DATA LINK LAYER DESIGN ISSUES

The data link layer uses the services of the physical layer to send and receive bits over communication channels. It has a number of functions, including:

1. Providing a well-defined service interface to the network layer.
2. Dealing with transmission errors.
3. Regulating the flow of data so that slow receivers are not swampedby fast senders.

*To accomplish these goals, the data link layer takes the packets it gets from the network layer and encapsulates them into **frames** for transmission.

*Each frame contains a frame header, a payload field for holding the packet, and a frame trailer, as illustrated in Fig. 3-1. Frame management forms the heart of what the data link layer does.

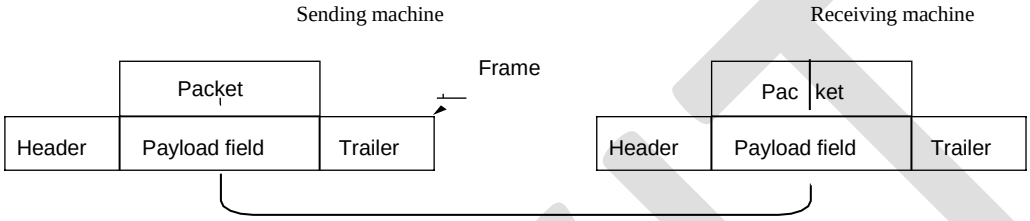


Figure 3-1. Relationship between packets and frames

*In fact, in many networks, these functions are found mostly in the upper layers, with the data link layer doing the minimal job that is ‘‘good enough.’’ However, no matter where they are found, the principles are pretty much the same.

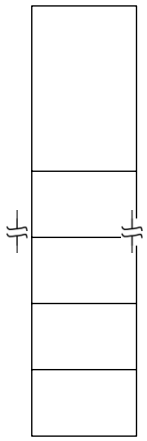
*.They often show up in their simplest and purest forms in the data link layer, making this a good place to examine them in detail.

❖ Services Provided to the Network Layer

- The function of the data link layer is to provide services to the network layer
- The principal service is transferring data from the network layer on the source ma- chine to the network layer on the destination machine
- On the source machine is an entity, call it a process, in the network layer that hands some bits to the data link layer for transmission to the destination.
- The job of the data link layer is to transmit the bits to the destination machine so they can be handed over to the net-work layer there, as shown in Fig. 3-2(a).

The data link layer can be designed to offer various services. The actual services that are offered vary from protocol to protocol. Three reasonable possibilities that we will consider in turn are:

1. Unacknowledged connectionless service.
2. Acknowledged connectionless service.
3. Acknowledged connection-oriented service.



❖ Framing

- Breaking up the bit stream into frames is more difficult than it at first appears. A good design must make it easy for a receiver to find the start of new frames while using little of the channel bandwidth. We will look at four methods:

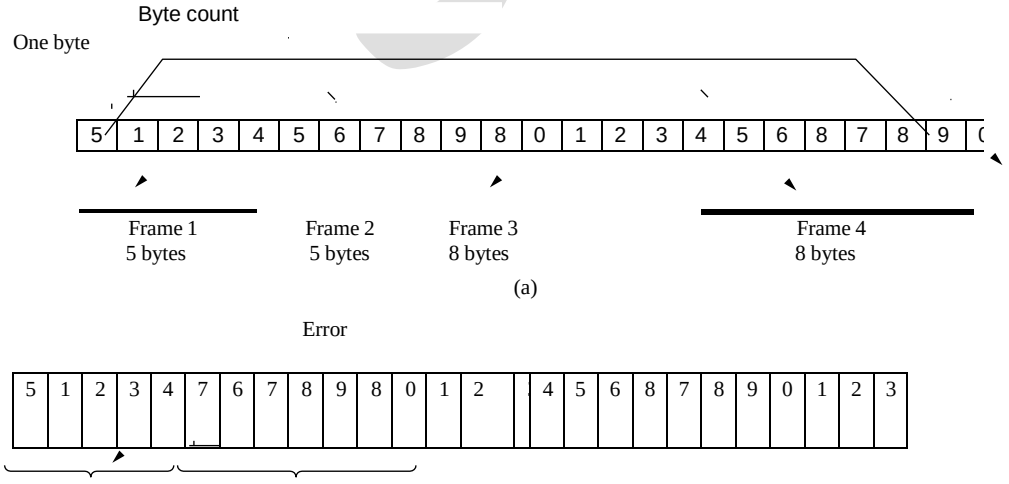
(a) Byte count.

(b) Flag bytes with byte stuffing.

(c) Flag bits with bit stuffing.

(d) Physical layer coding violations.

- The usual approach is for the data link layer to break up the bit stream into discrete frames, compute a short token called a checksum for each frame, and in- clude the checksum in the frame when it is transmitted. (Checksum algorithms will be discussed later in this chapter.)
- When a frame arrives at the destination, the checksum is recomputed. If the newly computed checksum is different from the one contained in the frame, the data link layer knows that an error has oc- curred and takes steps to deal with it (e.g., discarding the bad frame and possibly also sending back an error report).



Frame 1 Frame 2 Now a byte count

(Wrong)

(b)

- The first framing method uses a field in the header to specify the number of bytes in the frame. When the data link layer at the destination sees the byte count, it knows how many bytes follow and hence where the end of the frame is. This technique is shown in Fig. 3-3(a)

- The trouble with this algorithm is that the count can be garbled by a transmis- sion error. For example, if the byte count of 5 in the second frame of Fig. 3-3(b) becomes a 7 due to a single bit flip, the destination will get out of synchroniza- tion. It will then be unable to locate the correct start of the next frame.

- The second framing method gets around the problem of resynchronization after an error by having each frame start and end with special bytes. Often the same byte, called a flag byte, is used as both the starting and ending delimiter. This byte is shown in Fig. 3-4(a) as FLAG

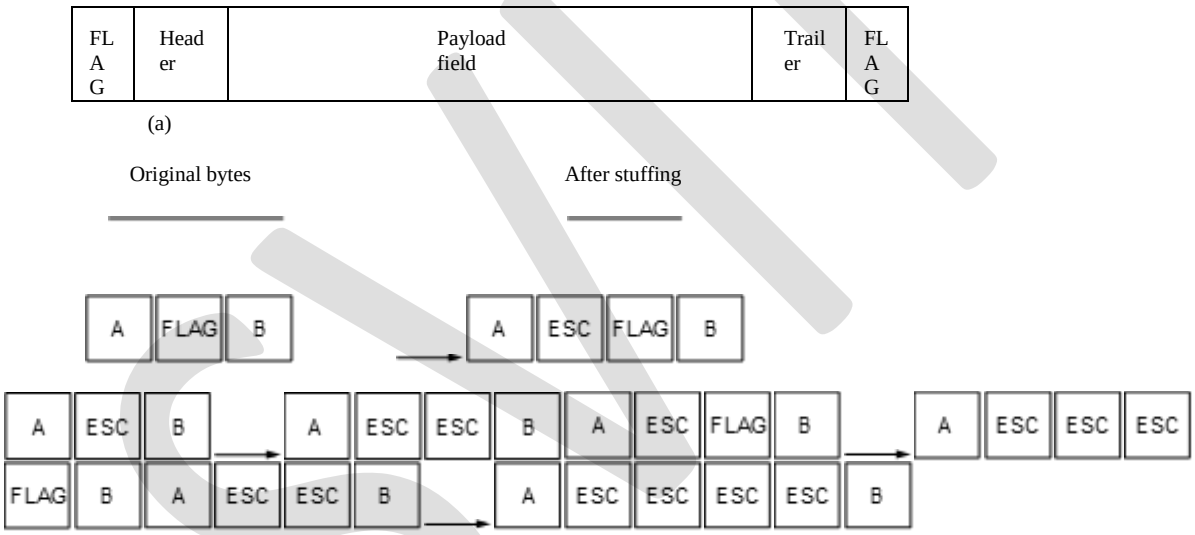


Figure 3-4. (a) A frame delimited by flag bytes. (b) Four examples of byte se- quences before and after byte stuffing.

- The data link layer on the receiving end re- moves the escape bytes before giving the data to the network layer. This techni- que is called **byte stuffing**.

- The byte-stuffing scheme depicted in Fig. 3-4 is a slight simplification of the one used in **PPP** (**Point-to-Point Protocol**), which is used to carry packets over communications links

Framing can be also be done at the bit level, so frames can contain an arbitrary number of bits made up of units of any size. It was developed for the once very popular **HDLC** (**High- level Data Link Control**) protocol.

- This **bit stuffing** is anal- ogous to byte stuffing, in which an escape byte is stuffed into the

outgoing charac- ter stream before a flag byte in the data

- When the receiver sees five consecutive incoming 1 bits, followed by a 0 bit, it automatically destuffs (i.e., deletes) the 0 bit. Just as byte stuffing is completely transparent to the network layer in both computers, so is bit stuffing. If the user data contain the flag pattern, 01111110, this flag is transmitted as 011111010 but stored in the receiver's memory as 01111110. Figure 3-5 gives an example of bit stuffing.

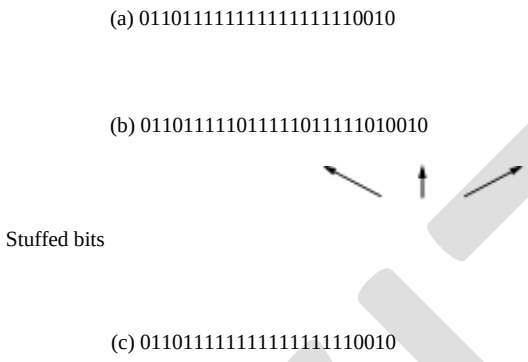


Figure 3-5. Bit stuffing. (a) The original data. (b) The data as they appear on the line. (c) The data as they are stored in the receiver's memory after destuf- fing.

- Many data link protocols use a combination of these methods for safety. A common pattern used for Ethernet and 802.11 is to have a frame begin with a well-defined pattern called a **preamble**.

❖ Error Control

- The usual way to ensure reliable delivery is to provide the sender with some feedback about what is happening at the other end of the line, Typically, the pro- tocol calls for the receiver to send back special control frames bearing positive or negative acknowledgements about the incoming frames

- If the sender receives a positive acknowledgement about a frame, it knows the frame has arrived safely, On the other hand, a negative acknowledgement means that something has gone wrong and the frame must be transmitted again

- An additional complication comes from the possibility that hardware troubles may cause a frame to vanish completely (e.g., in a noise burst). In this case, the receiver will not react at all, since it has no reason to react. Similarly, if the ac- knowledgement frame is lost, the sender will not know how to proceed

- It should be clear that a protocol in which the sender transmits a frame and then waits for an acknowledgement, positive or negative, will hang forever if a frame is ever lost due to, for example, malfunctioning hardware or a faulty communication channel

- This possibility is dealt with by introducing timers into the data link layer. When the sender

transmits a frame, it generally also starts a timer, However, if either the frame or the acknowledgement is lost, the timer will go off, alerting the sender to a potential problem

- The obvious solution is to just transmit the frame again. However, when frames may be transmitted multiple times there is a danger that the receiver will accept the same frame two or more times and pass it to the network layer more than once. To prevent this from hap- pening, it is generally necessary to assign sequence numbers to outgoing frames, so that the receiver can distinguish retransmissions from originals.

❖ Flow Control

- The design issue that occurs in the data link layer (and higher layers as well) is what to do with a sender that systematically wants to transmit frames faster than the receiver can accept them, is situation can occur when the sender is running on a fast, powerful computer and the receiver is running on a slow, low-end machine
- A common situation is when a smart phone requests a Web page from a far more powerful server, which then turns on the fire hose and blasts the data at the poor helpless phone until it is completely swamped. Even if the transmission is error free, the receiver may be unable to handle the frames as fast as they arrive and will lose some.
- Clearly, something has to be done to prevent this situation. Two approaches are commonly used. In the first one, **feedback-based flow control**, the receiver sends back information to the sender giving it permission to send more data, or at least telling the sender how the receiver is doing.
- In the second one, **rate-based flow control**, the protocol has a built-in mechanism that limits the rate at which senders may transmit data, without using feedback from the receiver.
- For example, hardware implementations of the link layer as **NICs** (**Network Interface Cards**) are some- times said to run at ‘‘wire speed,’’ meaning that they can handle frames as fast as they can arrive on the link. Any overruns are then not a link problem, so they are handled by higher layers.

❖ Error Detection and Correction

- The optical fiber in telecommunications networks, have tiny error rates so that transmission errors are a rare occurrence. But other channels, especially wireless links and aging local loops, have error rates that are orders of magnitude larger. For these links, transmission errors are the norm. They cannot be avoided at a reasonable expense or cost in terms of performance. The conclusion is that transmission errors are here to stay
- One strategy is to include enough redundant information to enable the receiver to deduce what the transmitted data must have been. The other is to include only enough redundancy to allow the receiver to deduce that an error has occurred (but not which error) and have it request a retransmission. The former strategy uses **error-correcting codes** and the latter uses **error-detecting codes**. The use of error-correcting codes is often referred to as **FEC** (**Forward Error Correction**)
- Each of these techniques occupies a different ecological niche. On channels that are highly reliable, such as fiber, it is cheaper to use an error-detecting code and just retransmit the occasional block found to be faulty, FEC is used on noisy channels because retransmissions are just as likely to be in error as the first transmission.
- A key consideration for these codes is the type of errors that are likely to oc- cur. Neither error-correcting codes nor error-detecting codes can handle all pos- sible errors since the

redundant bits that offer protection are as likely to be re- ceived in error as the data bits (which can compromise their protection).

- Other types of errors also exist. Sometimes, the location of an error will be known, perhaps because the physical layer received an analog signal that was far from the expected value for a 0 or 1 and declared the bit to be lost. This situation is called an **erasure channel**
- It is easier to correct errors in erasure channels than in channels that flip bits because even if the value of the bit has been lost, at least we know which bit is in error, Error-correcting codes are also seen in the physical layer, particularly for noisy channels, and in higher layers, particularly for eal-time media and content distribution. Error-detecting codes are commonly used in link, network, and transport layers.

❖ Error-Correcting Codes

➢ We will examine four different error-correcting codes:

1. Hamming codes.
2. Binary convolutional codes.
3. Reed-Solomon codes.
4. Low-Density Parity Check codes.

- All of these codes add redundancy to the information that is sent. A frame con- sists of *m* data (i.e., message) bits and *r* redundant (i.e. check) bits. In a **block code**, the *r* check bits are computed solely as a function of the *m* data bits with which they are associated, as though the *m* bits were looked up in a large table to find their corresponding *r* check bits.
- In a **systematic code**, the *m* data bits are sent directly, along with the check bits, rather than being encoded themselves before they are sent.
- In a **linear code**, the *r* check bits are computed as a linear function of the *m* data bits. Exclusive OR (XOR) or modulo 2 addition is a popu- lar choice. This means that encoding can be done with operations such as matrix multiplications or simple logic circuits.
- Let the total length of a block be *n* (i.e., *n* □ *m* □ *r*). We will describe this as an (*n*,*m*) code. An *n*-bit unit containing data and check bits is referred to as an *n*- bit **codeword**
- The **code rate**, or simply rate, is the fraction of the codeword that carries information that is not redundant, or *m/n*. The rates used in practice vary widely.
- To understand how errors can be handled, it is necessary to first look closely at what an error really is. Given any two codewords that may be transmitted or received—say, 10001001 and 10110001—it is possible to determine how many corresponding bits differ. In this case, 3 bits differ. To determine how many bits differ, just XOR the two codewords and count the number of 1 bits in the result.

→ For example:

10001001

10110001

00111000

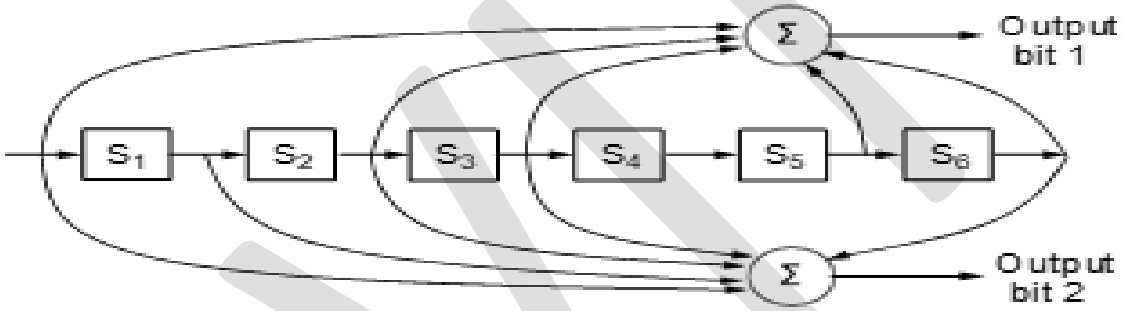
- The number of bit positions in which two codewords differ is called **Hamming distance** (Hamming, 1950). Its significance is that if two codewords are a Hamming distance *d* apart, it will require *d* single-bit errors to convert one into the other.
- In Hamming codes the bits of the codeword are numbered consecutively, starting with bit 1 at

the left end, bit 2 to its immediate right, and so on. The bits that are powers of 2 (1, 2, 4, 8, 16, etc.) are check bits. The rest (3, 5, 6, 7, 9, etc.) are filled up with the m data bits. This pattern is shown for an (11,7) Hamming code with 7 data bits and 4 check bits in Fig. 3-6

- If the check results are not all zero, however, an error has been detected. The set of check results forms the error syndrome that is used to pinpoint and correct the error. In Fig. 3-6, a single-bit error occurred on the channel so the check re- sults are 0, 1, 0, and 1 for k = 8, 4, 2, and 1, respectively.

- Hamming distances are valuable for understanding block codes, and Ham- ming codes are used in error-correcting memory, However, most networks use stronger codes. The second code we will look at is a **convolutional code**. This code is the only one we will cover that is not a block code

- In a convolutional code, an encoder processes a sequence of input bits and generates a sequence of output bits. There is no natural message size or encoding boundary as in a block code. The output depends on the current and previous input bits. That is, the encoder has memory. The number of previous bits on which the output depends is called the **constraint length** of the code



Convolutional codes are widely used in deployed networks, for example, as part of the GSM mobile phone system, in satellite communications, and in 802.11. This code is known as the NASA convolutional code of $r = 1/2$ and $k = 7$, since it was first used for the Voyager space missions starting in 1977. Since then it has been liberally reused, for example, as part of 802.11.

In Fig. above, each input bit on the left-hand side produces two output bits on the right-hand side that are XOR sums of the input and internal state. Since it deals with bits and performs linear operations, this is a binary, linear convolutional code. Since 1 input bit produces 2 output bits, the code rate is 1/2. It is not sys- tematic since none of the output bits is simply the input bit.

- Extensions of the Viterbi algorithm can work with these uncertainties to provide stronger error correction. This approach of working with the uncertainty of a bit is called **soft-decision decoding**. Conversely, decid- ing whether each bit is a 0 or a 1 before subsequent error correction is called **hard-decision decoding**.
- The third kind of error-correcting code we will describe is the **Reed-Solomon code**. Like Hamming codes, Reed-Solomon codes are linear block codes, and they are often systematic too.
- Reed-Solomon codes are widely used in practice because of their strong error-correction properties, particularly for burst errors. They are used for DSL, data over cable, satellite communications, and perhaps most ubiquitously on CDs, DVDs, and Blu-ray discs.
- Reed-Solomon codes are often used in combination with other codes such as a convolutional code. The thinking is as follows. Convolutional codes are effective at handling isolated bit errors, but they will fail, likely with a burst of errors, if there are too many errors in the received bit stream. the Reed-Solomon decoding can mop up the error bursts, a task at which it

is very good.

- The final error-correcting code we will cover is the **LDPC** (**Low-Density Parity Check**) code. LDPC codes are linear block codes that were invented by Robert Gallagher in his doctoral thesis (Gallagher, 1962), they were promptly forgotten, only to be reinvented in 1995 when advances in comput- ing power had made them practical.

- n an LDPC code, each output bit is formed from only a fraction of the input bits. This leads to a matrix representation of the code that has a low density of 1s, hence the name for the code. The received codewords are decoded with an approximation algorithm that iteratively improves on a best fit of the received data to a legal codeword. This corrects errors.

- LDPC codes are practical for large block sizes and have excellent error-cor- rection abilities that outperform many other codes (including the ones we have looked at) in practice.

❖ Error-Detecting Codes

- Error-correcting codes are widely used on wireless links, which are notori- ously noisy and error prone when compared to optical fibers. However, over fiber or high-quality copper, the error rate is much lower, so error detection and retrans- mission is usually more efficient there for dealing with the occasional error.

We will examine three different error-detecting codes. They are all linear, systematic block codes:

1. Parity.
2. Checksums.
3. Cyclic Redundancy Checks (CRCs).

- Consider the first error-detecting code, in which a single **parity bit** is appended to the data. The parity bit is chosen so that the number of 1 bits in the codeword is even (or odd).

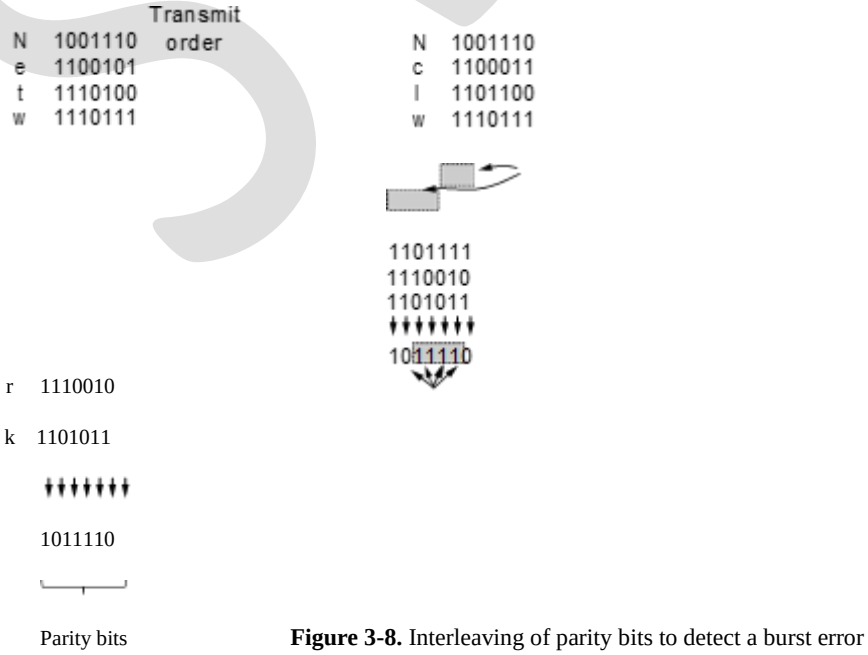


Figure 3-8. Interleaving of parity bits to detect a burst error

- there is something else we can do that provides better protection against burst errors: we can compute the parity bits over the data in a different order than the order in which the data bits are transmitted. Doing so is called **interleaving**.
- Interleaving is a general technique to convert a code that detects (or corrects) isolated errors into a code that detects (or corrects) burst errors. In Fig. 3-8, when a burst error of length n □ 7 occurs, the bits that are in error are spread across different columns.
- The second kind of error-detecting code, the **checksum**, is closely related to groups of parity bits. The word ‘‘checksum’’ is often used to mean a group of check bits associated with a message, regardless of how are calculated
- A group of parity bits is one example of a checksum. However, there are other, stronger checksums based on a running sum of the data bits of the message. The checksum is usually placed at the end of the message, as the complement of the sum func- tion. This way, errors may be detected by summing the entire received codeword, both data bits and checksum
- A better choice is **Fletcher's checksum** (Fletcher, 1982). It includes a posi- tional component, adding the product of the data and its position to the running sum. This provides stronger detection of changes in the position of data.
- Although the two preceding schemes may sometimes be adequate at higher layers, in practice, a third and stronger kind of error-detecting code is in wide- spread use at the link layer: the **CRC** (**Cyclic Redundancy Check**), also known as a **polynomial code**.
- Polynomial arithmetic is done modulo 2, according to the rules of algebraic field theory. It does not have carries for addition or borrows for subtraction. Both addition and subtraction are identical to exclusive OR.

10011011 + 11001010	00110011 + 11001101	11110000 – 10100110	01010101 – 10101111
01010001	11111110	01010110	11111010

Long division is carried out in exactly the same way as it is in binary except that the subtraction is again done modulo 2. A divisor is said ‘‘to go into’’ a dividend if the dividend has as many bits as the divisor.

- ♦ When the polynomial code method is employed, the sender and receiver must agree upon a **generator polynomial**, $G(x)$, in advance.
- ➢ The algorithm for computing the CRC is as follows:

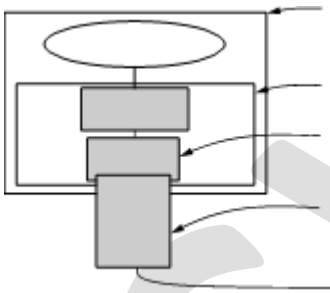
1. Let r be the degree of $G(x)$. Append r zero bits to the low-order end of the frame so it now contains m □ r bits and corresponds to the polynomial $x^r M(x)$.
2. Divide the bit string corresponding to $G(x)$ into the bit string corres- ponding to $x^r M(x)$, using modulo 2 division.
3. Subtract the remainder (which is always r or fewer bits) from the bit string corresponding to $x^r M(x)$ using modulo 2 subtraction. The re- sult is the checksummed frame to be transmitted. Call its polynomial $T(x)$.

- Both the high- and low- order bits of the generator must be 1. To compute the CRC for some frame with m bits corresponding to the polynomial $M(x)$, the frame must be longer than the generator polynomial.
- The idea is to append a CRC to the end of the frame in such a way that the polynomial

represented by the checksummed frame is divisi- ble by $G(x)$. When the receiver gets the checksummed frame, it tries dividing it by $G(x)$. If there is a remainder, there has been a transmission error.

❖ ELEMENTARY DATA LINK PROTOCOLS

- To start with, we assume that the physical layer, data link layer, and network layer are independent processes that communicate by passing messages back and forth. A common implementation is shown in Fig. 3-10
- The physical layer proc- ess and some of the data link layer process run on dedicate hardware called a **NIC** (**Network Interface Card**), three processes offloaded to ded- icated hardware called a **network accelerator**
- The rest of the link layer process and the network layer process run on the main CPU as part of the operating system, with the soft- ware for the link layer process often taking the form of a **device driver**



- Another key assumption is that machine *A* wants to send a long stream of data to machine *B*, using a reliable, connection-oriented service. Later, we will consid- er the case where *B* also wants to send data to *A* simultaneously. *A* is assumed to have an infinite supply of data ready to send and never has to wait for data to be produced. Instead, when *A*'s data link layer asks for data, the network layer is al- ways able to comply immediately.
- As far as the data link layer is concerned, the packet passed across the inter- face to it from the network layer is pure data, whose every bit is to be delivered to the destination's network layer. The fact that the destination's network layer may interpret part of the packet as a header is of no concern to the data link layer.

```
#define MAX PKT 1024                    /* determines packet size in bytes */

typedef enum {false, true} boolean;     /* boolean type */

typedef unsigned int seq nr;            /* sequence or ack numbers */ typedef
struct {unsigned char data[MAX PKT];} packet;   /* packet definition */

typedef enum {data, ack, nak} frame kind;   /* frame kind definition */

typedef struct {                        /* frames are transported in this layer */

  frame kind kind;                      /* what kind of frame is it? */

  seq nr seq;                           /* sequence number */
```

```
  – seq nr ack;                    /* acknowledgement number */
  packet info;                     /* the network layer packet */
} frame;


– – – /* Wait for an event to happen; return its type in event. */ void wait
for event(event type *event);

– – /* Fetch a packet from the network layer for transmission on the channel. */ void from
network layer(packet *p);

– – /* Deliver information from an inbound frame to the network layer. */ void to
network layer(packet *p);

– – /* Go get an inbound frame from the physical layer and copy it to r. */ void from
physical layer(frame *r);

– – /* Pass the frame to the physical layer for transmission. */ void to
physical layer(frame *s);

– – /* Start the clock running and enable the timeout event. */ void start
timer(seq nr k);

– – /* Stop the clock and disable the timeout event. */ void stop
timer(seq nr k);

– – – /* Start an auxiliary timer and enable the ack timeout event. */ void start
ack timer(void);

– – – /* Stop the auxiliary timer and disable the ack timeout event. */ void stop
ack timer(void);

– – – – /* Allow the network layer to cause a network layer ready event. */ void
enable network layer(void);

– – – – /* Forbid the network layer from causing a network layer ready event. */ void
disable network layer(void);

– /* Macro inc is expanded in-line: increment k circularly. */ #define inc(k)
if (k < MAX SEQ) k=k+ 1; else k = 0
```

Figure 3-11. Some definitions needed in the protocols to follow. These defini- tions are located in the file *protocol.h*.

- This procedure only returns when something has hap- pened (e.g., a frame has arrived). Upon return, the variable *event* tells what hap- pened. The set of possible events differs for the various protocols to be described and will be defined separately for each protocol.
- When a frame arrives at the receiver, the checksum is recomputed. If the checksum in the frame is incorrect (i.e., there was a transmission error), the data link layer is so informed (*event* □ *cksum err*). If the inbound frame arrived undamaged, the data link layer is also informed (*event* □ *frame arrival*) so that it can acquire the frame for inspection using *from physical layer*.
- A *frame* is composed of four fields: *kind*, *seq*, *ack*, and *info*, the first three of which contain control information and the last of which may contain actual data to be transferred. These control fields are collectively called the **frame header**.

❖ A Utopian Simplex Protocol

- Data are transmitted in one direction only. Both the transmitting and receiving network layers are always ready. Processing time can be ignored. Infinite buffer space is available. And best of all, the communication channel between the data link lay- ers never damages or loses frames

- The protocol consists of two distinct procedures, a sender and a receiver. The sender runs in the data link layer of the source machine, and the receiver runs in the data link layer of the destination machine. No sequence numbers or acknowl- edgements are used here, so *MAX SEQ* is not needed.

- The sender is in an infinite while loop just pumping data out onto the line as fast as it can. The body of the loop consists of three actions: go fetch a packet from the (always obliging) network layer, construct an outbound frame using the variable *s*, and send the frame on its way.

- The receiver is equally simple. Initially, it waits for something to happen, the only possibility being the arrival of an undamaged frame. Eventually, the frame arrives and the procedure *wait for event* returns, with *event* set to *frame arrival* (which is ignored anyway)

```
* Protocol 1 (Utopia) provides for data transmission in one direction only, from sender to
  receiver. The communication channel is assumed to be error free and the receiver is assumed to
  be able to process all the input infinitely quickly. Consequently, the sender just sits in a loop
  pumping data out onto the line as fast as it can. */

typedef enum {frame arrival} event type;

 #include "protocol.h"

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 */
  }
}
```

Figure 3-12. A utopian simplex protocol.

The utopia protocol is unrealistic because it does not handle either flow con- trol or error correction. Its processing is close to that of an unacknowledged con- nectionless service that relies on higher layers to solve these problems, though even an unacknowledged connectionless service would do some error detection.

❖ A Simplex Stop-and-Wait Protocol for an Error-Free Channel

- Now we will tackle the problem of preventing the sender from flooding the receiver with frames faster than the latter is able to process them. This situation can easily happen in practice so being able to prevent it is of great importance.
- One solution is to build the receiver to be powerful enough to process a con- tinuous stream of back-to-back frames, It must have sufficient buffering and processing abilities to run at the line rate and must be able to pass the frames that are received to the network layer quickly enough.
- A more general solution to this problem is to have the receiver provide feed- back to the sender. After having passed a packet to its network layer, the receiver sends a little dummy frame back to the sender which, in effect, gives the sender permission to transmit the next frame.
- Protocols in which the sender sends one frame and then waits for an acknowl- edgement before proceeding are called **stop-and-wait.**

- As in protocol 1, the sender starts out by fetching a packet from the network layer, using it to construct a frame, and sending it on its way. The only difference between *receiver1* and *receiver2* is that after delivering a packet to the network layer, *receiver2* sends an acknowledgement frame back to the sender before entering the wait loop again.

❖ Simplex Stop-and-Wait Protocol for a Noisy Channel

- The normal situation of a communication channel that makes errors. Frames may be either damaged or lost completely. However, we assume that if a frame is damaged in transit, the receiver hardware will detect this.
- when it computes the checksum. If the frame is damaged in such a way that the checksum is nevertheless correct—an unlikely occurrence—this protocol (and all other protocols) can fail (i.e., deliver an incorrect packet to the network layer).

- At first glance it might seem that a variation of protocol 2 would work: adding a timer. The sender could send a frame, but the receiver would only send an ac- knowledgement frame if the data were correctly received.

- If a damaged frame ar- rived at the receiver, it would be discarded. After a while the sender would time out and send the frame again. This process would be repeated until the frame finally arrived intact.

```
/* Protocol 2 (Stop-and-wait) also provides for a one-directional flow of data from sender to receiver.
   The communication channel is once again assumed to be error

   free, as in protocol 1. However, this time the receiver has only a finite buffer capacity and a finite
   processing speed, so the protocol must explicitly prevent the sender from flooding the receiver
   with data faster than it can be handled. */

typedef enum {frame arrival} event type; #include
"protocol.h"

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 */

  }

}
```

Figure 3-13. A simplex stop-and-wait protocol.

- The network layer on machine *A* gives a series of packets to its data link layer, which must ensure that an identical series of packets is delivered to the net- work layer on machine *B* by its data link layer.
- In particular, the network layer on *B* has no way of knowing that a packet has been lost or duplicated, so the data link layer must guarantee that no combination of transmission errors, however unlike- ly, can cause a duplicate packet to be delivered to a network layer.

➢ Consider the following scenario:

1. The network layer on *A* gives packet 1 to its data link layer. The packet is correctly received at *B* and passed to the network layer on *B*. *B* sends an acknowledgement frame back to *A*.
2. The acknowledgement frame gets lost completely. It just never ar- rives at all. Life would be a great deal simpler if the channel man- gled and lost only data frames and not control frames, but sad to say, the channel is not very discriminating.
3. The data link layer on *A* eventually times out. Not having received an acknowledgement, it (incorrectly) assumes that its data frame was lost or damaged and sends the frame containing packet 1 again.
4. The duplicate frame also arrives intact at the data link layer on *B* and is unwittingly passed to the network layer there. If *A* is sending a file to *B*, part of the file will be duplicated (i.e., the copy of the file made by *B* will be incorrect and the error will not have been detected). In other words, the protocol will fail.

- An example of this kind of protocol is shown in Fig. 3-14. Protocols in which the sender waits for a positive acknowledgement before advancing to the next data item are often called **ARQ** (**Automatic Repeat reQuest**) or **PAR** (**Positive Acknowledgement with Retransmission**). Like protocol 2, this one also trans- mits data only in one direction.
- protocol 3 differs from its predecessors in that both sender and receiver have a variable whose value is remembered while the data link layer is in the wait state. The sender remembers the sequence number of the next frame to send in *next frame to send*; the receiver remembers the sequence number of the next frame expected in *frame expected*.
- After transmitting a frame and starting the timer, the sender waits for some- thing exciting to happen. Only three possibilities exist: an acknowledgement frame arrives undamaged, a damaged acknowledgement frame staggers in, or the timer expires. If a valid acknowledgement comes in, the sender fetches the next packet from its network layer and puts it in the buffer, overwriting the previous packet.
- When a valid frame arrives at the receiver, its sequence number is checked to see if it is a duplicate. If not, it is accepted, passed to the network layer, and an acknowledgement is generated.

❖ SLIDING WINDOW PROTOCOLS

- The previous protocols, each using a separate link for simplex data traffic (in different directions). Each link is then comprised of a ‘‘forward’’ channel (for data) and a ‘‘reverse’’ channel (for acknowledgement data frames were transmitted in one direction only. In most practical situations, there is a need to transmit data in both directions.
- A better idea is to use the same link for data in both directions. After all, in protocols 2 and 3 it was already being used to transmit frames both ways, and the reverse channel normally has the same capacity as the forward channel.

- Although interleaving data and control frames on the same link is a big im- provement over having two separate physical links, yet another improvement is possible. When a data frame arrives, instead of immediately sending a separate control frame, the receiver restrains itself and waits until the network layer passes it the next packet.

- The acknowledgement is attached to the outgoing data frame (using the *ack* field in the frame header). In effect, the acknowledgement gets a free ride on the next outgoing data frame. The technique of temporarily delaying outgoing acknowledgements so that they can be hooked onto the next outgoing data frame is known as **piggybacking**.

- The *ack* field in the frame header costs only a few bits, whereas a separate frame would need a header, the acknowledgement, and a checksum.

- The next three protocols are bidirectional protocols that belong to a class called **sliding window** protocols. The three differ among themselves in terms of ef- ficiency, complexity, and buffer requirements, as discussed later. In these, as in all sliding window protocols, each outbound frame contains a sequence number, ranging from 0 up to some maximum.

- The essence of all sliding window protocols is that at any instant of time, the sender maintains a set of sequence numbers corresponding to frames it is permit- ted to send. These frames are said to fall within the **sending window**. Similarly, the receiver also maintains a **receiving window** corresponding to the set of frames it is permitted to accept. The sender's window and the receiver's window need not have the same lower and upper limits or even have the same size.

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */

#define MAX SEQ 1                    /* must be 1 for protocol 3 */ typedef
enum {frame arrival, cksum err, timeout} event type;

#include "protocol.h"

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) {
```

```
            stop timer(s.ack);              /* turn the timer off */
            from network layer(&buffer); /* get the next one to send */
            inc(next frame to send);        /* invert next frame to send */
        }
    }
}
}

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);                        /* send acknowledgement */
    }
  }
}
```

Figure 3-14. A positive acknowledgement with retransmission protocol.

❖ A One-Bit Sliding Window Protocol

- Before tackling the general case, let us examine a sliding window protocol with a window size of 1. Such a protocol uses stop-and-wait since the sender transmits a frame and waits for its acknowledgement before sending the next one.

- Figure 3-16 depicts such a protocol. Like the others, it starts out by defining some variables. *Next frame to send* tells which frame the sender is trying to send. Similarly, *frame expected* tells which frame the receiver is expecting. In both cases, 0 and 1 are the only possibilities.

```
/* Protocol 4 (Sliding window) is bidirectional. */

#define MAX SEQ 1                          /* must be 1 for protocol 4 */ typedef
enum {frame arrival, cksum err, timeout} event type;

#include "protocol.h"

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;                    /* frame expected next */

    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 */

    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 seq number expected next */

            }

            if (r.ack == next frame to send) { /* handle outbound frame stream */ stop
                timer(r.ack);              /* turn the timer off */

                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 */
    }
}
```

Figure 3-16. A 1-bit sliding window protocol.

- Under normal circumstances, one of the two data link layers goes first and transmits the first frame. In other words, only one of the data link layer programs should contain the *to physical layer* and *start timer* procedure calls outside the main loop. The starting machine fetches the first packet from its network layer, builds a frame from it, and sends it.

- The acknowledgement field contains the number of the last frame received without error. If this number agrees with the sequence number of the frame the sender is trying to send, the sender knows it is done with the frame stored in *buff- er* and can fetch the next packet from its network layer.

- When the first valid frame arrives at computer *B*, it will be accepted and *frame expected* will be set to a value of 1. All the subsequent frames received will be rejected because *B* is now expecting frames with sequence number 1, not0. Furthermore, since all the duplicates will have *ack* □ 1 and *B* is still waiting for an acknowledgement of 0, *B* will not go and fetch a new packet from its network layer.

- After every rejected duplicate comes in, *B* will send *A* a frame containing *seq* □ 0 and *ack* □ 0. Eventually, one of these will arrive correctly at *A*, causing *A* to begin sending the next packet. No combination of lost frames or premature timeouts can cause the protocol to deliver duplicate packets to either network layer, to skip a packet, or to deadlock.

- In part (a), the normal opera- tion of the protocol is shown. In (b) the peculiarity is illustrated. If *B* waits for *A*'s first frame before sending one of its own, the sequence is as shown in (a), and every frame is accepted.

- if *A* and *B* simultaneously initiate communication, their first frames cross, and the data link layers then get into situation (b). In (a) each frame arrival brings a new packet for the network layer; there are no duplicates. In (b) half of the frames contain duplicates, even though there are no transmission errors. Simi- lar situations can occur as a result of premature timeouts, even when one side clearly starts first. In fact, if multiple premature timeouts occur, frames may be sent three or more times, wasting valuable bandwidth.

❖ A Protocol Using Go-Back-N

- The transmission time re- quired for a frame to arrive at the receiver plus the transmission time for the ac- knowledgement to come back is negligible. Sometimes this assumption is clearly false. In these situations the long round-trip time can have important implications for the efficiency of the bandwidth utilization.

- The problem described here can be viewed as a consequence of the rule requiring a sender to wait for an acknowledgement before sending another frame. If we relax that restriction, much better efficiency can be achieved. Basically, the solution lies in allowing the sender to transmit up to *w* frames before blocking, in- stead of just 1.

- To find an appropriate value for *w* we need to know how many frames can fit inside the channel as they propagate from sender to receiver. This capacity is de- termined by the bandwidth in bits/sec multiplied by the one-way transit time, or the **bandwidth-delay product** of the link. We can divide this quantity by the number of bits in a frame to express it as a number of frames. Call this quantity *BD.*

- For smaller window sizes, the utilization of the link will be less than 100% since the sender will be blocked sometimes. We can write the utilization as the fraction of time that the sender is not blocked:

link utilization □ w 1 □ 2*BD*

- This value is an upper bound because it does not allow for any frame processing time and treats the acknowledgement frame as having zero length, since it is usually short. The equation shows the need for having a large window *w* when- ever the bandwidth-delay product is large.

- This technique of keeping multiple frames in flight is an example of **pipelin- ing**. Pipelining frames over an unreliable communication channel raises some serious issues. First, what happens if a frame in the middle of a long stream is damaged or lost? Large numbers of succeeding frames will arrive at the receiver before the sender even finds out that anything is wrong.

- One option, called **go-back-n**, is for the receiver simply to discard all subsequent frames, sending no acknowledgements for the discarded frames. This strategy corresponds to a receive window of size 1. In other words, the data link layer refuses to accept any frame except the next one it must give to the network layer. If the sender's window fills up before the timer runs out, the pipeline will begin to empty.

- In Fig. 3-18(b) we see go-back-n for the case in which the receiver's window is large. Frames 0 and 1 are correctly received and acknowledged. Frame 2, however, is damaged or lost. The sender, unaware of this problem, continues to send frames until the timer for frame 2 expires. Then it backs up to frame 2 and starts over with it, sending 2, 3, 4, etc. all over again.

- The other general strategy for handling errors when frames are pipelined is called **selective repeat**. When it is used, a bad frame that is received is discarded, but any good frames received after it are accepted and buffered.

- Two basic approaches are available for dealing with errors in the presence of pipelining, both of which are shown in Fig. 3-18.

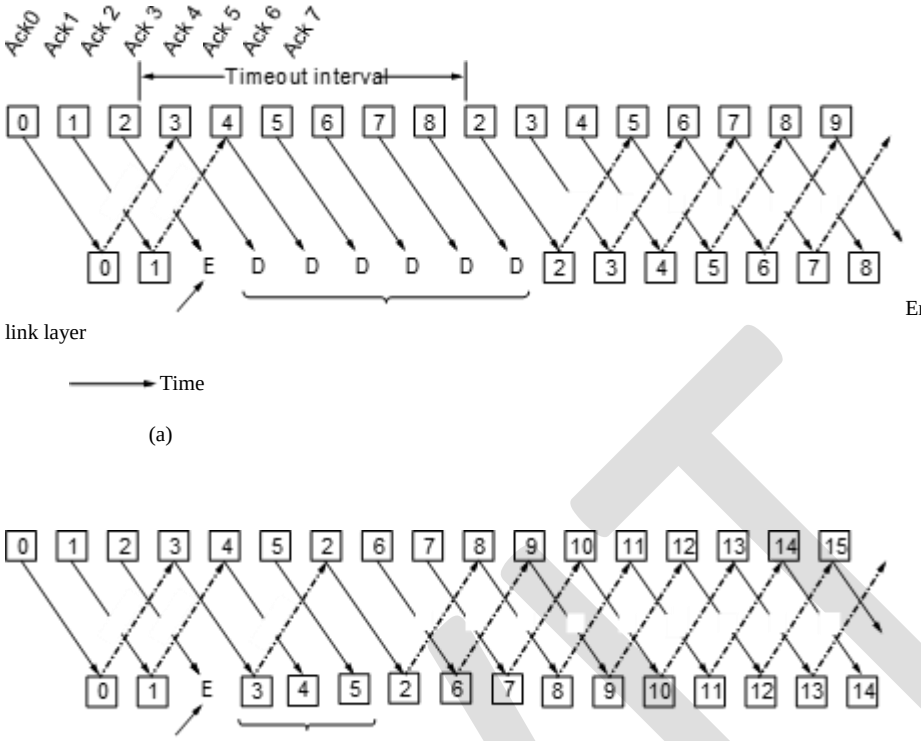


Figure 3-18. Pipelining and error recovery. Effect of an error when

(a) receiver's window size is 1 and (b) receiver's window size is large

- When the sender times out, only the oldest unacknowledged frame is retransmitted. If that frame arrives correctly, the receiver can deliver to the network layer, in sequence, all the frames it has buffered. Selective repeat corresponds to a receiver window larger than 1.
- Selective repeat is often combined with having the receiver send a negative acknowledgement (NAK) when it detects an error, for example, when it receives a checksum error or a frame out of sequence. NAKs stimulate retransmission be- fore the corresponding timer expires and thus improve performance.
- When that arrives, the data link layer now has 2, 3, 4, and 5 and can pass all of them to the network layer in the correct order. It can also acknowledge all frames up to and including 5, as shown in the figure. If the NAK should get lost, eventually the sender will time out for frame 2 and send it (and only it) of its own accord, but that may be a quite a while later.
- When the network layer has a packet it wants to send, it can cause a *net- work layer ready* event to happen. However, to enforce the flow control limit on the sender window or the number of unacknowledged frames that may be out- standing at any time, the data link layer must be able to keep the network layer from bothering it with more work. The library procedures *enable network layer* and *disable network layer* do this job.
- The maximum number of frames that may be outstanding at any instant is not

the same as the size of the sequence number space. For go-back-n, *MAX SEQ* frames may be outstanding at any instant, even though there are *MAX SEQ* □ 1 distinct sequence numbers (which are 0, 1, . . . , *MAX SEQ*). We will see an even tighter restriction for the next protocol, selective repeat. To see why this res- triction is required, consider the following scenario with *MAX SEQ* □ 7:

1. he sender sends frames 0 through 7.
2. A piggybacked acknowledgement for 7 comes back to the sender.
3. The sender sends another eight frames, again with sequence numbers 0 through 7.
4. Now another piggybacked acknowledgement for frame 7 comes in.

```
/* Protocol 5 (Go-back-n) allows multiple outstanding frames. The sender may transmit up to MAX SEQ
frames without waiting for an ack. In addition, unlike in the previous

protocols, the network layer is not assumed to have a new packet all the time. Instead, the network layer
causes a network layer ready event when there is a packet to send. */

#define MAX SEQ 7

typedef enum {frame arrival, cksum err, timeout, network layer ready} event type; #include
"protocol.h"


static boolean between(seq nr a, seq nr b, seq nr c)

{

/* Return true if a <= b < c circularly; false otherwise. */

 if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))

      return(true); else

      return(false);

}

static void send data(seq nr frame nr, seq nr frame expected, packet buffer[ ])

{

/* Construct and send a data frame. */

 frame s;                                  /* scratch variable */

 s.info = buffer[frame nr];                /* insert packet into frame */

 s.seq = frame nr;                         /* insert sequence number into frame */ s.ack =
(frame expected + MAX SEQ) % (MAX SEQ + 1); /* piggyback ack */

 to physical layer(&s);                    /* transmit the frame */

 start timer(frame nr);                    /* start the timer running */

}


void protocol5(void)

{

 seq nr next frame to send;                /* MAX SEQ > 1; used for outbound stream */ seq nr ack
 expected;                                 /* oldest frame as yet unacknowledged */
```

```
––seq nr frame expected;                /* next frame expected on inbound stream */
frame r;                                /* scratch variable */
–packet buffer[MAX SEQ + 1];            /* buffers for the outbound stream */
–seq nr nbuffered;                      /* number of output buffers currently in use */
––seq nr i;                             /* used to index into the buffer array */ event type
event;


––––enable network layer();             /* allow network layer ready events */
–ack expected = 0;                      /* next ack expected inbound */
–––next frame to send = 0;              /* next frame going out */
–frame expected = 0;                    /* number of frame expected inbound */
nbuffered = 0;                          /* initially no packets are buffered */
while (true) {
  –––wait for event(&event);            /* four possibilities: see event type above */
  switch(event) {
    ––case network layer ready:         /* the network layer has a packet to send */
        /* Accept, save, and transmit a new frame. */
        –––––from network layer(&buffer[next frame to send]); /* fetch new packet */
        nbuffered = nbuffered + 1;      /* expand the sender's window */
        ––––––––send data(next frame to send, frame expected, buffer);/* transmit the frame */
        inc(next frame to send);        /* advance sender's upper window edge */ break;


    –––case frame arrival:              /* a data or control frame has arrived */ from physical
        layer(&r);                      /* get incoming frame from physical layer */
        –if (r.seq == frame expected) {
            /* Frames are accepted only in order. */
            ––to network layer(&r.info);  /* pass packet to network layer */
            –inc(frame expected);       /* advance lower edge of receiver's window */
         }

        /* Ack n implies n □ 1, n □ 2, etc. Check for this. */
        ––––while (between(ack expected, r.ack, next frame to send)) {
```

```
            /* Handle piggybacked ack. */

            nbuffered = nbuffered □ 1;          /* one frame fewer buffered */

            ---stop timer(ack expected);        /* frame arrived intact; stop timer */ inc(ack
            expected);                          /* contract sender's window */

        }

        break;

    case cksum err: break;          /* just ignore bad frames */

  ----case timeout:                 /* trouble; retransmit all outstanding frames */ next frame
        to send = ack expected;     /* start retransmitting here */

        for (i = 1; i <= nbuffered; i++) {

            --------send data(next frame to send, frame expected, buffer);/* resend frame */
            inc(next frame to send);    /* prepare to send the next one */

        }

  }

  -----if (nbuffered < MAX SEQ) enable
        network layer();

  else


 }

}
```

❖ A Protocol Using Selective Repeat

- The go-back-n protocol works well if errors are rare, but if the line is poor it wastes a lot of bandwidth on retransmitted frames. An alternative strategy, the selective repeat protocol, is to allow the receiver to accept and buffer the frames following a damaged or lost one.
- In this protocol, both sender and receiver maintain a window of outstanding and acceptable sequence numbers, respectively. The sender's window size starts out at 0 and grows to some predefined maximum. The receiver's window, in con- trast, is always fixed in size and equal to the predetermined maximum.
- The re- ceiver has a buffer reserved for each sequence number within its fixed window. Associated with each buffer is a bit (*arrived*) telling whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked by the function *between* to see if it falls within the window. If so and if it has not already been re- ceived, it is accepted and stored.
- This action is taken without regard to whether or not the frame contains the next packet expected by the network layer. Of course, it must be kept within the data link layer and not passed to the network layer until all the lower-numbered frames have already been delivered to the network layer in the correct order.

- Nonsequential receive introduces further constraints on frame sequence num- bers compared to protocols in which frames are only accepted in order. We can illustrate the trouble most easily with an example. Suppose that we have a 3-bit sequence number, so that the sender is permitted to transmit up to seven frames before being required to wait for an acknowledgement.
- It is at this point that disaster strikes in the form of a lightning bolt hitting the telephone pole and wiping out all the acknowledgements. The protocol should operate correctly despite this disaster

```
/* Protocol 6 (Selective repeat) accepts frames out of order but passes packets to the network layer in order.
   Associated with each outstanding frame is a timer. When the timer

   expires, only that frame is retransmitted, not all the outstanding frames, as in protocol 5. */

---#define MAX SEQ 7                                   /* should be 2^n  1 */ #define
NR BUFS ((MAX SEQ + 1)/2)

------typedef enum {frame arrival, cksum err, timeout, network layer ready, ack timeout} event type; #include
"protocol.h"

-boolean no nak = true;                                 /* no nak has been sent yet */

---seq nr oldest frame = MAX SEQ + 1;                   /* initial value is only for the simulator */

---static boolean between(seq nr a, seq nr b, seq nr c)

{

/* Same as between in protocol 5, but shorter and more obscure. */ return ((a <= b) &&
  (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));

}

------static void send frame(frame kind fk, seq nr frame nr, seq nr frame expected, packet buffer[ ])

{

/* Construct and send a data, ack, or nak frame. */

  frame s;                                              /* scratch variable */

  --s.kind = fk;                                        /* kind == data, ack, or nak */ if (fk ==
  data) s.info = buffer[frame nr % NR BUFS];

  ----s.seq = frame nr;                                 /* only meaningful for data frames */ s.ack =
  (frame expected + MAX SEQ) % (MAX SEQ + 1);

  -if (fk == nak) no nak = false;                       /* one nak per frame, please */

  -----to physical layer(&s);                           /* transmit the frame */ if (fk ==
  data) start timer(frame nr % NR BUFS);

  --stop ack timer();                                   /* no need for separate ack frame */

}


 void protocol6(void)
```

```
{
  seq nr ack expected;                    /* lower edge of sender's window */
  seq nr next frame to send;              /* upper edge of sender's window + 1 */
  seq nr frame expected;                  /* lower edge of receiver's window */
  seq nr too far;                         /* upper edge of receiver's window + 1 */
  int i;                                  /* index into buffer pool */
  frame r;                                /* scratch variable */
  packet out buf[NR BUFS];                /* buffers for the outbound stream */
  packet in buf[NR BUFS];                 /* buffers for the inbound stream */
  boolean arrived[NR BUFS];               /* inbound bit map */
  seq nr nbuffered;                       /* how many output buffers currently used */ event type
event;

  enable network layer();                 /* initialize */
  ack expected = 0;                       /* next ack expected on the inbound stream */
  next frame to send = 0;                 /* number of next outgoing frame */ frame
expected = 0;
  too far = NR BUFS;
  nbuffered = 0;                          /* initially no packets are buffered */ for (i = 0; i
< NR BUFS; i++) arrived[i] = false;
  while (true) {
    wait for event(&event);               /* five possibilities: see event type above */ switch(event)
    {
      case network layer ready:           /* accept, save, and transmit a new frame */ nbuffered =
          nbuffered + 1;                  /* expand the window */
          from network layer(&out buf[next frame to send % NR BUFS]); /* fetch new
          packet */ send frame(data, next frame to send, frame expected, out buf);/* transmit the frame */ inc(next
          frame to send);                 /* advance upper window edge */
          break;
      case frame arrival:                 /* a data or control frame has arrived */
          from physical layer(&r);        /* fetch incoming frame from physical layer */ if (r.kind ==
          data) {
            /* An undamaged frame has arrived. */ if ((r.seq
            != frame expected) && no nak)
          send frame(nak, 0, frame expected, out buf); else start ack timer();
```

```
        ---if (between(frame expected,r.seq,too far) && (arrived[r.seq%NR BUFS]==false)) {
            /* Frames may be accepted in any order. */
            -arrived[r.seq % NR BUFS] = true;        /* mark buffer as full */
            ----in buf[r.seq % NR BUFS] = r.info; /* insert data into buffer */ while
            (arrived[frame expected % NR BUFS]) {
                /* Pass frames and advance window. */
                -----to network layer(&in buf[frame expected % NR BUFS]);
                -no nak = true;
                --arrived[frame expected % NR BUFS] = false;
                ----inc(frame expected); /* advance lower edge of receiver's window */ inc(too
                far);                    /* advance upper edge of receiver's window */ start ack
                timer();                 /* to see if a separate ack is needed */
            }
        }
    }
    ----------if((r.kind==nak) && between(ack expected,(r.ack+1)%(MAX SEQ+1),next frame to send))
        send frame(data, (r.ack+1) % (MAX SEQ + 1), frame expected, out buf);

    ----while (between(ack expected, r.ack, next frame to send)) { nbuffered =
        nbuffered  1;                    /* handle piggybacked ack */
        ---stop timer(ack expected % NR BUFS);   /* frame arrived intact */
        -inc(ack expected);              /* advance lower edge of sender's window */
    }
    -break;  case  cksum
err:
    ----if (no nak) send frame(nak, 0, frame expected, out buf); /* damaged frame */
    break;
case timeout:
    ----send frame(data, oldest frame, frame expected, out buf); /* we timed out */ break;
-case ack timeout:
    ---send frame(ack,0,frame expected, out buf);    /* ack timer expired; send ack */
}
-----if (nbuffered < NR BUFS) enable network layer(); else disable network layer();
}
```

Figure 3-21. A sliding window protocol using selective repeat.

- The sender is happy to learn that all its transmitted frames did actually arrive correctly, so it advances its window and immediately sends frames 7, 0, 1, 2, 3, 4, and 5. Frame 7 will be accepted by the receiver and its packet will be passed di- rectly to the network layer.

- The essence of the problem is that after the receiver advanced its window, the new range of valid sequence numbers overlapped the old one.

❖ EXAMPLE DATA LINK PROTOCOLS

- Within a single building, LANs are widely used for interconnection, but most wide-area network infrastructure is built up from point-to-point lines. In Chap. 4, we will look at LANs.

- The first situation is when packets are sent over SONET optical fiber links in wide-area networks. These links are widely used, for example, to connect routers in the different loca- tions of an ISP's network.

- The second situation is for ADSL links running on the local loop of the tele- phone network at the edge of the Internet. These links connect millions of individ- uals and businesses to the Internet.

- he Internet needs point-to-point links for these uses, as well as dial-up mo- dems, leased lines, and cable modems, and so on. A standard protocol called **PPP.**

THE MEDIUM ACCESS CONTROL SUBLAYER

- In any broadcast network, the key issue is how to determine who gets to use the channel when there is competition for it. To make this point, consider a con- ference call in which six people, on six different telephones, are all connected so that each one can hear and talk to all the others.
- When only a single channel is available, it is much harder to determine who should go next. Many protocols for solving the problem are known. They form the contents of this chap- ter. In the literature, broadcast channels are sometimes referred to as **multiaccess channels** or **random access channels**.
- The protocols used to determine who goes next on a multiaccess channel be- long to a sublayer of the data link layer called the **MAC** (**Medium Access Con- trol**) sublayer. The MAC sublayer is especially important in LANs, particularly wireless ones because wireless is naturally a broadcast channel.
- WANs, in con- trast, use point-to-point links, except for satellite networks. Because multiaccess channels and LANs are so closely related, in this chapter we will discuss LANs in eneral, including a few issues that are not strictly part of the MAC sublayer, but the main subject here will be control of the channel.
- Technically, the MAC sublayer is the bottom part of the data link layer, so logically we should have studied it before examining all the point-to-point proto- cols in Chap. 3. Nevertheless, for most people, it is easier to understand protocols involving multiple parties after two-party protocols are well understood.

❖ THE CHANNEL ALLOCATION PROBLEM

- The central theme of this chapter is how to allocate a single broadcast channel among competing users. The channel might be a portion of the wireless spectrum in a geographic region, or a single wire or optical fiber to which multiple nodes are connected.
- In both cases, the channel connects each user to all other users and any user who makes full use of the channel interferes with other users who also wish to use the channel.
- We will first look at the shortcomings of static allocation schemes for bursty traffic. Then, we will lay out the key assumptions used to model the dynamic schemes that we examine in the following sections.

❖ Static Channel Allocation

- The traditional way of allocating a single channel, such as a telephone trunk, among multiple competing users is to chop up its capacity by using one of the multiplexing schemes we described in Sec. 2.5, such as FDM (Frequency Division Multiplexing).
- When there is only a small and constant number of users, each of which has a steady stream or a heavy load of traffic, this division is a simple and efficient allocation mechanism. A wireless example is FM radio stations. Each station gets a portion of the FM band and uses it most of the time to broadcast its signal.
- when the number of senders is large and varying or the traffic is bursty, FDM presents some problems. If the spectrum is cut up into N regions and fewer than N users are currently interested in communicating, a large piece of valuable spectrum will be wasted.
- The poor performance of static FDM can easily be seen with a simple queue- ing theory calculation. Let us start by finding the mean time delay, T, to send a frame onto a channel of capacity C bps. We assume that the frames arrive ran- domly with an average arrival rate of λ frames/sec, and that the frames vary in length with an average length of $1/\mu$ bits. With these parameters, the service rate of the channel is μC frames/sec.

  A standard queueing theory result is: $T = 1/(\mu C - \lambda)$
- The mean delay for the divided channel is N times worse than if all the frames were somehow magically arranged orderly in a big central queue. Since none of the traditional static channel allocation methods work well at all with bursty traffic, we will now explore dynamic methods.

❖ Assumptions for Dynamic Channel Allocation

Before we get to the first of the many channel allocation methods in this chap- ter, it is worthwhile to carefully formulate the allocation problem. Underlying all the work done in this area are the following five key assumptions:

1. **Independent Traffic**. The model consists of N independent **stations** (e.g., computers, telephones), each with a program or user that gener- ates frames for transmission. The expected number of frames gener- ated in an interval of length Δt is $\lambda \Delta t$, where λ is a constant (the arri- val rate of new frames). Once a frame has been generated, the sta- tion is blocked and does nothing until the frame has been suc- cessfully transmitted.
2. **Single Channel**. A single channel is available for all communica- tion. All stations can transmit on it and all can receive from it. The stations are assumed to be equally capable, though protocols may assign them different roles (e.g., priorities).

3. **Observable Collisions**. If two frames are transmitted simultan- eously, they overlap in time and the resulting signal is garbled. This event is called a **collision**. All stations can detect that a collision has occurred. A collided frame must be transmitted again later. No er- rors other than those generated by collisions occur.
4. **Continuous or Slotted Time**. Time may be assumed continuous, in which case frame transmission can begin at any instant. Alterna- tively, time may be slotted or divided into discrete intervals (called slots). Frame transmissions must then begin at the start of a slot. A slot may contain 0, 1, or more frames, corresponding to an idle slot, a successful transmission, or a collision, respectively.

5. **Carrier Sense or No Carrier Sense**. With the carrier sense as- sumption, stations can tell if the channel is in use before trying to use it. No station will attempt to use the channel while it is sensed as busy. If there is no carrier sense, stations cannot sense the channel before trying to use it. They just go ahead and transmit. Only later can they determine whether the transmission was successful.
    - The first one says that frame arrivals are independent, both across stations and at a particular station, and that frames are generated unpredictably but at a constant rate.
    - **Poisson models**, as they are frequently called, are useful because they are mathematically tractable. They help us analyze protocols to understand roughly how performance changes over an operating range and how it compares with other designs.
    - The single-channel assumption is the heart of the model. No external ways to communicate exist. Stations cannot raise their hands to request that the teacher call on them, so we will have to come up with better solutions.
    - The collision assumption is basic. Stations need some way to detect collisions if they are to retransmit frames rather than let them be lost. For wired channels, node hardware can be designed to detect collisions when they occur. The stations can then terminate their transmissions prematurely to avoid wasting capacity.
    - The reason for the two alternative assumptions about time is that slotted time can be used to improve performance. However, it requires the stations to follow a master clock or synchronize their actions with each other to divide time into dis- crete intervals. Hence, it is not always available.
    - similarly, a network may have carrier sensing or not have it. Wired networks will generally have carrier sense. Wireless networks cannot always use it ef- fectively because not every station may be within radio range of every other sta- tion. Similarly, carrier sense will not be available in other settings in which a sta- tion cannot communicate directly with other stations, for example a cable modem in which stations must communicate via the cable headend.
    - To avoid any misunderstanding, it is worth noting that no multiaccess proto- col guarantees reliable delivery. Even in the absence of collisions, the receiver may have copied some of the frame incorrectly for various reasons. Other parts of the link layer or higher layers provide reliability.

❖ MULTIPLE ACCESS PROTOCOLS

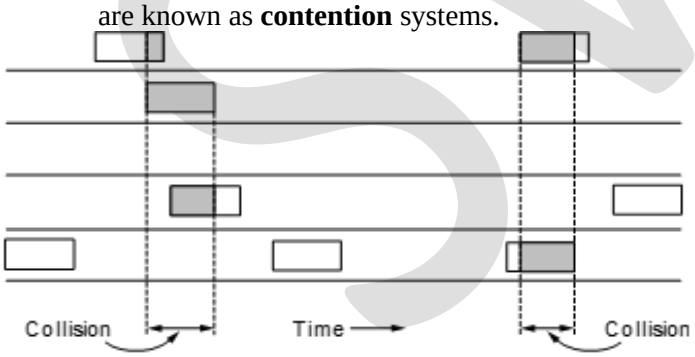
- Any algorithms for allocating a multiple access channel are known. In the following sections, we will study a small sample of the more interesting ones and give some examples of how they are commonly used in practice.

ALOHA

- The story of our first MAC starts out in pristine Hawaii in the early 1970s. In this case, ‘‘pristine’’ can be interpreted as ‘‘not having a working telephone sys- tem.’’ This did not make life more pleasant for researcher Norman Abramson and his colleagues at the University of Hawaii who were trying to connect users on re- mote islands to the main computer in Honolulu.
- The one they found used short-range radios, with each user terminal sharing the same upstream frequency to send frames to the central computer. It included a simple and elegant method to solve the channel allocation problem.
- Their work has been extended by many researchers since then (Schwartz and Abramson, 2009). Although Abramson’s work, called the ALOHA system, used ground- based radio broadcasting, the basic idea is applicable to any system in which uncoordinated users are competing for the use of a single shared channel.
- We will discuss two versions of ALOHA here: pure and slotted. They differ with respect to whether time is continuous, as in the pure version, or divided into discrete slots into which all frames must fit.

Pure ALOHA

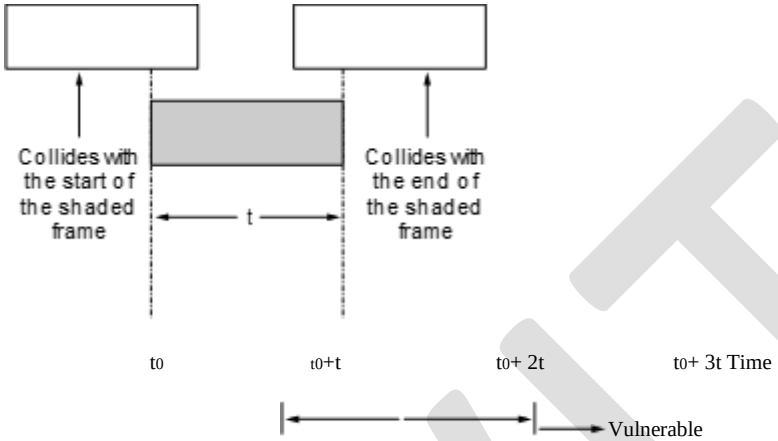
- The basic idea of an ALOHA system is simple: let users transmit whenever they have data to be sent. There will be collisions, of course, and the colliding frames will be damaged. Senders need some way to find out if this is the case. In the ALOHA system, after each station has sent its frame to the central computer, this computer rebroadcasts the frame to all of the stations
- If the frame was destroyed, the sender just waits a random amount of time and sends it again. The waiting time must be random or the same frames will collide over and over, in lockstep, Systems in which multiple users share a common channel in a way that can lead to conflicts are known as **contention** systems.



- A user is always in one of two states: typing or wait- ing. Initially, all users are in the typing state. When a line is finished, the user stops typing, waiting for a response. The station then transmits a frame con- taining the line over the shared channel to the central computer and checks the channel to see if it was successful.
- If $N > 1$, the user community is generating frames at a higher rate than the channel can handle, and nearly every frame will suffer a collision. For reasonable throughput, we would expect $0 < N < 1$.
- Let us further assume that the old and new frames combined are well modeled by a Poisson distribution, with mean of G frames per frame time. Clearly, $G \geq N$. At low load (i.e., $N \to 0$), there will be few collisions, hence few retransmissions, so $G \to N$. At high load, there will be

many collisions, so $G > N$.

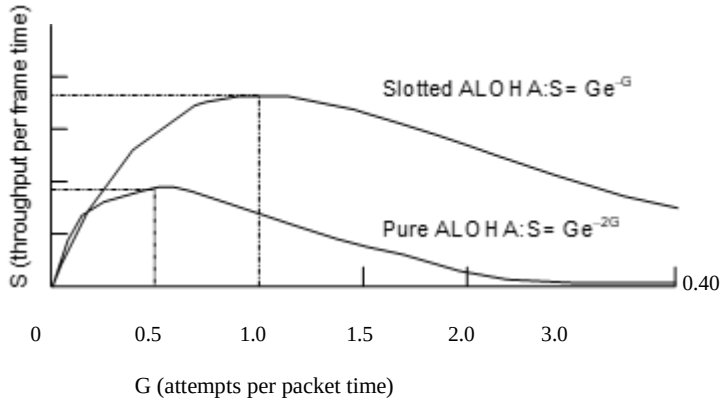
- If any other user has generated a frame between time t_0 and $t_0 + t$, the end of that frame will collide with the beginning of the shaded one.
- In fact, the shaded frame's fate was already sealed even before the first bit was sent, but since in pure ALOHA a station does not listen to the channel before transmitting, it has no way of knowing that another frame was already underway. Similarly, any other frame started between $t_0 - t$ and $t_0 + 2t$ will bump into the end of the shaded frame.



- The probability that k frames are generated during a given frame time, in which G frames are expected, is given by the Poisson distribution
- so the probability of zero frames is just e^{-G}. In an interval two frame times long, the mean number of frames generated is $2G$. The probability of no frames being initiated during the entire vulnerable period is thus given by $P_0 = e^{-2G}$. Using $S = GP_0$, we get
- $S = Ge^{-2G}$
- The relation between the offered traffic and the throughput is shown in Fig. 4-3. The maximum throughput occurs at $G = 0.5$, with $S = 1/2e$, which is about 0.184. In other words, the best we can hope for is a channel utilization of 18%. This result is not very encouraging, but with everyone transmitting at will, we could hardly have expected a 100% success rate.

➢ Slotted ALOHA

- Soon after ALOHA came onto the scene, Roberts (1972) published a method for doubling the capacity of an ALOHA system. His proposal was to divide time into discrete intervals called **slots**, each interval corresponding to one frame.
- In Roberts' method, which has come to be known as **slotted ALOHA**—in contrast to Abramson's **pure ALOHA**—a station is not permitted to send when- ever the user types a line.



0.30

0.20

0.10

Figure 4-3. Throughput versus offered traffic for ALOHA systems.

- Instead, it is required to wait for the beginning of the next slot. Thus, the continuous time ALOHA is turned into a discrete time one. This halves the vulnerable period. To see this, look at Fig. 4-3 and imagine the collisions that are now possible. The probability of no other traffic during the same slot as our test frame is then e^{-G}, which leads to
- $S = Ge^{-G}$

- As you can see from Fig. 4-3, slotted ALOHA peaks at $G = 1$, with a throughput of $S = 1/e$ or about 0.368, twice that of pure ALOHA. If the system is operating at $G = 1$, the probability of an empty slot is 0.368 (from Eq. 4-2). The best we can hope for using slotted ALOHA is 37% of the slots empty, 37% successes, and 26% collisions.
- Operating at higher values of G reduces the number of empties but increases the number of collisions exponentially. To see how this rapid growth of collisions with G comes about, consider the transmission of a test frame. The probability that it will avoid a collision is e^{-G}, which is the probabil- ity that all the other stations are silent in that slot.
- The probability of a collision is then just $1 - e^{-G}$. The probability of a transmission requiring exactly k attempts (i.e., $k - 1$ collisions followed by one success) is
- $P_k = e^{-G}(1 - e^{-G})^{k-1}$

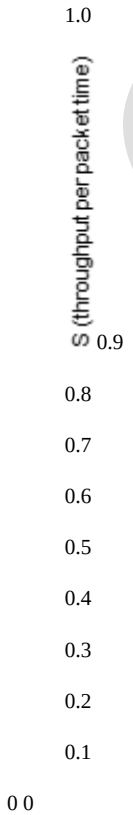
♦ Carrier Sense Multiple Access Protocols

- This low result is hardly surprising, since with stations transmitting at will, with- out knowing what the other stations are doing there are bound to be many collis- ions.
- In LANs, however, it is often possible for stations to detect what other sta- tions are doing, and thus adapt their behavior accordingly. These networks can achieve a much better utilization than $1/e$. In this section, we will discuss some protocols for improving performance.
- Protocols in which stations listen for a carrier (i.e., a transmission) and act accordingly are called **carrier sense protocols**.
- A number of them have been proposed, and they were long ago analyzed in detail. For example, see Kleinrock and Tobagi (1975). Below we will look at several versions of carrier sense proto-

cols.

♦ **Persistent and Nonpersistent CSMA**

- The first carrier sense protocol that we will study here is called **1-persistent CSMA** (**Carrier Sense Multiple Access**). That is a bit of a mouthful for the sim- plest CSMA scheme. When a station has data to send, it first listens to the chan- nel to see if anyone else is transmitting at that moment.
- If the first station's signal has not yet reached the second one, the latter will sense an idle channel and will also begin sending, resulting in a collision. This chance depends on the number of frames that fit on the channel, or the **bandwidth-delay product** of the channel.
- This protocol has better performance than pure ALOHA because both stations have the decency to desist from interfering with the third station's frame. Exactly the same holds for slotted ALOHA.
- A second carrier sense protocol is **nonpersistent CSMA**. In this protocol, a conscious attempt is made to be less greedy than in the previous one. As before, a station senses the channel when it wants to send a frame, and if no one else is sending, the station begins doing so itself.
- The last protocol is **p-persistent CSMA**. It applies to slotted channels and works as follows. When a station becomes ready to send, it senses the channel. If it is idle, it transmits with a probability *p*. With a probability *q* □ 1 □ *p*, it defers until the next slot. If that slot is also idle, it either transmits or defers again, with probabilities *p* and *q*.
- This process is repeated until either the frame has been transmitted or another station has begun transmitting. In the latter case, the unlucky station acts as if there had been a collision (i.e., it waits a random time and starts again). If the station initially senses that the channel is busy, it waits until the next slot and applies the above algorithm. IEEE 802.11 uses a refinement of p-persistent CSMA that we will discuss in Sec. 4.4.



..

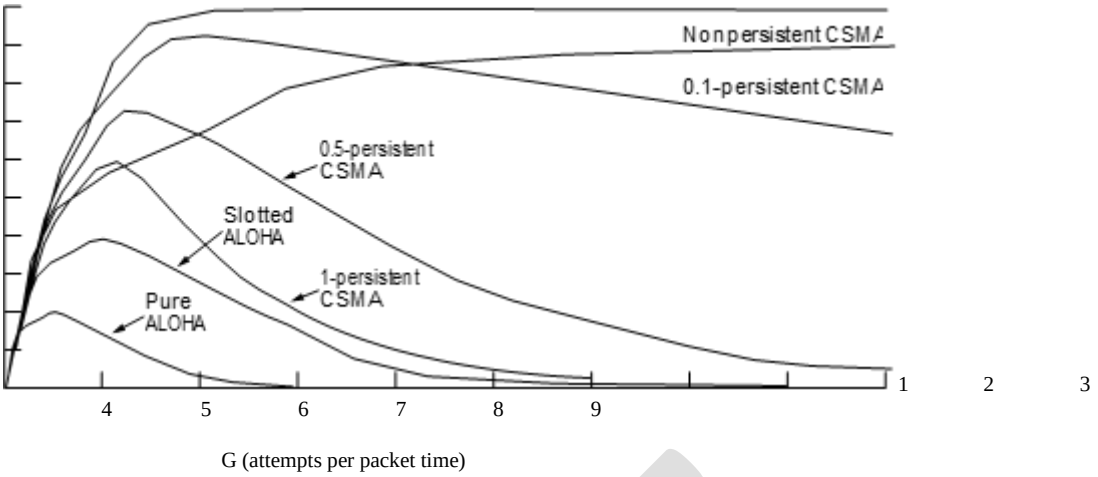


Figure 4-4. Comparison of the channel utilization versus load for various ran- dom access protocols.

Figure 4-4 shows the computed throughput versus offered traffic for all three protocols, as well as for pure and slotted ALOHA.

❖ CSMA with Collision Detection

- Persistent and nonpersistent CSMA protocols are definitely an improvement over ALOHA because they ensure that no station begins to transmit while the channel is busy.
- Another improvement is for the stations to quickly detect the collision and abruptly stop transmitting, (rather than finishing them) since they are irretrievably garbled anyway. This strategy saves time and bandwidth.
- This protocol, known as **CSMA/CD** (**CSMA with Collision Detection**), is the basis of the classic Ethernet LAN, so it is worth devoting some time to looking at it in detail. It is important to realize that collision detection is an analog proc- ess. The station's hardware must listen to the channel while it is transmitting.
- CSMA/CD, as well as many other LAN protocols, uses the conceptual model of Fig. 4-5. At the point marked t_0, a station has finished transmitting its frame. Any other station having a frame to send may now attempt to do so. If two or more stations decide to transmit simultaneously, there will be a collision.

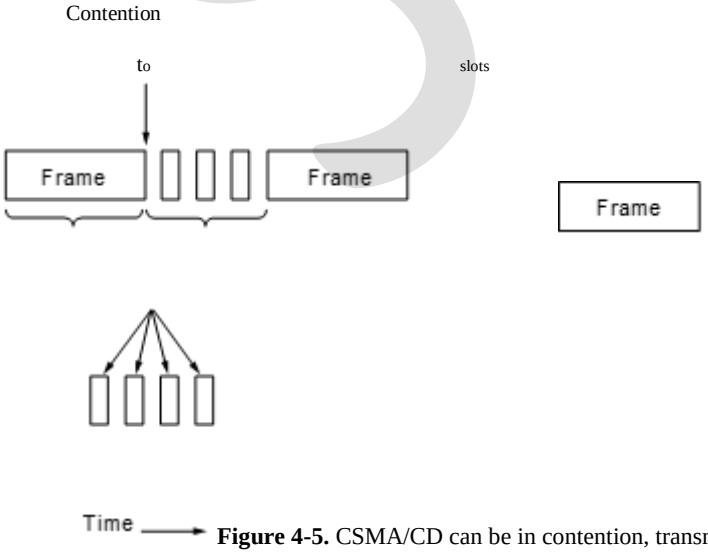


Figure 4-5. CSMA/CD can be in contention, transmission, or idle state.

- The minimum time to detect the collision is just the time it takes the signal to propagate from

one station to the other. Based on this information, you might think that a station that has not heard a collision for a time equal to the full cable propagation time after starting its transmission can be sure it has seized the cable.

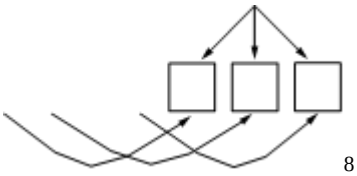
- Consider the following worst-case scenario. Let the time for a signal to pro- pagate between the two farthest stations be □. At t_0, one station begins trans- mitting. At t_0 □ □ □ □, an instant before the signal arrives at the most distant sta- tion, that station also begins transmitting.
- it detects the collision al- most instantly and stops, but the little noise burst caused by the collision does not get back to the original station until time 2□□ □. In other words, in the worst case a station cannot be sure that it has seized the channel until it has transmitted for 2□ without hearing a collision.
- The difference for CSMA/CD compared to slotted ALOHA is that slots in which only one station transmits (i.e., in which the channel is seized) are followed by the rest of a frame. This difference will greatly improve performance if the frame time is much longer than the propagation time.

♦ Collision-Free Protocols

- Although collisions do not occur with CSMA/CD once a station has unambi- guously captured the channel, they can still occur during the contention period. These collisions adversely affect the system performance, especially when the bandwidth-delay product is large, such as when the cable is long (i.e., large □) and the frames are short.
- Most of these protocols are not currently used in major systems, but in a rapidly changing field, having some protocols with excellent properties available for fu- ture systems is often a good thing.
- In the protocols to be described, we assume that there are exactly *N* stations, each programmed with a unique address from 0 to *N* □ 1. It does not matter that some stations may be inactive part of the time.

♦ A Bit-Map Protocol

- In our first collision-free protocol, the **basic bit-map method**, each con- tention period consists of exactly *N* slots. If station 0 has a frame to send, it trans- mits a 1 bit during the slot 0.
- In general, station *j* may announce that it has a frame to send by inserting a 1 bit into slot *j*. After all *N* slots have passed by, each station has complete knowledge of which stations wish to transmit.

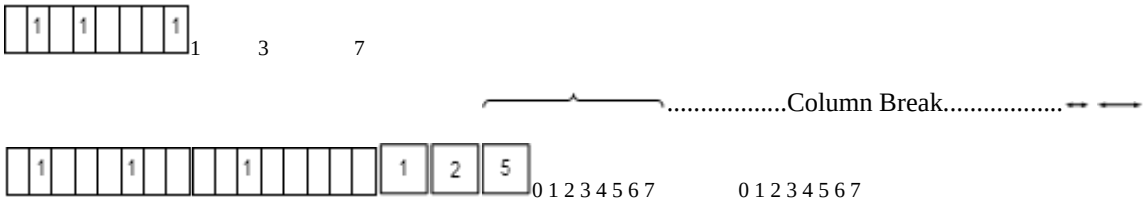


8 Contention slots Frames 8 Contention slots

1 d

0 1 2 3 4 5 6 7

.....Column Break.....



- Since everyone agrees on who goes next, there will never be any collisions. After the last ready station has transmitted its frame, an event all stations can easi- ly monitor, another *N*-bit contention period is begun.

- Protocols like this in which the desire to transmit is broadcast before the ac- tual transmission are called **reservation protocols** because they reserve channel ownership in advance and prevent collisions.

- Under conditions of low load, the bit map will simply be repeated over and over, for lack of data frames. Consider the situation from the point of view of a low-numbered station, such as 0 or 1. Typically, when it becomes ready to send, the ‘‘current’’ slot will be somewhere in the middle of the bit map.

- The prospects for high-numbered stations are brighter. Generally, these will only have to wait half a scan (*N*/2 bit slots) before starting to transmit. High- numbered stations rarely have to wait for the next scan.

- The channel efficiency at low load is easy to compute. The overhead per frame is *N* bits and the amount of data is *d* bits, for an efficiency of $d/(d + N)$.

- At high load, when all the stations have something to send all the time, the *N*- bit contention period is prorated over *N* frames, yielding an overhead of only 1 bit per frame, or an efficiency of $d/(d + 1)$.

- The mean delay for a frame is equal to the sum of the time it queues inside its station, plus an additional $(N - 1)d + N$ once it gets to the head of its internal queue. This interval is how long it takes to wait for all other stations to have their turn sending a frame and another bitmap.

Token Passing

- The essence of the bit-map protocol is that it lets every station transmit a frame in turn in a predefined order. Another way to accomplish the same thing is to pass a small message called a **token.** from one station to the next in the same predefined order. The token represents permission to send.
- In a **token ring** protocol, the topology of the network is used to define the order in which stations send. The stations are connected one to the next in a single ring. Passing the token to the next station then simply consists of receiving the token in from one direction and transmitting it out in the other direction.

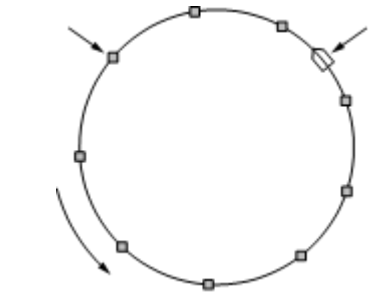


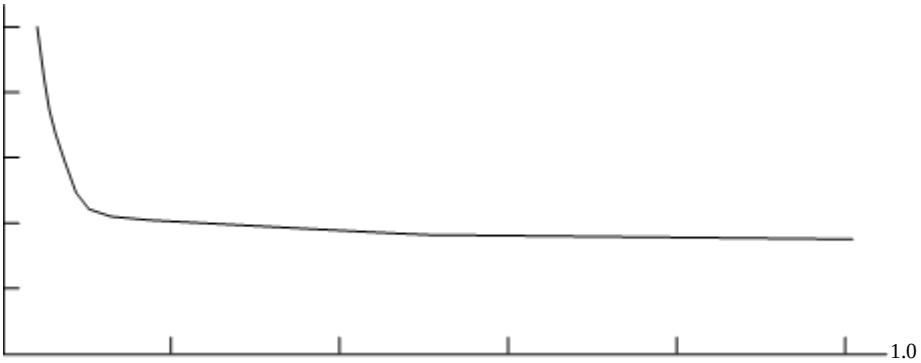
Figure 4-7. Token ring.

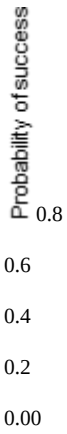
- The channel connecting the stations might instead be a single long bus. Each station then uses the bus to send the token to the next station in the predefined sequence. Possession of the token allows a station to use the bus to send one frame, as be- fore. This protocol is called **token bus**.
- In the 1990s, a much faster token ring called **FDDI** (**Fiber Distributed Data Interface**) was beaten out by switched Ethernet. In the 2000s, a token ring called **RPR** (**Resi- lient Packet Ring**) was defined as IEEE 802.17 to standardize the mix of metro- politan area rings in use by ISPs. We wonder what the 2010s will have to offer.

Binary Countdown

Limited-Contention Protocols

- The contention and collision-free protocols, arriving at a new protocol that used con- tention at low load to provide low delay, but used a collision-free technique at high load to provide good channel efficiency. Such protocols, which we will call **limited-contention protocols**, do in fact exist, and will conclude our study of car- rier sense networks.
- The only contention protocols we have studied have been symmet- ric. That is, each station attempts to acquire the channel with some probability, p, with all stations using the same p.
- Each has a probability p of transmitting during each slot. The probability that some station successfully acquires the channel during a given slot is the probability that any one station transmits, with probability p, and all other $k - 1$ stations defer, each with probability $1 - p$. This value is $kp(1 - p)^{k-1}$.





.....Column Break.....

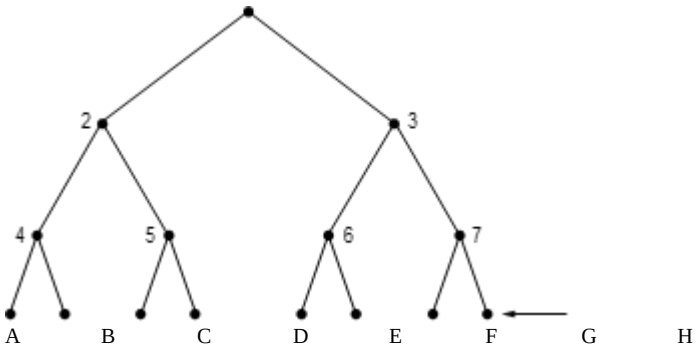
5 10 15 20 25

Number of ready stations

- They first divide the stations into (not necessarily disjoint) groups. Only the members of group 0 are permitted o compete for slot 0. If one of them succeeds, it acquires the channel and trans- mits its frame
- The limiting case is a single group containing all stations (i.e., slotted ALOHA). What we need is a way to assign stations to slots dynamically, with many stations per slot when the load is low and few (or even just one) station per slot when the load is high.

- **The Adaptive Tree Walk Protocol**

- A portion of each sample was poured into a single test tube. This mixed sample was then tested for antibodies. If none were found, all the soldiers in the group were declared healthy. If antibodies were present, two new mixed samples were prepared, one from soldiers 1 through *N*/2 and one from the rest. The proc- ess was repeated recursively until the infected soldiers were determined.



.....Column Break.....Stations

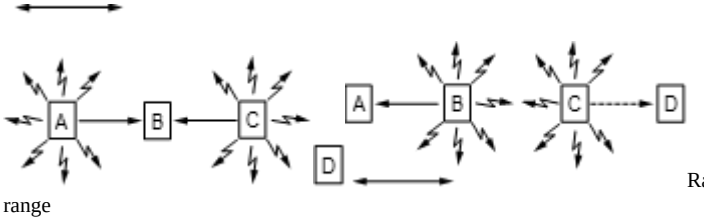
- In essence, if a collision occurs during slot 0, the entire tree is searched, depth first, to locate all ready stations. Each bit slot is associated with some particular node in the tree. If a collision occurs, the search continues recursively with the node's left and right children.
- When the load on the system is heavy, it is hardly worth the effort to dedicate slot 0 to node 1 because that makes sense only in the unlikely event that precisely one station has a frame to send. Similarly., one could argue that nodes 2 and 3 should be skipped as well for the same reason
- the expected number of them below a specific node at level i is just $2^{-i}q$. Intuitively, we would expect the optimal level to begin searching the tree to be the one at which the mean number of contending stations per slot is 1, that is, the level at which $2^{-i}q \approx 1$. Solving this equation, we find that $i \approx \log_2 q$.
- Numerous improvements to the basic algorithm have been discovered and are discussed in some detail by Bertsekas and Gallager (1992). For example, consid- er the case of stations *G* and *H* being the only ones wanting to transmit.

♦ Wireless LAN Protocols

◊ A system of laptop computers that communicate by radio can be regarded as a wireless LAN, as we discussed in Sec. 1.5.3. Such a LAN is an example of a broadcast channel. It also has somewhat different properties than a wired LAN, which leads to different MAC protocols.

◊ There is an even more important difference between wireless LANs and wired LANs. A station on a wireless LAN may not be able to transmit frames to or re- ceive frames from all other stations because of the limited radio range of the sta- tions.

◊ The naive approach to using a wireless LAN might be to try CSMA: just listen for other transmissions and only transmit if no one else is doing so. The trouble is, this protocol is not really a good way to think about wireless because what mat- ters for reception is interference at the receiver, not at the sender.



(a) (b)

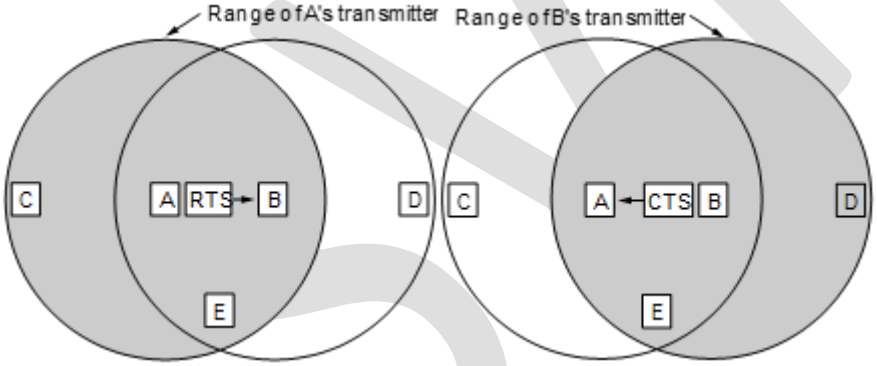
Figure 4-11. A wireless LAN. (a) *A* and *C* are hidden terminals when trans- mitting to *B*. (b) *B* and *C* are exposed terminals when transmitting to *A* and *D*.

- First consider what happens when *A* and *C* transmit to *B*, as depicted in Fig. 4-11(a). If *A* sends and then *C* immediately senses the medium, it will not hear *A* because *A* is out of range. Thus *C* will falsely conclude that it can transmit to *B*. If *C* does start transmitting, it will interfere at *B*, wiping out the frame from A.
- We want a MAC protocol that will prevent this kind of collision from happening because it wastes bandwidth. The problem of a station not being able to detect a potential competitor for the medium because the competitor is too far away is called the **hidden terminal problem**.
- In fact, such a transmission would cause bad reception only in the zone between *B* and *C*, where neither of the intended receivers is located. We want a MAC protocol that prevents this kind of deferral from happening because it wastes bandwidth. The problem is called the **exposed terminal problem**.
- We want this concurrency to happen as the cell gets larger and larger, in the same way that people at a party should not wait for everyone in the room to go silent before they talk; multiple conversations can take place at once in a large room as long as they are not directed to the same location.

- An early and influential protocol that tackles these problems for wireless LANs is **MACA** (**Multiple Access with Collision Avoidance**) (Karn, 1990). The basic idea behind it is for the sender to stimulate the receiver into outputting a short frame, so stations nearby can detect this transmission and avoid transmitting for the duration of the upcoming (large) data frame.

- MACA is illustrated in Fig. 4-12. Let us see how *A* sends a frame to *B*. *A* starts by sending an **RTS** (**Request To Send**) frame to *B*, as shown in Fig. 4-12(a). This short frame (30 bytes) contains the length of the data frame that will eventu- ally follow.
- Then *B* replies with a **CTS** (**Clear To Send**) frame, as shown in Fig. 4-12(b). The CTS frame contains the data length (copied from the RTS frame). Upon receipt of the CTS frame, *A* begins transmission.

receipt of the CTS frame, *A* begins transmission.



(a) (b)

Figure 4-12. The MACA protocol. (a) *A* sending an RTS to *B*. (b) *B* responding with a CTS to *A*.

- Therefore, it hears the RTS from *A* but not the CTS from *B*. As long as it does not interfere with the CTS, it is free to transmit while the data frame is being sent. In contrast, *D* is within range of *B* but not *A*. Station *E* hears both control messages and, like *D*, must be silent until the data frame is complete.
- Despite these precautions, collisions can still occur. For example, *B* and *C* could both send RTS frames to *A* at the same time. These will collide and be lost. In the event of a collision, an unsuccessful transmitter (i.e., one that does not hear a CTS within the expected time interval) waits a random amount of time and tries again later.