, 00011 00000 received
- Might do better
  - 00111 00111 received, 11111 11111 corrected
  - and contains 4 single-bit errors

Fried Ham

- All possible data words are legal
- Choosing careful redundant bits can results in large Hamming distance
  - to be better able to detect/correct errors
- To detect *d* 1-bit errors requires having a Hamming Distance of at least *d+1* bits
  - Why?
- To correct *d* errors requires *2d+1* bits.
  - Why?

Designing Codewords

- Fewest number of bits needed for 1-bit errors?
  - `n=m+r` bits to correct all 1-bit errors
- Each message has *n* illegal codewords a distance of 1 from it
  - form codeword (n-bits)
  - invert each bit, one at a time
- Need *n+1* bits for each message
  - *n* that are one bit away and 1 for the message

Designing Codewords (cont)

- The total number of bit patterns = 2^n
  - So, $(n+1)\ 2^m \leq 2^n$
  - So, $(m+r+1) \leq (2^{m+r}) / 2^m$
  - Or, $(m+r+1) \leq 2^r$
- Given *m*, have lower limit on the number of check bits required to detect (and correct!) 1-bit errors

Example

- 8-bit codeword
- How many check bits required to detect and correct 1-bit errors?
- $(8 + r + 1) < 2^r$
  - Is 3 bits enough?
  - Is 5 bits enough?
- Use *Hamming code* to achieve lower limit

Hamming Code

- Bits are numbered left-to-right starting at 1
- Powers of two (1, 2, 4 ...) are check bits
- Check bits are parity bits for previous set
- Bit checked by only those check bits in the expansion
  - example: bit 19 expansion = 1 + 2 + 16
- Examine parity of each check bit, *k*
  - If not, add *k* to a *counter*
- If 0, no errors else *counter* gives bit to correct

Ham It Up

- ASCII character ‘`a`’ = `1100001`
- Check bit 1 covers bits 1, 3, 5 ...
- Check bit 2 covers bits 2, 3, 6, 7, 10, 11 ...
- Check bit 3 covers bits 4, 5, 6, 7, 12, 13 ...
- Check bit 4 covers bits 8, 9, 10, 11, 12 ...
  - (Work through on board)
- ASCII character ‘`d`’ = `1100100`
  - (Work through on board)

Hamming Code and Burst Errors

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	11111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	00101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

Ladies and Gentlemen … the Great *Hamdini*!

A volunteer from the audience?

Pick a number, any number, between 1 and 50

Is the Number in Here?

1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49

Is the Number in Here?

2 3 6 7 10 11 14 15 18 19 22 23 26 27 30 31 34 35 38 39 42 43 46 47 50

Is the Number in Here?

4 5 6 7 12 13 14 15 20 21 22 23 28 29 30 31 36 37 38 39 44 45 46 47

Is the Number in Here?

8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31 40 41 42 43 44 45 46 47

Is the Number in Here?

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 49 50

Is the Number in Here?

32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48
49 50

And the Number is ….

- (Drum roll ….)
- How is it done?

Error Correction

- Expensive
  - example: 1000 bit message
  - Correct single errors?
  - Detect single errors?
- Useful mostly:
  - simplex links (one-way)
  - long delay links (say, satellite)
  - links with very high error rates
    - would get garbled every time resent

Error Detection

- Most popular use *Polynomial Codes* or *Cyclic Redundancy Codes* (CRCs)
  - checksums
- Acknowledge correctly received frames
- Discard incorrect ones
  - may ask for retransmission

Polynomial Codes

- Bit string as polynomial w/0 and 1 coeffs
  - ex: k bit frame, then x^{k-1} to x^0
  - ex: `10001` is $1x^4+0x^3+0x^2+0x^1+1x^0 = x^4+x^0$
- Polynomial arithmetic mod 2

```
 10011011   11110000   00110011
+11001010  -10100110  +11001101
 01010001   01010110   11111110
```

- Long division same, except subtract as above
- “Ok, so how do I use this information?”

Doing CRC

- Sender + receiver agree *generator polynomial*
  - $G(x)$, ahead of time, part of protocol
  - with low and high bits a ‘1’, say `1001`
- Compute checksum to frame (m bits)
  - $M(x)$ + checksum to be evenly divisible by $G(x)$
- Receiver will divide by $G(x)$
  - If no remainder, frame is ok
  - If remainder then frame has error, so discard
- “But how do we compute the checksum?”

Computing Checksums

- Let r be *degree* of $G(x)$
  - *If* $G(x) = x^2+x^0$ = `101`, then r is 2
- Append r zero bits to frame $M(x)$
  - get $x^r M(x)$
  - ex: `1001` + *`00`* = `100100`
- Divide $x^r M(x)$ by $G(x)$ using mod 2 division
  - ex: `100100 / 101`
- Care about *remainder*
- "Huh? Do you have an example?"

Dividing $x^r M(x)$ by $G(x)$

```
          ____1011__
101  |  100100
          101
           011
           000
            110
            101
             110
             101
              11    ← Remainder
```

"Ok, now what?"

Computing Checksum Frame

- Subtract (mod 2) remainder from $x^r M(x)$

```
100100
    11
100111
```

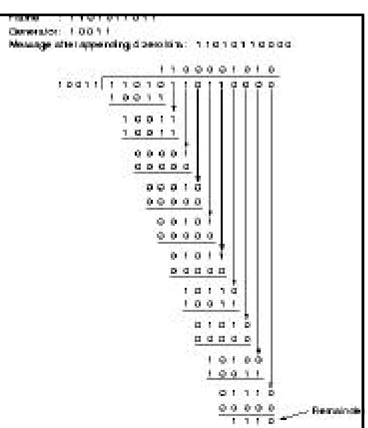
- Result is checksum frame to be transmitted
  - $T(x)$ = `100111`
- What if we divide $T(x)$ by $G(x)$?
  - Comes out evenly, with no remainder
  - Ex: 210,278 / 10,941 remainder 2399
  - 210,279 - 2399 is divisible by 10,941
- "Cool!"

Let's See if it Worked

```
          ____1011__
101  |  100111
          101
           011
           000
            111
            101
             101
             101
               0    ← yeah!
```

Another Example

(Figure 3-7)



Power of CRC?

- Assume an error, $T(x) + E(x)$ arrives
- Each 1 bit in $E(x)$ is an inverted bit
- Receiver does $[T(x) + E(x)] / G(x)$
- Since $T(x) / G(x) = 0$, result is $E(x) / G(x)$
- If $G(x)$ factor of $E(x)$, then error slips by
  - all other errors are caught
- Consider a 1-bit error, $E(x) = x^i$
  - i is the bit in error
  - If $G(x)$ contains two+ terms, never divides $E(x)$ so will catch all 1-bit errors

Power of CRC

- If there are two isolated single bit errors
  - $E(x) = x^i + x^j$ where $i > j$
  - $E(x) = x^j(x^{i-j} + 1)$
- If G(x) does not divide x^k+1 up to max frame length, will catch all double errors
- Some known polynomials:
  - $X^{15}+x^{14}+1$ will not divide x^k+1 up to k=32,768

Power of CRC!

- Odd number of bits in *E*(x)
  - ex: x^5+x^2+1, not x^2+1
- Then, x+1 will not divide it
- So, make x+1 a factor of G(x)
  - catch *all* errors with odd number of bits
- Polynomial w/ *r* check bits detect bursts $\leq r$
  - r+1 burst only if identical to *G*(x)
  - probability of bits after 1 are the same: $(1/2)^{r-1}$
  - burst > (*r*+1), $(1/2)^r$

Power of CRC!!

- Standards:
  - CRC-12 $= x^{12}+x^{11}+x^3+x^2+x^1+1$ (for 6-bit chars)
  - CRC-16 $= x^{16}+x^{15}+x^2+1$
  - CRC-CCITT $= x^{16}+x^{12}+x^5+1$
- Catch:
  - *all* single and double errors
  - *all* errors with odd number of bits
  - *all* burst errors 16 bits or less
  - 99.997% of all 17 bit errors
  - 99.998% of all 18-bit or longer bursts

Topics

- Introduction ✓
- Errors ✓
- Protocols ←
  - simple
  - sliding window
- Modeling ?
- Examples ?

Protocols Purpose

- Agreed means of communication between sender and receiver
- Handle reliability
- Handle flow control
- We'll move through basic to complex

Data Link Protocols

- Machine *A* wants stream of data to *B*
  - assume reliable, 1-way, connection-oriented
- Physical, Data Link, Network are all *processes*
- Assume:
  - `to_physical_layer()` to send frame
  - `from_physical_layer()` to receive frame
  - both do checksum
  - `from_physical_layer()` reports *success* or *failure*

Frame

kind	seq	ack	info

- first 3 are control (*frame header*)
- *info* is data
- *kind*: tells if data, some are just control
- *seq*: sequence number
- *ack*: acknowledgements
- Network has *packet*, put in frame's *info*
- Header is not passed up to network layer

Unrestricted Simplex Protocol

- Simple, simple, simple
- One-way data transmission (simplex)
- Network layers always ready
  - infinitely fast
- Communication channel error free
- "Utopia"

"Utopia"

```
void sender1(void)

 frame s;                      /* buffer for an outbound frame */
 packet buffer;                /* buffer for an outbound packet */

 while (true) {
     from_network_layer(&buffer);        /* go get something to send */
     s.info = buffer;          /* copy it into s for transmission */
     to_physical_layer(&s);    /* send it on its way */
 }                             /* Tomorrow, and tomorrow, and tomorrow,
                                  Creeps in this petty pace from day to day
                                  To the last syllable of recorded time
                                      - Macbeth, V, v */

void receiver1(void)

 frame r;
 event_type event;             /* filled in by wait, but not used here */

 while (true) {
     wait_for_event(&event); /* only possibility is frame_arrival */
     from_physical_layer(&r);/* go get the inbound frame */
     to_network_layer(&r.info);          /* pass the data to the network layer */
 }
```

Simplex Stop-and-Wait Protocol

- One-way data transmission (simplex)
- Communication channel error free
- <u>Remove</u> assumption that network layers are always ready
  - (or that receiver has infinite buffers)
- Could add timer so won't send too fast?
  - Why is this a bad idea?
- What else can we do?

Stop and Wait

```
void sender2(void)
{
 frame s;                      /* buffer for an outbound frame */
 packet buffer;                /* buffer for an outbound packet */
 event_type event;             /* frame_arrival is the only possibility */

 while (true) {
     from_network_layer(&buffer);                /* go get something to send */
     s.info = buffer;          /* copy it into s for transmission */
     to_physical_layer(&s);    /* bye bye little frame */
     wait_for_event(&event);   /* do not proceed until given the go ahead */
 }
}

void receiver2(void)
{
 frame r, s;                   /* buffers for frames */
 event_type event;             /* frame_arrival is the only possibility */
 while (true) {
     wait_for_event(&event);   /* only possibility is frame_arrival */
     from_physical_layer(&r);  /* go get the inbound frame */
     to_network_layer(&r.info); /* pass the data to the network layer */
     to_physical_layer(&s);    /* send a dummy frame to awaken sender */
 }
}
```

Simplex Protocol for Noisy Channel

- One-way data transmission (simplex)
- Remove assumption that communication channel error free
  - frames lost or damaged
- Damaged frames not acknowledged
  - look as if lost
- Can we just add a timer in the sender?
  - Why not? (Hint: think of acks)

Why a Timer Alone Will Not Work

- A sends frame to B
- B receives frame, passes to network layer
- B sends ack to A
- ack gets lost
- A times out. Assumes data frame lost
- A re-sends frame to B
- B receives frame, passes to network layer
  – duplicate!

Why a Sequence Number Alone Will Not Work

- A sends frame 0 to B
- B receives frame, passes to network layer
- A times out, resends 0
- B sends ack to A
- A receives ack, sends frame 1, frame 1 lost
- B receives frame 0 again, sends ack only
- A receives ack, sends frame 2
  – frame 1 never accepted!
- How to fix?

PAR Protocol - Sender

```
void sender3(void)
{
  seq_nr next_frame_to_send;              /* seq number of next outgoing frame */
  frame s;                                /* scratch variable */
  packet buffer;                          /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;                 /* initialize outbound sequence numbers */
  from_network_layer(&buffer);            /* fetch first packet */
  while (true) {
      s.info = buffer;                    /* construct a frame for transmission */
      s.seq = next_frame_to_send;         /* insert sequence number in frame */
      to_physical_layer(&s);              /* send it on its way */
      start_timer(s.seq);                 /* if (answer takes too long, time out */
      wait_for_event(&event);             /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);        /* get the acknowledgement */
          if (s.ack == next_frame_to_send) {
              from_network_layer(&buffer);/* get the next one to send */
              inc(next_frame_to_send); /* invert next_frame_to_send */
          }
      }
  }
}
```

PAR Protocol - Receiver

```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);             /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {       /* a valid frame has arrived. */
          from_physical_layer(&r);        /* go get the newly arrived frame */
          if (r.seq == frame_expected) {  /* this is what we have been waiting for. */
              to_network_layer(&r.info); /* pass the data to the network layer */
              inc(frame_expected);        /* next time expect the other sequence nr */
          }
          s.ack = 1 – frame_expected;     /* tell which frame is being acked */
          to_physical_layer(&s);          /* none of the fields are used */
      }
  }
}
```

Sliding Window Protocols

- Remove assumption that one-way data transmission
  – duplex
- Error prone channel
- Finite speed (and buffer) network layer

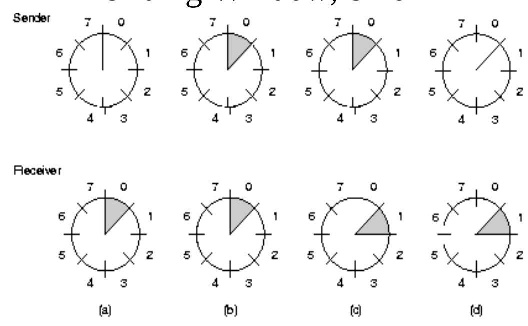
Two-Way Communication

- Seems efficient since acks already
- Have two kinds of frames (kind field)
  – Data
  – Ack (seq num of last correct frame)
- May want *data* with ack
  – delay a bit before sending data
  – *piggybacking* - add acks to data frames going other way
- How long to wait before just ack?

Sliding Window Protocols

- More than just 1 outstanding packet
  - "Window" of frames that are outstanding
- Sequence number is *n* bits, 2^{n-1}
- Sender has sending window
  - frames it can send (can change size)
- Receiver has receiving window
  - frames it can receive (always same size)
- Window sizes can differ
- Note, still passed to network layer in order!

Sliding Window, Size 1



1-Bit Sliding Window Protocol

```
void protocol4 (void)
{
  seq_nr next_frame_to_send;          /* 0 or 1 only */
  seq_nr frame_expected;              /* 0 or 1 only */
  frame r, s;                         /* scratch variables */
  packet buffer;                      /* current packet being sent */
  event_type event;

  next_frame_to_send = 0;             /* next frame on the outbound stream */
  frame_expected = 0;                 /* number of frame arriving frame expected */
  from_network_layer(&buffer);        /* fetch a packet from the network layer */
  s.info = buffer;                    /* prepare to send the initial frame */
  s.seq = next_frame_to_send;         /* insert sequence number into frame */
  s.ack = 1 – frame_expected;         /* piggybacked ack */
  to_physical_layer(&s);              /* transmit the frame */
  start_timer(s.seq);                 /* start the timer running */
```

(initialization)

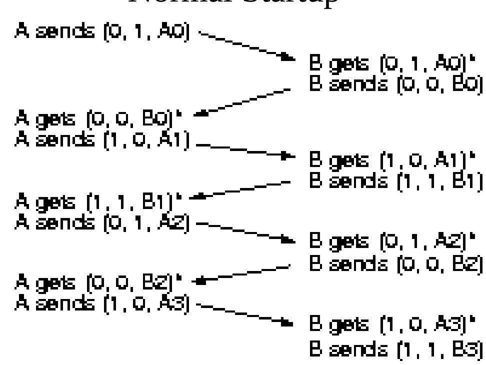
1-Bit Sliding Window Protocol

```
while (true) {
    wait_for_event(&event);          /* frame_arrival, cksum_err, or timeout */
    if (event == frame_arrival) { /* a frame has arrived undamaged. */
        from_physical_layer(&r);     /* go get it */

        if (r.seq == frame_expected) {
            /* Handle inbound frame stream. */
            to_network_layer(&r.info);    /* pass packet to network layer */
            inc(frame_expected);   /* invert sequence number expected next */
        }

        if (r.ack == next_frame_to_send) { /* handle outbound frame stream. */
            from_network_layer(&buffer);        /* fetch new pkt from network layer */
            inc(next_frame_to_send);    /* invert sender's sequence number */
        }
    }
    s.info = buffer;                 /* construct outbound frame */
    s.seq = next_frame_to_send;      /* insert sequence number into it */
    s.ack = 1 – frame_expected;      /* seq number of last received frame */
    to_physical_layer(&s);           /* transmit a frame */
    start_timer(s.seq);              /* start the timer running */
}
```

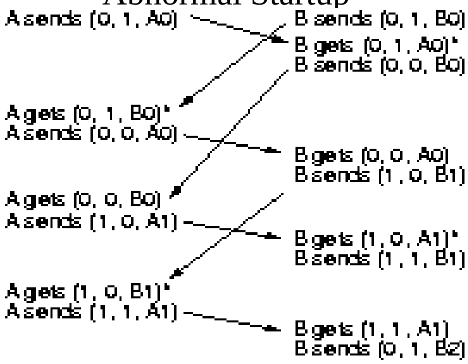
Does it Work?

- Consider A with a too-short time-out
- A sends: seq=0, ack = 1 over and over
- B gets 0, sets *frame_expected* to 1
  - will reject all 0 frames
- B sends A frame with seq=0, ack=0
  - eventually one makes it to A
- A gets ack, sets *next_frame_to_send* to 1
- Above scenario similar for lost/damaged frames or acknowledgements
- But … what about startup?

Normal Startup



Abnormal Startup



Transmission Factors

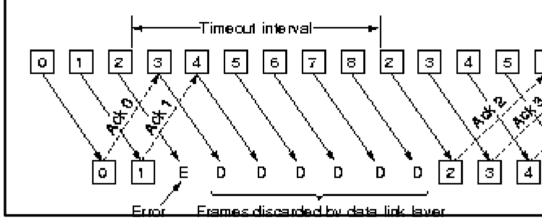
- Assume a satellite channel, 500 msec rt delay
  - super small ack's
- 50 kbps, sending 1000-bit frames
- t = 0, sending starts
- t = 20 msec frame sent
- t = 270 frame arrives
- t = 520 ack back at sender
- 20 / 520 about 4% utilization!
- All of: long delay, high bwidth, small frames
- Solution?

Allow Larger Window

- Satellite channel, 500 msec rt delay
- 50 kbps, sending 1000-bit frames
- Each frame takes 20 msec
  - 25 frames outstanding before first ack arrives
- Make window size 25
- Called *pipelining*
- (See p.211, protocol 5)
  - added enable/disable network layer
  - MAX_SEQ - 1 outstanding - timer per frame
- Frame in the middle is damaged?

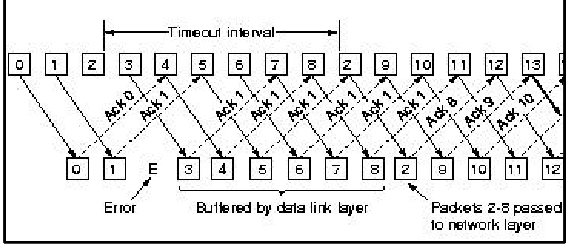
Go Back N

- If error, receiver discards all addtl frames
- Sender window fills, pipeline empties
- Sender times out, retransmits
- Waste of bandwidth if many errors



Selective Repeat

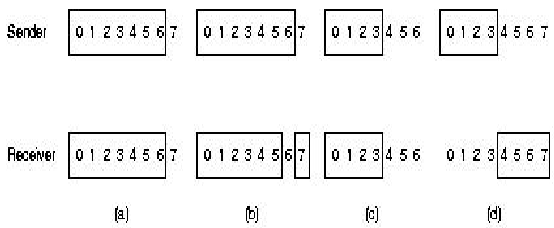
- Receiver stores all frames, waits for incorrect one
- Window size greater than 1



Latest and Greatest: Non-Sequential Receive

- Tanenbaum, Protocol 6
- Ack latest packet in sequence received
- Acks not always piggybacked
  - Protocol 5 will block until return data available
  - *start_ack_timer*
  - How long ack timeout relative to date timeout?
- Negative acknowledgement (NAK)
  - damaged frame arrives
  - non-expected frame arrives

Problem?



- Window size (MAX_SEQ) / 2
- How many buffers are needed? MAX_SEQ?

Closing Thoughts...

- If constant round-trip propagation delay
  - set timer just slightly higher than delay
- If variable round-trip propagation delay
  - small timer has unnecessary retransmissions
  - large has many periods of idle network
  - same is true of variable *processing* delay
- Constant, then “tight” timer
- Variable, then “loose” timer
  - NAKs can really help bandwidth efficiency

Topics

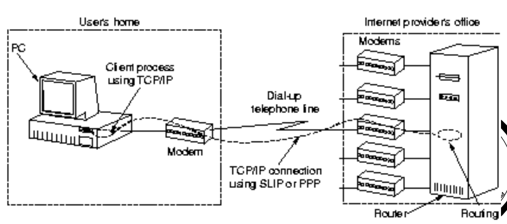
- Introduction ✓
- Errors ✓
- Protocols ✓
- Modeling ✗
  - complex specification and verification
- Examples ←

Examples

- HDLC ✗
  - IBM SNA
- Internet ←
  - SLIP
  - PPP
- ATM ←

The Internet

- Point-to-Point on leased lines between routers
- Home user to Internet Service Provider (ISP)
  - SLIP and PPP



Serial Line IP (SLIP)

- Character based, with special byte for frame
- Character stuffing
- 1984, newer versions do compression (CSLIP)
- No error correction or detection!
- No authentication
- Not a formally *approved* Internet standard

Point-to-Point Protocol (PPP)

- Bit-based frame
  - resorts to character based over a modem
- Line control: up, down, options
  - Link Control Protocol (LCP)
- Network control options
  - NCP (Network Control Protocol)
  - Service for: IP, IPX, AppleTalk …

ATM Layers

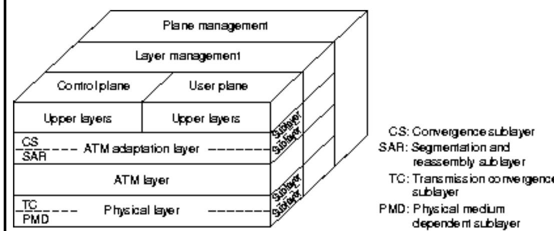


Fig. 1-30. The B-ISDN ATM reference model.

ATM Data Link Layer

- ATM Physical Layer is Data Link + Physical
  - Transmission Convergence (TC) sub-layer is like data link layer
- Checksums
  - on cell headers only, not payload
  - 5 byte header, includes 1 byte checksum
  - $x^8 + x^2 + x + 1$
  - called Header Error Control (HEC)

ATM Data Link Layer

- Why Header only?
  - Fiber, 99.64% of errors are 1 bit only
  - HEC corrects single-bit errors
- HEC used for *framing*, too
  - synchronization looking for 53 bytes
  - if out of synch, look for HEC
  - note, violates layers since must use header from above!
- Operation and Maintenance (AOM) cells
  - *idle cell* if no data to send
  - “pad” if receiver slower standard