Computer Networks
Data Link Layer

Paolo Costa
costa@cs.vu.nl
http://www.cs.vu.nl/~costa

Vrije Universiteit Amsterdam

(Version May 9, 2008)

Notes

Data Link Layer

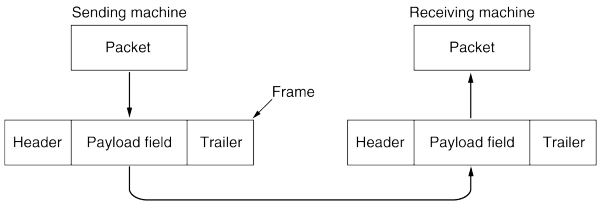
- Physical: Describes the transmission of raw bits in terms of mechanical and electrical issues.
- Data Link: Describes how a shared communication channel can be accessed, and how a data frame can be reliably transmitted.
- Network: Describes how routing is to be done. Mostly needed in subnets.
- Transport: The hardest one: generally offers connection-oriented as well as connectionless services, and varying degrees of reliability. This layer provides the actual network interface to applications.
- Application: Contains the stuff that users see: e-mail, remote logins, the Web's exchange protocol, etc.

Note

We'll just concentrate on transmission issues. Channel access is discussed in Chapter 4

Notes

Design Issues



1. Provide well-defined interface to network layer
2. Handle transmission errors
3. Regulate flow of data: get sender and receiver in the same pace

Notes

Basic Services

- Network layer passes a number of bits (frame) to the data link layer.
- Data link layer is responsible for transmitting the frame to the destination machine.
- Receiving layer passes received frame to its network layer.

Basic services commonly provided:

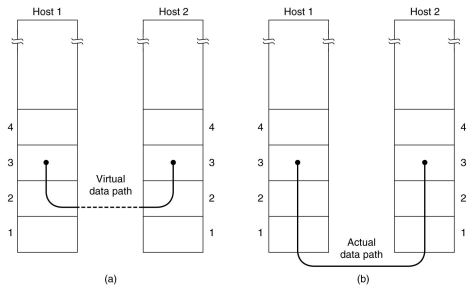
- unacknowledged connectionless service (LANs)
- acknowledged connectionless service (Wireless systems).
- acknowledged connection-oriented service (WANs).

Question

- Why are we so concerned about error-free frame transmissions? Can't the higher layers take care of that?
  It's not mandatory but it may improve efficiency (fine-grained recovery: frames vs. messages)

Notes

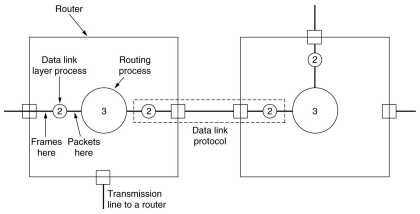
Transmission



- The actual transmissions follow the path of (b) but it is easier to think in terms of two data link process communicating using a data link protocol (a)

Notes

Routing

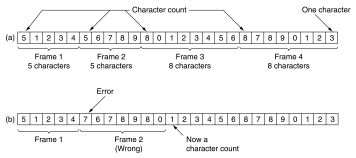


- The network layer consists of routing
  - they are connected through point=to-point links
- The router would really its packet to be sent correctly, guaranteed, and in the order it was issued.
- It is up to the data link to make unreliable connections look perfect, or at least, fairly good

Notes

Frames
Counting

- The physical layer doesn't do much: it just pumps bits from one end to the other.
- But things may go wrong
  - ⇒ the data link layer needs a means to do retransmissions.
- The unit of retransmission is a frame (which is just a fixed number of bits).
- Problem: How can we break up a bit stream into frames?
  - naive solution: counting

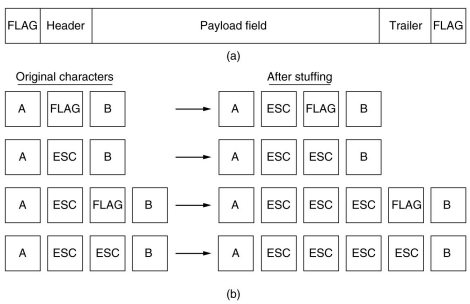


(a) Without errors (b) With errors

Notes

Frames
Byte Stuffing

- Byte stuffing: Mark the beginning and end of a *byte* frame with two special flag bytes – a special bit sequence (e.g. 01111110). If such bytes appear in the original frame, escape them:



Notes

Frames
Bit Stuffing

- Byte stuffing is closely tied to the use of 8-bit character
- Solution
  - ⇒ Bit stuffing: Escape the flag byte (e.g., 01111110) through an additional bit
    - whenever the sender's data link encounters five consecutive 1s in the data, it automatically stuffs a 0 bit

(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

(a) frame to send (b) frame transmitted over the wire (c) reconstructed frame

Notes

Error Correction and Detection

- Problem: Suppose something went wrong during frame transmission. How do we actually notice that something's wrong, and can it be corrected by the receiver?
- Definition: The Hamming distance between two frames $\vec{a}$ and $\vec{b}$ is the number of bits at the same position that differ.

Example

10001001 and 10110001 are at Hamming distance 3

1	0	0	0	1	0	0	1	
1	0	1	1	0	0	0	1	$\oplus \longleftarrow$ XOR
0	0	1	1	1	0	0	0	

- To detect d errors, you need a distance $d + 1$ code
  - d-single bit errors cannot change a valid codeword in another
- To correct d errors, you need a distance 2d+1 code
  - this way, even with $2d$ changes, the original codeword is still closer that any other codeword

Notes

Error Detection
Parity

- Add a bit to a bit string such that the total number of 1-bits is even (or odd)
  - e.g., (even parity) 1011010 ⇒ 10110100
  - (odd parity) 1011010 ⇒ 10110101
- The distance between two (legal) frames is at least $d = 2$.
  - any single-bit error produces a code word with the wrong parity
- We can detect a single error (i.e., $k = 1$ and $d = 2$)
  - remember that to detect d errors, you need a distance d+1 code

Notes

Error Correction
Example

- Consider a code with only four valid codewords:
  1. 0000000000
  2. 0000011111
  3. 1111100000
  4. 1111111111
- This code has distance 5
  - The XOR between any pair of the above codeword gives at least five 1 bits
  - ⇒ it means it can correct double errors
  - remember that to correct d errors, you need a distance 2d+1 code
- E.g., if the codeword 0000000111 is received, the receiver knows that the original must have been 0000011111
- If, however, a triple error changes 0000000000 into 0000000111, the error will not be corrected properly

Notes

Error Correction
Lower Bound

- Problem: we want to design a code with m message bits and r check bits to correct all single errors
- Each of the 2^m legal messages has n illegal codewords at distance 1
  - these are obtained by systematically change each of the n bits in the n-bit codeword
- Each of the 2^m legal messages require $n+1$ bit patterns dedicated to it
- The total number of bit pattern is 2^n

  $\Rightarrow (n+1)2^m \leq 2^n \quad \overset{n=m+r}{\Longrightarrow} \quad (m+r+1) \leq 2^r$
- Given m, it is possible to have a lower bound to the number r of check bits needed to correct single errors

Notes

Error Correction: Hamming
Definition

- The theoretical lower limit can be achieved using a method due to Hamming (1950)
  - Enumerate all bits in a codeword starting with bit 1 at the left
  - The bits at position 2^k ($k \geq 0$) are check bits, the rest (3,5,6,7,...) are filled with the m data bits
  - Every check bit is used as a parity bit for those positions to which it contributes

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11
1	X		X		X		X		X		X
2		X	X			X	X			X	X
4				X	X	X	X				
8								X	X	X	X

  - Rewrite every data bit as sums of power of 2
    - e.g., 7 = 1 + 2 + 4 and 11 = 1 + 2 + 8
  - A bit is checked by just those bits occurring in its expansion
    - e.g., bit 11 is checked by bit 1, 2, and 8

Notes

Error Correction: Hamming
Coding

	b1	**b2**	b3	**b4**	b5	b6	b7	**b8**	b9	b10	b11
1	**X**		X		X		X		X		X
2		**X**	X			X	X			X	X
4				**X**	X	X	X				
8								**X**	X	X	X
	1	**0**	1	**1**	1	0	0	**1**	0	0	1

- Check bits are used to even out the number of 1 bits they contribute to
  - e.g., assume the message is 1100001
  - Check bit at position 2, is used to even out the bits for positions 2, 3, 6, 7, 10, and 11

    $\Rightarrow$ $b2 = 0$ as the number of 1s is even (b3 and b11)
  - Other check bits are computed in the same way
  - The final codeword is 1011100101

Notes

Error Correction: Hamming
Decoding

- If a check bit at position p is wrong upon receipt, the receiver increments a counter v with p
- The value of v will, in the end, give the position of the wrong bit, which should then be swapped. E.g.:

	b1	**b2**	b3	**b4**	b5	b6	b7	**b8**	b9	b10	b11
1	**X**		X		X		X		X		X
2		**X**	X			X	X			X	X
4				**X**	X	X	X				
8								**X**	X	X	X
S:	**1**	**0**	1	**1**	1	0	0	**1**	0	0	1
R:	**1**	**0**	1	**1**	1	0	0	**1**	1	0	1
C:	**0**	**0**	1	**1**	1	0	0	**0**	1	0	1
F:	**1**	**0**	1	**1**	1	0	0	**1**	0	0	1

S: string sent
R: string received
C: string corrected on the check bits: #1 and #8 corrected $\Rightarrow$ bit #9 is wrong ($v = 1 + 9$)
F: final result after correction

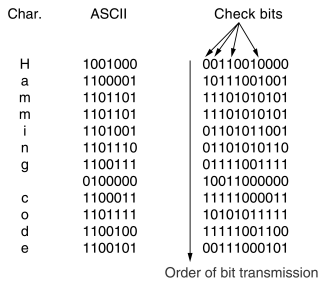
Notes

Error Correction: Hamming
Burst Errors

- Hamming codes can only detect single errors
- However, there is a trick to correct burst errors

- A sequence of k codewords are arranged as a matrix, one codeword per row
- Data are transmitted one column at a time
- If a burst error of length k occurs, at most 1 bit per codeword is affected
  - Hamming code can correct one error per codeword

  $\Rightarrow$ the entire block can be restored



Notes

Error Detection
CRC

- Error correcting codes are simply too expensive
  - e.g., for 10,000 bit blocks, 10 check bits are needed

  $\Rightarrow$ only use error detection combined with retransmissions
- Cyclic Redundancy Check (CRC), a.k.a. polynomial code, is based upon treating bit strings as polynomials with coefficient of 0 and 1 only
  - a k-bit frame is regarded as the coefficient list for a polynomial with k terms, from x^{k-1} to x^0
  - e.g., a 6-bit block

$$M(x) = m_{k-1}x^{k-1} + \ldots + m_1x^1 + m_0x^0$$

$$\begin{aligned} 110001 &\Rightarrow 1\dot{x}^5 + 1\dot{x}^4 + 0\dot{x}^3 + 0\dot{x}^2 + 0\dot{x}^1 + 1\dot{x}^0 \\ &= x^5 + x^4 + 1 \end{aligned}$$

  - Polynomial arithmetic is done in modulo 2, according to the rule of algebraic field theory

Notes

Polynomial Arithmetic
Addition and Subtraction

- Since the coefficients are constrained to a single bit, any math operation on CRC polynomials must map the coefficients of the result to either 0 or 1
- Addition

$$(x^3 + x) + (x + 1) = x^3 + 2x + 1 \equiv x^3 + 1$$

  - note that $2x$ becomes zero because addition of coefficient is done modulo 2

$$2x = x + x = (1 + 1)x = 0x = 0$$

  - analogous to exclusive OR (XOR)

$$1010 \oplus 0011 \rightarrow 1001$$

- Subtraction
  - works like addition (only absolute values are used)

$$(x^4 + x^2 + 1) - (x + 1) = x^4 + x^2 - x \equiv x^4 + x^2 + x$$

Notes

Polynomial Arithmetic
Multiplication and Division

- Multiplication
  - It follows the rule of traditional polynomials but, again, addition of coefficient is done module 2

$$(x^2 + x)(x + 1) = x^3 + x^2 + x^2 + 1 = x^3 + 2x^2 + 1 \equiv x^3 + 1$$

- Division
  - we can also divide polynomials mod 2 and find the quotient and remainder

$$\frac{x^3 + x^2 + x}{x + 1} = (x^2 + 1) - \frac{1}{x - 1}$$

  - or in other words:

$$(x^3 + x^2 + x) = (x^2 + 1)(x + 1) - 1$$

Notes

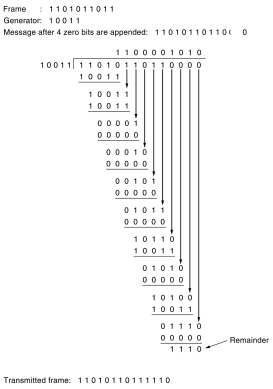
CRC
Algorithm

- Let's go back our problem
- Sender and receiver agree upon a generator polynomial $G(x)$
  - both the high- and low-order bits of the generator must be 1
  - the frame $M(x)$ must be longer than the generator ($m > r$)
- Idea: append a checksum to the end of the frame in such a way that the polynomial obtained is divisible (i.e., no remainder) by $G(x)$
  - when the receivers gets the checksummed frame, it tries dividing it by $G(x)$
  - if there is a remainder, there has been a transmission error
- Algorithm
  1. append r zero bits to the low order of the frame
     - now it contains $m + r$ bits and corresponds to $x^r M(x)$
  2. divide $x^r M(x)$ by $G(x)$
  3. subtract (mod 2) the remainder from $x^r M(x)$. The result is the checksummed frame $T(x)$ to be transmitted

Notes

CRC
Division Example

1. Check the input bit above the leftmost divisor
   - If it is 0, do nothing
   - If it is 1, the divisor is XORed into (i.e., subtracted from) the input
2. Move the divisor to the right by one bit
3. The process is repeated until the divisor reaches the right-hand end of the input row
4. The left bits are the remainder of the division to be appended to the frame



Notes

CRC
Decoding

- If a transmission error occurs, instead of the original polynomial $T(x)$, $T(x) + E(x)$ arrives
  - each 1 bit in $E(x)$ corresponds to a bit that has been inverted
  - if there are k 1 bits in $E(x)$, k single bit errors have occurred
  - $E(x)$ is characterized by an initial 1, a mixture of 0s and 1s, and a final 1
- The receiver divided the received frame by $G(x)$, i.e., $[T(x) + E(x)]/G(x)$ and computes the remainder
  - in case of error, the remainder will be $E(x)/G(x)$
- If there is a single-bit error, $E(x) = x^i$ (being i the position of the wrong bit)
  - if $G(x)$ contains two or more terms, it will never divide $E(x) \Rightarrow$ all single-bit errors will be detected
  - the simplest error-detection system, the parity bit, is in fact a trivial CRC: it uses the two-bit-long divisor 11

Notes

CRC
Burst Error

- By carefully choosing $G(x)$, burst errors can be correctly detected
  - a burst error is a contiguous sequence of bits, containing wrong bits
- It can be shown, that a polynomial code with r check bits, will detect all bursts errors of length $\leq r$
- If the burst length is $r + 1$, the probability of accepting an incorrect frame is $\frac{1}{2^{r-1}}$
- It can also be shown that for bursts longer than $r + 1$ bit, such probability becomes $\frac{1}{2^r}$, assuming that all bit patterns are equally likely (not always true)
- Certain polynomial have become standards
  - e.g., the one used in IEEE 802 (Wi-Fi) is:
    $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$
  - it detects all bursts of length 32 or less and all bursts affecting an odd number of bits

Notes

CRC
Applications

- CRC is not used by the data link layer but also by applications to check the correctness of a given file
- e.g., WinZip

Notes

Data Link Layer Protocols

- Concentrated on design aspects and error control
  - frame size (stuffing)
  - error correction (Hamming Code)
  - error detection (CRC)
- Now: basic protocols and real-world examples

Some basic assumptions:

- We have a machine *A* that wants to send data to machine *B*
- There is always enough data for *A* to send
- There is a well-defined interface to the network layer, and to the physical layer
  - `void from_network_layer(packet *)`
  - `void to_network_layer(packet *)`
  - `void from_physical_layer(packet *)`
  - `void to_physical_layer(packet *)`
- The receiver generally waits for an event to happen by calling
  - `void wait_for_event(event_type *event)`

Notes

Unrestricted Simplex Protocol

```
1  typedef enum {false, true} boolean;
2  typedef unsigned int seq_nr;
3  typedef struct {unsigned char data[MAX_PKT];} packet;
4  typedef enum {data, ack, nak} frame_kind;
5
6  typedef struct {
7    frame_kind kind;  /* what kind of a frame is it? */
8    seq_nr seq;       /* sequence number */
9    seq_nr ack;       /* acknowledgment number */
10   packet info;      /* the network layer packet */
11 } frame;
12
13 typedef enum {frame_arrival} event_type;
14
15 void sender1(void){
16   frame s; packet buffer;
17   while (true) {
18         from_network_layer(&buffer);
19         s.info = buffer;
20         to_physical_layer(&s);
21   }
22 }
23
24 void receiver1(void){
25   frame r; event_type event;
26   while (true) {
27         wait_for_event(&event);
28         from_physical_layer(&r);
29         to_network_layer(&r.info);
30   }
31 }
```


Notes

Unrestricted Simplex Protocol
Issues

Question

- What are some of the underlying assumptions here? How does the flow control manifest itself?
  Underlying assumptions are error-free transmission, and sender-receiver are in pace.

- Issue: Fast senders would make receivers collapse

Question

- Would a simple delay on sender's side fix the problem ?
  Not really! It's hard to predict when receiver's congestion may occur.

- Assuming always the worst-case behavior would be highly inefficient

Notes

Simplex Stop-and-Wait

```
1  typedef enum {frame_arrival} event_type;
2  #include "protocol.h"
3
4  void sender2(void){
5    frame s; packet buffer;
6    event_type event;
7    while (true) {
8          from_network_layer(&buffer);
9          s.info = buffer;
10         to_physical_layer(\&s);
11         wait_for_event(&event);
12   }
13 }
14
15 void receiver2(void){
16   frame r, s; event_type event;
17   while (true) {
18         wait_for_event(&event);
19         from_physical_layer(\&r);
20         to_network_layer(&r.info);
21         to_physical_layer(&s);
22   }
23 }
```

- Here the receiver provide the feedback to the sender
  - it sends a little dummy packet (acknowledgment) back to the sender
- After having sent a frame, the sender is required to wait for the ack

Notes

Simplex Protocol for Noisy Channel

Question

- What are the assumptions in this case? We are assuming only error-free transmission.

- Ok, so assume that damaged frames can be detected

Question

- Idea: send the ack only upon correct receipt! Works ? No, because also the ack can be lost.

Question

- Second idea: let the sender use a timer by which it simply retransmits unacknowledged frames after some time! Works ? No again! Because the data link layer cannot detect duplicate transmissions.

- Let's use sequence numbers!
  - Problem: sequence number cannot go on for ever and they wast space
  - Solution: no worries, we need just two (0 & 1)

Notes

Simplex Protocol for Noisy Channel
Sender

- `next_frame_to_send` indicates which is the last message sent
- A timer is started after sending a message
- Only if the correct ack is received, the next message is sent

```
1  #define MAX_SEQ 1
2  typedef enum {frame_arrival, cksum_err, timeout} event_type;
3  void sender3(void) {
4    int        next_frame_to_send;
5    frame      s;
6    packet     buffer;
7    event_type event;
8
9    next_frame_to_send = 0;
10   from_network_layer(&buffer);
11   while (true) {
12     s.info = buffer;
13     s.seq = next_frame_to_send;
14     to_physical_layer(&s);
15     start_timer(s.seq);
16     wait_for_event(&event);
17
18     if (event == frame_arrival) {
19       from_physical_layer(&s);
20       if (s.ack == next_frame_to_send) {
21         stop_timer(s.ack);
22         from_network_layer(&buffer);
23         inc(next_frame_to_send);
24   } } } }
```


Notes

Simplex Protocol for Noisy Channel
Receiver

- `frame_expected` enables filtering out duplicate
- Its value is increased each time a frame is delivered to the network layer
  - ⇒ only two values (0 and 1) are needed

```
1  void receiver3(void) {
2
3    seq_nr frame_expected; frame r, s; event_type event;
4
5    frame_expected = 0;
6    while (true) {
7
8      wait_for_event(&event);
9      if(event == frame_arrival) {
10
11       from_physical_layer(&r);
12       if (r.seq == frame_expected) {
13
14         to_network_layer(&r.info);
15         inc(frame_expected);
16       }
17       s.ack = 1 - frame_expected;
18       to_physical_layer(&s);
19     }
20   }
21 }
```


Notes

From Simplex to Duplex

- Problem: We want to allow symmetric frame transmission between two communicating parties, rather than transmission in one direction.
  - don't waste channels, so use the same channel for duplex communication.
- Solution: Just transmit frames, but distinguish between data, acknowledgments (acks), and possibly negative acks (nacks) in the frame's type field.
- Idea: If the other party is going to send data as well, it might as well send acknowledgments along with its data frames
  ⇒ piggybacking.

Question

- What's good and bad about piggybacking?
  Good: save bandwidth.
  Bad: poor performance with irregular transmission rate.

Notes

Sliding Windows

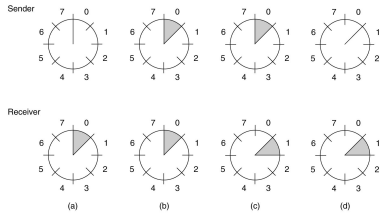
- Principle: Rather than just sending a single frame at a time, permit the sender to transmit a set of frames, called the sending window.
  - a frame is removed from the sending window iff it has been acknowledged.
  - the receiver has a receiving window containing frames it is permitted to receive.
- A damaged frame is kept in the receiving window until a correct version is received. Also, frame N is kept in the window until frame N-1 has been received.
  - if both window sizes equal one, we are dealing with the stop-and-wait protocols.

Question

- Is there a relationship between window size, transmission rate, and propagation delay ? If the propagation delay is high and the window size is small, transmission rate decreases dramatically because the sender while waiting for acks cannot send new messages

Notes

Window Size & Sequence Number



(a) Initially
(b) After sending frame #1.
(c) After receiving frame #1.
(d) After receiving ack for frame #1

Question

- How would you interpret the shaded areas?
  Shaded area is to-be-acknowledged (at sender) and to-be-received next (at receiver).

Notes

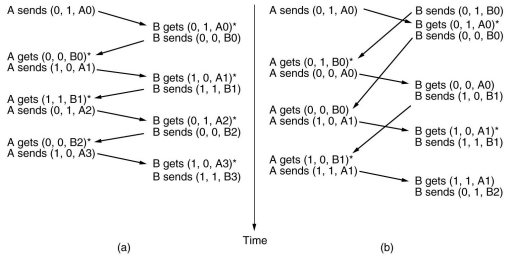
1-Bit Sliding Window

```
1  void protocol4 (void) {
2    seq_nr next_frame_to_send, frame_expected;
3    frame r, s; packet buffer;
4    event_type event;
5
6    next_frame_to_send = 0; frame_expected = 0;
7    from_network_layer(&buffer);
8    s.info = buffer;
9    s.seq  = next_frame_to_send;
10   s.ack  = 1 - frame_expected;
11   to_physical_layer(&s); start_timer(s.seq);
12
13   while (true) {
14     wait_for_event(&event);
15     if (event == frame_arrival) {
16       from_physical_layer(&r);
17       if (r.seq == frame_expected){
18          to_network_layer(&r.info);
19          inc(frame_expected);}
20
21       if (r.ack == next_frame_to_send){
22       from_network_layer(&buffer);
23       inc(next_frame_to_send);}
24
25     }
26     s.info = buffer;
27     s.seq = next_frame_to_send;
28     s.ack = 1 - frame_expected;
29     to_physical_layer(&s); start_timer(s.seq);
30   }
31 }
```

- `next_frame_to_send` tells which frame the sender is trying to send
- `frame_expected` tells which frames the receiver is expecting
- If the received frame is the one expected, the frame is passed to the network layer
- If the received ack is the one expected, the next frame is sent

Notes

1-Bit Sliding Window
Simultaneous Transmissions



The notation is (seq, ack, packet number). * means: passed to network layer

- All things go well, but behavior is a bit strange when *A* and *B* transmit simultaneously.
- We are transmitting more than once, just because the two senders are more or less out of sync.

Notes

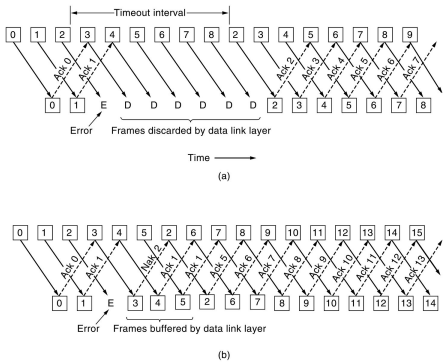
Error Control

Problem: What should the receiver do if a frame is damaged ?

- Go back n
  Simply request retransmission of all frames starting from frame #N. If any other frames had been received in the meantime (and stored in the receiver's window), they'll just be ignored
- Selective Repeat
  Request just retransmission of the damaged frame, and wait until it comes in before delivering any frames after that

Notes

Error Control
Example



(a) Go back n (b) Selective Repeat.

Notes

Error Control
Pros & Cons

- Go-back-N is really simple: the sender keeps a frame in its window until it is acknowledged.
  - if the window is full, the network layer is not allowed to submit new packets.
  - the receiver hardly needs to keep an account on what happens: if a frame is damaged, its successors in the receive window are ignored.
  - it is equivalent to a receive window of size 1
- Selective repeat seems to do better because frames aren't discarded, but the administration is much harder
  - it also requires large amounts of memory if the window is large
- Trade-off between bandwidth and data link layer buffer space

Notes

Go-back-n (1/2)

```
1  #define MAX_SEQ 7 /* should be 2^n - 1 */
2  typedef enum {frame_arrival, cksum_err,
3    timeout, network_layer_ready} event_type;
4  #include "protocol.h"
5
6  static boolean between(seq_nr a, seq_nr b, seq_nr c) {
7    /* Return TRUE iff a <= b < c  (cyclic) */
8    ...
9  }
10
11 static void send_data(
12  seq_nr frame_nr, seq_nr frame_expected, packet buffer[]){
13   frame s;
14   s.info = buffer[frame_nr]; s.seq = frame_nr;
15   s.ack  = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
16   to_physical_layer(&s); start_timer(frame_nr);
17 }
18
19 void protocol5(void) {
20   seq_nr next_frame_to_send, ack_expected, frame_expected;
21   frame r;
22   packet buffer[MAX_SEQ + 1];
23   seq_nr nbuffered, i;
24   event_type event;
25
26   enable_network_layer();
27   ack_expected = 0; next_frame_to_send = 0; frame_expected = 0;
28   nbuffered = 0;
```

- `enable_network_layer` prevents network layer to input more frames when there are `MAX_SEQ` unacknowledged frames

Notes

Go-back-n (2/2)

```
1    while (true) {
2      wait_for_event(&event);
3      switch(event) {
4      case network_layer_ready:
5        from_network_layer(&buffer[next_frame_to_send]);
6        nbuffered = nbuffered + 1;
7        send_data(next_frame_to_send, frame_expected, buffer);
8        inc(next_frame_to_send);
9        break;
10     case frame_arrival:
11       from_physical_layer(&r);
12       if (r.seq == frame_expected) {
13         to_network_layer(&r.info);
14         inc(frame_expected); }
15       while (between(ack_expected, r.ack, next_frame_to_send)) {
16         nbuffered = nbuffered - 1;
17         stop_timer(ack_expected);
18         inc(ack_expected); }
19       break;
20     case cksum_err: break; /* just ignore bad frames */
21     case timeout: /* trouble; retransmit outstanding frames */
22       next_frame_to_send = ack_expected;
23       for (i = 1; i <= nbuffered; i++) {
24         send_data(next_frame_to_send, frame_expected, buffer);
25         inc(next_frame_to_send); }
26     }
27     if (nbuffered < MAX_SEQ)
28           enable_network_layer();
29     else
30           disable_network_layer();
31   } }
```


Notes

Go-back-n
Comments

- An ack for frame n, automatically acknowledges also frames n-1, n-2, etc.
- Note that a maximum of `MAX_SEQ` frames and not `MAX_SEQ` + 1 frames may be outstanding
  - → `packet buffer[MAX_SEQ+1]`
  - → `if nbuffered < MAX_SEQ`
  - The reason is the following:
    1. The sender sends frames 0 through 7
    2. An ack for frame 7 comes back to the sender
    3. The sender sends other eight frames with sequence number 0 through 7
    4. Now another piggybacked ack for frame 7 comes in
- Problem: Did all eight frames belonging to the second batch arrive correctly or did all get lost ?
  - the sender has no way to detect it
- Solution Restrict the maximum number of outstanding frames to `MAX_SEQ`

Notes

Selective Repeat

- Go-back-n works well if errors are rare, but if the line is poor, it wastes a lot of bandwidth
- An alternative strategy is to allow receiver to accept and buffer the frames following a damaged or lost one
  - both sender and receiver maintain a window larger than 1
  - whenever a frame arrives, its sequence number is checked by the function `between` to see if it falls within the window
  - however, it must be kept within the data link layer and not passed to the network layer until all lower numbered frames are received

Notes

Selective Repeat (1/3)

```
1  static boolean between(seq_nr a, seq_nr b, seq_nr c) {...}
2  static void send_frame(frame_kind fk, seq_nr frame_nr,
3                         seq_nr frame_expected, packet buffer[]){
4    frame s; s.kind = fk;
5    if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
6    s.seq = frame_nr;
7    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
8    if (fk == nak) no_nak = false;
9    to_physical_layer(&s);
10   if (fk == data) start_timer(frame_nr % NR_BUFS);
11   stop_ack_timer();
12 }
13 void protocol6(void){
14   seq_nr ack_expected, next_frame_to_send, frame_expected;
15   seq_nr nbuffered, too_far; event_type event;int i; frame r;
16   packet out_buf[NR_BUFS], in_buf[NR_BUFS];
17   boolean arrived[NR_BUFS];
18
19   enable_network_layer();
20   ack_expected = 0; next_frame_to_send = 0; frame_expected = 0;
21   too_far = NR_BUFS; nbuffered = 0;
22   for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
23   while (true) {
24     wait_for_event(&event);
25     switch(event) {
26     case network_layer_ready:
27       nbuffered = nbuffered + 1;
28       from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]);
29       send_frame(data,next_frame_to_send,frame_expected,out_buf);
30       inc(next_frame_to_send);
31       break;
```


Notes

Example: Selective Repeat (2/3)

```
1        case frame_arrival:
2          from_physical_layer(&r);
3          if (r.kind == data) {
4            if ((r.seq != frame_expected) && no_nak)
5              send_frame(nak, 0, frame_expected, out_buf);
6            else start_ack_timer();
7            if (between(frame_expected, r.seq, too_far) &&
8                if(arrived[r.seq % NR_BUFS] == false)) {
9              arrived[r.seq % NR_BUFS] = true;
10             in_buf[r.seq % NR_BUFS] = r.info;
11             while (arrived[frame_expected % NR_BUFS]) {
12               to_network_layer(&in_buf[frame_expected % NR_BUFS]);
13               no_nak = true;
14               arrived[frame_expected % NR_BUFS] = false;
15               inc(frame_expected);
16               inc(too_far);
17               start_ack_timer(); }
18           }
19         }
20         if ((r.kind == nak) &&
21         if between(ack_expected, (r.ack+1) % (MAX_SEQ+1)
22                    next_frame_to_send))
23           send_frame(data, (r.ack+1) % (MAX_SEQ + 1),
24                      frame_expected,out_buf);
25
26         while (between(ack_expected, r.ack, next_frame_to_send)) {
27           nbuffered = nbuffered - 1;
28           stop_timer(ack_expected % NR_BUFS);
29           inc(ack_expected);
30         }
31         break;
```


Notes

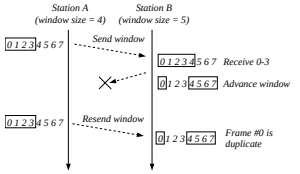
Example: Selective Repeat (3/3)

```
1        case cksum_err:
2          if (no_nak) send_frame(nak, 0, frame_expected, out_buf);
3          break;
4
5        case timeout:
6          send_frame(data, oldest_frame, frame_expected, out_buf);
7          break;
8
9        case ack_timeout:
10         send_frame(ack,0,frame_expected, out_buf);
11       }
12       if (nbuffered < NR_BUFS) enable_network_layer();
13       else disable_network_layer();
14     }
15   }
```


Notes

Selective Repeat
Comments

- Frames need not be received in order,
  - we may have an undamaged frame N, while still waiting for an undamaged version of $N-1$.
- If the receiver delivers all frames in its window just after sending an ack for the entire window, we may have a serious problem:



- Solution: we must avoid overlapping send and receive windows
  ⇒ the highest sequence number must be at least twice the window size.

Notes

Data Link Protocols
Demo

http://www.csi.uottawa.ca/~elsaddik/abedweb/applets/Applets/Sliding_Window/sliding-window/index.html

Notes

Data Link Layer Protocols

Now let's take a look how point-to-point connections are supported in, for example, the Internet.
Recall:

- The data link layer is responsible for transmitting frames from sender to receiver.
- It can use only the physical layer, which supports only transmission of a bit at a time.
- The DLL has to take into account that transmission errors may occur
  - ⇒ error control (ACKs, NACKs, checksums, etc.)
- The DLL has to take into account that sender and receiver may operate at different speeds
  - ⇒ flow control (windows, frame numbers, etc.)

Notes

High-Level Data Link Control

- HDLC: A pretty old, but widely used protocol for point-to-point connections. Is bit-oriented.

Bits	8	8	8	⩾ 0	16	8
	0 1 1 1 1 1 1 0	Address	Control	Data	Checksum	0 1 1 1 1 1 1 0

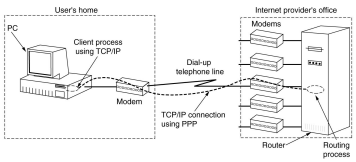
- HDLC uses a sliding window protocol with 3-bit sequencing
- The control field is used to distinguish different kinds of frames
  - contains sequence numbers, acks, nacks, etc.

Question
- What do we need the address field for? Useful on line with multiple terminals

Notes

Internet Point-to-Point Connections

- This is what may happen when you have an Internet connection to a provider:



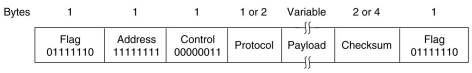
- We'd like to use the Internet protocol stack at our home (e.g., ADSL)
  - the bottom line is that we'll have to transfer IP (network) packets across our telephone line
- Issues:
  - how can we embed IP packets into frames ?
  - how can we unpacked at the other end ?

Notes

PPP: Point-to-Point Protocol

- PPP is the data link protocol for point-to-point connections with respect to the Internet (home-ISP and router-to-router):
  - Proper framing, i.e. the start and end of a frame can be unambiguously detected.
  - A separate protocol for controlling the line (setup, testing, negotiating options, and tear-down) (LCP)
  - Supports many different network layer protocols, not just IP.
  - No need for fixed network addresses.

The default frame:



Address	(no data link addresses)	*Control*	no seq. number (no reliable transmission)
Protocol	LCP / NCP / IP / ...	*Payload*	(up to a negotiated maximum)
Checksum	(usually 2 bytes)		

Notes

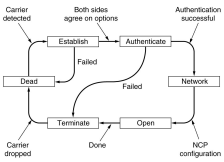
PPP
Connection Setup

Suppose you want to set up a true Internet connection to your provider.

1. Set up a physical connection through your modem
2. Your PC starts sending a number of Link Control Packets (LCP) to negotiate the kind of PPP connection you want.
   - the maximum payload size in data frames
   - do authentication (e.g. ask for a password)
   - monitor the quality of the link (e.g. how many frames didn't come through).
   - compress headers (useful for slow links between fast computers)
3. Then, we negotiate network layer stuff, like getting an IP address that the provider's router can use to forward packets to you.

Notes

PPP
Example



Question

- If an IP address is dynamically assigned, who does the assignment ? The provider

Question

- If an IP address is dynamically assigned, Can someone else ever send you data (they don't know your address, do they?) We need to contact them first (our address is included in the request).

Notes

Notes