An Introduction to Computer Communications

Larry Hughes, PhD

Department of Electrical and Computer Engineering
Dalhousie University
Halifax, Nova Scotia
Canada.

Whale Lake Press
P.O. Box 631, Station M
Halifax, Nova Scotia
Canada.

The editor of this book was Sandy Cook.

Printed and bound by *The PRINTER*, Halifax, Nova Scotia.

First printing.

Whale Lake Press
P.O. Box 631, Station M
Halifax, Nova Scotia
B3H 3C3
CANADA

Table of Contents

Preface

Moving data has always been one of the fundamental operations supported by computers. At the hardware level, a bit stored in a flip-flop can be moved to another flip-flop for subsequent processing. Machine instructions exist to copy groups of bits from memory to a CPU register; for example, in the Motorola 6811, instructions such as:

```
LDAA     Count
```

move (copy) the contents of memory location `Count` to the CPU accumulator verb!ACCA!. (A CPU's **fetch** cycle is another example of moving bits; however, rather than moving data, the fetch cycle moves instructions from memory to the CPU.) Most computers have instructions for moving data within the CPU and from the CPU to memory; once again, using the Motorola 6811 as the example:

```
TAB              ; Transfer contents of ACCA to ACCB
STAB     Next
```

Other, move advanced CPUs, such as the Intel 80x86 family, have instructions to handle the transfer of blocks of data from one set of contiguous memory locations to another:

```
MOV      SI,0200  ; Source address
MOV      DI,0300  ; Destination address
MOV      CX,0020  ; Word count
REPZ              ; Repeat next instruction while CX is not zero
MOVSW             ; [DI++] = [SI++]; CX--
```

Not surprisingly, high-level computer languages also support the transfer of data from one location to another. For example, in C, the contents of one structure can be copied to another simply using an assignment statement (*lvalue* = *rvalue*):

```
employee[new_emp_no] = new_employee;
```

Without exception, the movement of data is not restricted to the CPU and primary memory. Data can be moved between external devices (or peripherals), the CPU, and memory:

- An editor copies files stored on a device (such as a disk) into memory, operations take place upon the file, and, if required by the user, copied back to disk.
- Characters entered on a keyboard can result in an interrupt, causing the CPU to read the keyboard register to obtain the value of the character.

In these, and all other applications involving the movement of data, there is a **source**, a **destination**, and an **intermediary** that is responsible for holding or transferring the data:

For example, when copying from memory (the source) to the CPU (the destination), the data is transferred over a **bus** (the intermediary).

When an organization acquires more than one computer, there will ultimately come a time when someone wants to transfer (i.e., move) data between the two machines. The simplest, and probably most secure, involves copying the data onto a medium (such as a magnetic tape or disk) from one computer, and physically carrying the medium to the second computer. In such cases, a specific set of operations would have to be performed: mounting the medium at the first computer, copying the data to the medium, transferring it to the second computer, mounting the medium, and copying it to the second computer.

In most organizations, physically carrying data from one computer to another has long been replaced by an electronic medium such as an bus or ring network. However, even when replacing the mountable medium with an electronic one, rules still apply to the transfer. These rules, known as a **protocol** are central to all successful communications between computers.

The term **computer communications** refers to the movement of data between computers. It covers all aspects of the communication: how the data is transmitted, the different types of media, how the communication is controlled, and what to do should the communication fail. Computer communications is not concerned with the contents of the data that is transmitted; that is a higher level function and is handled by application software that uses the underlying communications system.

This book presents a structured approach to the study of computer communications. Communications between computers are examined in several phases, beginning with communications over an unshared medium. Given the growth of shared communication media, the book discusses various types of local area networks, then considers how communications can take place within a single organization across different networks. Communications across networks operated by more than one organization is then considered, first in terms of wide area networks, then as the internetwork technology. Finally, the book looks at various layered architectures.

The book is divided into seven Parts.

Part 1 introduces the reader to the basic concepts and much of the terminology associated with communications in general and computer communications in particular. The material presented in Chapter 1 is used throughout the remainder of the text. This Chapter concludes, as do all subsequent ones, with a series of exercises covering the material presented in the Chapter.

Prior to the rapid growth of networking technologies, most communications between computers across an unshared channel between pairs of devices. Although times are changing, Part 2 examines two techniques for communications across an unshared channel since many of the concepts are still applicable to today's networking technologies.

Asynchronous communications are introduced in Chapter 2 using the asynchronous communication hardware available on all PCs. The chapter presents a detailed examination of how asynchronous communications are achieved.

Chapter 3 examines techniques for sending information that minimize the overheads associated with asynchronous communications by reducing the amount of control information sent, while simultaneously improving the reliability of the data transfer.

One of the biggest headaches in the detection and correction of errors in any communication system (shared or unshared) is trying to determine what is actually being sent across the communication channel between the communicating devices. This is true in the commercial world as well as in a course on computer communications.

Part 3 considers techniques for sharing a single channel amongst several communicating computers. Moving away from a dedicated, unshared channel comes at a price – each message must contain a means whereby the destination can be identified. Chapter 4 is the first of three chapters that consider methods of sharing a single communication channel between a number of communicating devices. The remaining two chapters consider other means of sharing a channel using a **local area network**. In Chapter 5, bus local area networks are discussed in light of existing commercial local area networks such as the Ethernet. The 3COM 3C5X9 network interface card is examined in detail.

Chapter 6 builds upon the material covered in the previous Chapter to explain ring networks and how they operate. The Chapter initially considers the token bus and then progresses into ring local area networks.

To assist in error detection and correction, as well as to monitor what is taking place on the communication channel, commercial tools known as line analysers and network analysers are available. These and other testing tools are discussed in Chapter 7.

Computer communications in many large organizations requires the use of more than one network. Multiple interconnected networks in a single organization are referred to as **enterprise networks** or **intranetworks**. Part 4 consists of one chapter, Chapter 8 which examines some of the problems associated with interconnecting networks. The Chapter includes examples of existing internetworking standards.

All communications can fail due to hardware or software errors. If reliable communications are to be supported (and in most applications today, reliability is a requirement), the network must be sufficiently robust to identify the error and recover from it. In Part 5, four approaches to error detection and recovery are considered. The simplest reliable protocol, **stop-and-wait**, is discussed in Chapter 9, while Chapter 10 explains the **go-back-N** protocol, and extension of the stop-and-wait protocol. Error recovery in the go-back-N protocol is costly, and has resulted in the development of another type of protocol known as **selective retransmission**; Chapter 11 examines selective retransmission. A fourth type of protocol which incorporates both error detection and recovery in the receiver (i.e., the transmitter is not required to retransmit anything), is discussed in Chapter 12.

Part 6 considers the problem of communicating across networks other than intranetworks. Two types of network are considered: the wide area network and the internetwork. Wide area networks, their architecture and protocols are discussed in Chapter 13. A generic internetwork is developed in Chapter 14, starting with the need for an internetwork, leading to today's Internet.

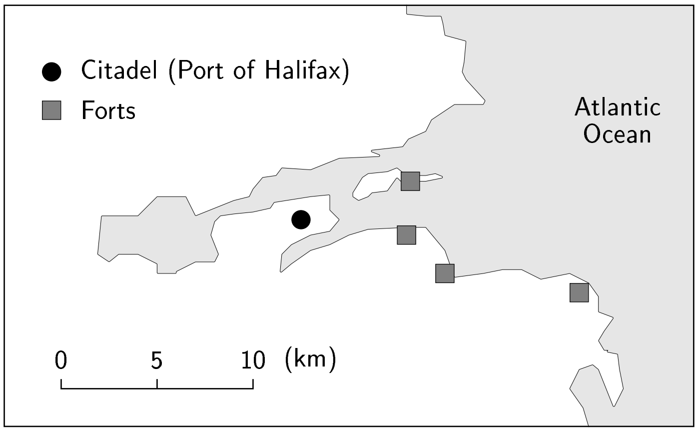
Support for communications between **processes** on different computers is given in Part 7. Chapter 15 discusses interprocess communication, layering, and various layered architectures.

The ASCII character set can be found in Appendix A.

Part 1

Background

After the American Revolution, the one major seaport on the eastern coast of North America to which the British Navy had access was Halifax. Since the British did not want to lose the rest of their North American colonies, it was imperative that the port of Halifax be defended. Accordingly, a form of 'early warning system' was installed, consisting of a series of forts stretching from the Atlantic Ocean to the port of Halifax. These forts could relay information about a ship sighted off the coast to the Citadel (a fortress) in Halifax in about 15 minutes:



Flags were used to communicate between the forts: whenever a ship was spotted off the coast, this information was encoded as a unique sequence of flags and then passed from one fort to the next until it reached the Citadel. If the officers in the Citadel were to react to the information, it was important that the encoded information be sent and received correctly by each fort up the coast. For example, the results could have been disastrous if a fort identified a ship as a neutral man-of-war only to have the information corrupted to indicate a French man-of-war. Similarly, if an Admiral's ship was identified, but fog caused the information to be read as 'the incoming ship is a scow', the Admiral might not have been too pleased with the welcome he received.

This seemingly simple example illustrates all of the major issues surrounding data communications:

- information must be **encoded** before it can be **transmitted**. The sailors watching for incoming ships had to decide amongst themselves the type of ship, and then translate that information into a set of flags.

- **rules** must be in place to ensure that any information sent is received correctly. At a minimum, both the transmitter and receiver must agree upon the same method of translating the encoded information. In 18^{th} century Halifax, all of the forts would be required to use the same flags and interpret them in the same way.
- when an error is **detected**, there must be an agreed set of procedures to follow which allow the errors to be **corrected**. If errors are permitted to propagate through the system (or, for example, up the chain of command from the officers in the forts to those in the Citadel) the information is of no value and cannot be used or acted upon safely.
- mechanisms must be available to ensure that both the transmitter and receiver remain **synchronized** (or **in-step**). Should the communicating parties lose synchronization, information may be lost, with disastrous results (for example, an enemy warship might enter the harbour undetected).

The remainder of Part 1 considers some of the terminology associated with the computer communications. Chapter 1 introduces much of the terminology used in computer communications both through a series of examples and by considering how humans communicate.

Chapter 1

Basic Concepts and Terminology

1.1 Introduction

Communications, whether between humans, animals, or computers, involves the **transfer of information**. Considering that people have developed computers, it should not be surprising that many of the problems that the designers of data communication systems face are, in many respects, similar to those encountered when people communicate on a day-to-day basis. For example:

- when two people want to talk, some agreement must be made to ensure that the person to begin talking eventually stops so that the other person has a chance to respond.
- when a person misses part of what was said in a conversation, mechanisms should exist which allow the lost information to be repeated.
- when a person finishes speaking, it is often considered polite (and sometimes necessary) to ask whether what has been said was heard and comprehended.

In this Chapter, some of the general terms and concepts associated with communications are introduced. In all cases, everyday examples are used to reinforce the ideas.

1.2 Entities

Communications, unless otherwise stated, are assumed to be between pairs of objects, often called **entities**. The transfer of information occurs across a **channel** (sometimes referred to as a **line**); there are two types of channel:

- a **simplex** channel is one in which communication can occur in one direction only, and
- a **duplex** channel is one in which a communication can occur in both directions simultaneously.

The channel can be any **medium**, including air, copper wire, or optical fibres. At any moment, an entity can be:

- a **transmitter**, that is, it sends information.
- a **receiver**, that is, it receives information.
- *both* a **transmitter** and a **receiver**; that is, it can both transmit and receive information simultaneously.

There are two broad categories that describe the type of communication that can take place between entities:

- a communication that permits either entity to transmit, but not simultaneously, is known as a **half-duplex** communication. A typical half-duplex communication involves one entity transmitting its information, while the other receives. The rôles are then reversed, and the entity that was originally receiving is now transmitting (while the original transmitter is now receiving). As shown in Figure 1.1, the reversing of rôles is known as **line turnaround**. (In this and subsequent diagrams, 'T' denotes the transmitting entity and 'R' denotes the receiving entity.)

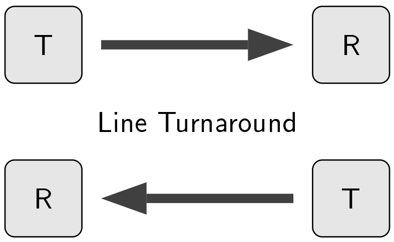


Figure 1.1: Half-duplex communication and line-turnaround

  An example of a half-duplex communication is a 'tin-can telephone' consisting of two tin cans connected by a piece of string. At any point in time, only one person can speak (i.e., transmit) while the other listens (i.e., receives). Line turnaround requires the transmitter to inform the receiver that there is no more information to be transmitted and that the receiver can now transmit.

  A half-duplex communication can be implemented using a pair of simplex channels (one for each direction) or a duplex channel. In either case, both entities must agree to a mechanism whereby line turnaround can be signalled and their rôles reversed.

- the second category of communication permits either entity to transmit and receive simultaneously, and is known as a **full-duplex** communication. Full-duplex communications are generally used when both entities have large volumes of data to send or when the transmitter wants to determine if the information sent has been received correctly (without having to perform a line turnaround). A representation of a full-duplex communication is given in Figure 1.2.

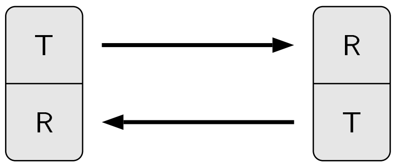


Figure 1.2: A full-duplex communication

Examples of full-duplex communications include sending mail (a person might send a letter to a friend and receive one from the same friend 'at the same time'), and talking to someone (both people might attempt to talk at the same time).

Full-duplex communications can be realized in a number of ways. If a duplex channel is used in a full-duplex communication, each entity is allocated a distinct part of the channel (for example, in Figure 1.3 one entity may transmit on frequency X and receive on frequency Y, while the other entity would transmit on frequency Y and receive on frequency X).

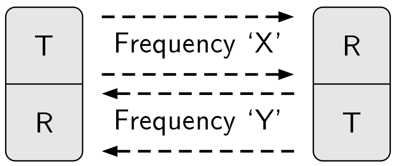


Figure 1.3: Implementing full-duplex with different frequencies on a single channel

It is possible to support a full-duplex communication using of a pair of simplex channels. In this situation, as illustrated in Figure 1.4, each entity transmits on one channel and receives on the other.

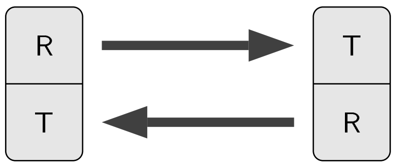


Figure 1.4: Implementing full-duplex communication using a pair of simplex channels

In a communication involving humans, it is obvious who the entities are: the people involved in the communication. However, in a communication involving computers, it is not so easy to determine the entities, since an entity may be the application processes (i.e., the software requiring the communication), the support software (i.e., the software supporting the communication requirements of the application processes, perhaps needing the communication facilities offered by the underlying hardware), or the processors (computers) themselves.

For example, a communication between two processes may involve one or more processors. In the single processor case, only the processor's communication software is necessary. However, if the processes reside on separate interconnected processors, both communication software and hardware is needed.

The structuring of software so that one level uses the services of an adjacent, lower level is known as **layering**. Layering will be discussed further in subsequent Chapters.

1.3 Protocols

At the start of this Chapter, a number of examples of communications between humans were presented. These examples illustrated the need for a set of rules to ensure that the communication would proceed in spite of problems such as people attempting to speak simultaneously and information being misunderstood or missed entirely.

Other communicating entities, such as computers or processes, also require rules or **protocol**s to ensure that the communication can proceed. Protocols are intended both to control the communication between the stations as well as to define certain characteristics about the communication. Regardless of what the entity is, all entities participating in the communication must agree on the protocol or the communication may fail. For example, if a full-duplex protocol will probably fail if the communicating entities are connected by a single simplex channel.

Broadly speaking, any communication protocol can be discussed in terms of the following:

Coding of Information: how the information is represented between the various entities.

Control: how the communication is controlled by the entities involved in the communication.

Error checking and recovery: how the entities ensure that the information is sent and received correctly.

Channel utilization: how efficiently the channel is used by the communicating entities.

Synchronization and Timing: how the entities remain 'in-step' during the progress of a communication.

Transparency: how the mechanisms supporting the communication are hidden from the entities.

1.3.1 Coding of Information

For a communication to take place, both the transmitting and receiving entities must agree upon a common language, written or verbal. For example, for a person in England to carry on a conversation with a person in, say, France, requires that either the person in England know how to speak French, or the person in France know how to speak English, or both agree to speak a third language.

Although there is no universal **standard** for the coding of information in computers, all manufacturers agree upon the value and structure of the **bit**: a single cell having the value of zero or one.

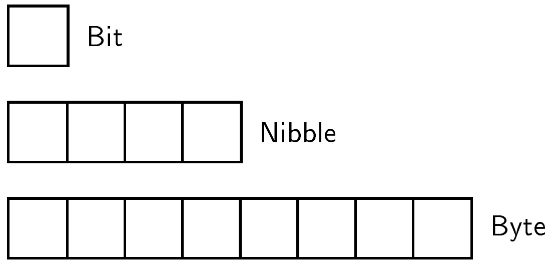


Figure 1.5: The bit, nibble, and byte structures

Bits can be grouped into four-bit quantities known as **nibbles**, and eight-bit quantities known as **bytes**. Figure 1.5 shows the organization of these three structures.

The description of the byte data structure depends entirely upon the manufacturer, although some standards do exist (see below). For example, some manufacturers label bytes from `1` to `8`, while others label bytes from `0` to `7`. Similarly, the labelling may run in ascending order from right to left (i.e., from `0` to `7`, the rightmost bit being `0`, and the leftmost, `7`), while other systems use descending order from right to left (i.e., the rightmost bit being `7`, and the leftmost bit, `0`).

The value and interpretation of the byte depends, once again, on the manufacturer. If bytes are considered to be **unsigned**, their possible values are `0` through `255`, however, if they are **signed**, the range of values is `-128` through `+127`.

Bytes are also used to represent **characters**: the two most common character codes used by computer manufacturers are:

EBCDIC (or Extended Binary Coded Decimal Interchange Code) a code made popular by IBM in their large mainframe computers in the 1960s, and

ASCII (or American Standard Code for Information Interchange) an alternate standard adopted by almost all non-IBM computer manufacturers; now so widespread that even IBM uses it for their PCs.

Appendix C contains character code tables for the 5-bit Baudot code, 7-bit ASCII, and 8-bit EBCDIC.

Not all bytes are intended to represent textual, printable characters; some bytes are used to convey control information such as line feed, carriage return, and form feed. For example, in the table of seven-bit ASCII bytes, the first two columns contain **control characters**, while the remaining six columns (with the exception of *DEL* at the bottom right of the table), are used to represent printable (or **graphic**) characters.

Although most communications usually occur at the bit or byte level, applications may deal with other data structures that comprise of groupings of bytes. Depending upon the manufacturer, a **word** may be a 16-bit (two-byte) or a 32-bit (four-byte) data structure. Floating point numbers, consisting of a fractional part (the **mantissa**), and an **exponent**, may be stored and interpreted in a variety of ways, depending upon the precision offered by the manufacturer, and the number of bits available to represent the number (floating point numbers can be 32-bit, 64-bit, or even 128-bit quantities). Fixed point (or **Binary Coded Decimal**, **BCD**) numbers can be stored in one of several different ways:

Zoned decimal format has each digit in the BCD number occupying a single byte, the leftmost nibble, the **zone nibble**, contains '1111', while the rightmost contains the digit (one of 0000 through 1001). The sign (one of 1100, positive; 1101, negative; or 1111, unsigned, assumed positive) is stored in the leftmost nibble of the low-order digit. Figure 1.6 illustrates how -218 would be stored.

1111 0010	1111 0001	1101 1000

Figure 1.6: Storing -218 as a zoned decimal

Packed decimal format removes the zone nibble from each byte and stores each digit in adjacent nibbles. The sign nibble is the same as for zoned decimal but is stored in the rightmost nibble of the low-order byte. The internal representation of -218 stored in packed decimal is shown in Figure 1.7.

0010 0001	1000 1101

Figure 1.7: Storing -218 as a packed decimal

How information is represented on different computers is an important issue in data communications because unless the entities involved in the communication can agree upon how the information is coded, there can be no meaningful exchange of information. For example, if a file of ASCII information is transmitted to an EBCDIC computer, unless some form of translation is made available (i.e., the ASCII bytes are translated into EBCDIC or the EBCDIC computer can interpret ASCII), the information received by the EBCDIC computer may be meaningless.

When a memory location is accessed, the order of the bytes returned also varies from manufacturer to manufacturer. For example, in the memory example give in Figure 1.8, accessing a two-byte word starting at location 0000 will return *either* FF0E or 0EFF, depending upon the way the manufacturer decides to order memory in multiple byte structures.

0000	FF
0001	0E
0002	14
0003	2A

Figure 1.8: Sample memory

A computer's **byte-ordering** is important, especially when dealing with interconnected, communicating computers. The relationship between the structure's memory location and its most and least significant bytes is shown in Table 1.1. If the low-order memory location has the most-significant byte, then the ordering is said to be **big-endian**; whereas storing the most-significant byte in the high-order memory location is said to have **little-endian** ordering. From this, it should be clear that `FF0E` is a big-endian ordering and `0EFF` is little-endian ordering.

The same rules apply to four-byte structures. A big-endian four-byte structure from location `0000` is `FF0E142A`; the little-endian structure is `2A140EFF`.

Table 1.1: Big-endian vs little-endian byte ordering

Low-order memory location	High-order memory location	Byte Ordering
Most-significant byte	Least-significant byte	Big-endian
Least-significant byte	Most-significant byte	Little-endian

If data is transmitted between a big-endian and a little-endian computer as a series of 8-bit ASCII bytes, the hardware ensures that each byte is read from and written to memory in its proper order. However, if the information is sent as a series of 16-bit words, the receiving processor will receive the bytes in the reverse order. Similar problems can occur when attempting to transmit 16-bit integers as pairs of bytes. In situations such as these, a common byte ordering must be agreed upon (this is discussed in more detail in Chapter 15).

1.3.2 Control

When two people talk, there are many techniques and cues that are used to signal whether the information has been understood, if the information should be repeated, and when the other person can speak. These signals are not usually part of the information that is being conveyed, but the signals are necessary to allow the communication to continue. For example, to gain someone's attention, a person might prefix what they are about to say with the second person's name; however, the person's name has no bearing on the actual information being conveyed. Other types of control information exist as well: in order to determine if the receiver has actually received (and understood) the information, the speaker may end what has been said with a question, expecting the receiver to respond to it. The additional information that is used to control the communication is known as **control information**. (To distinguish between the information being transmitted and the control information, the non-control information is often referred to as a **message**.) The extra information required to ensure that the communication can continue is known as an **overhead**.

Although control information does impose an overhead on all messages sent in a communication, the overhead is intended to ensure that the communication can proceed and that the message is received correctly. In a communication between two (computing) entities, messages are rarely, if ever, sent without some type of control information. Figure 1.9 shows how a message, plus the control information, is sent in a **frame** or a **packet**. Both frames and packets must conform to a standard agreed upon by all entities in the communication.

What is transmitted as control information depends, in part, upon the protocol being used. At a

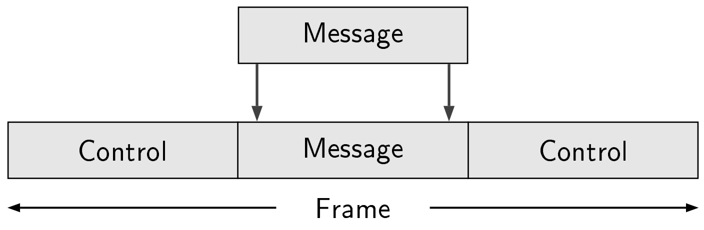


Figure 1.9: The relationship between a message and its control information

minimum, the control information should signal the receiver that a message is about to begin and that the message has ended. In situations where there may be many entities that could accept a message, the control information could contain a **destination identifier** which identifies the intended destination entity. A **source identifier** is usually included to permit the destination entity determine the transmitter of the message.

As an analogy, the frame can be considered an envelope into which the message (a letter) is placed. The destination address is the address on the front of the envelope and the source address is the return address.

Some protocols may require that responses be generated at certain intervals during a communication. These responses, which are also a form of control information, must be distinguishable from the information within the frame, otherwise it will be impossible for the entity receiving the frame to determine whether it is a message or control information.

1.3.3 Error Checking and Recovery

During a communication, an entity may not receive the incoming message correctly. At this point it is necessary for the receiving entity to indicate to the transmitting entity that the message was not received correctly. For example, when two people are speaking, if a person does not understand what is being said or missed what was being said because of some form of interference (such as a loud noise), the speaker may be asked to repeat what has just been said.

When a message is damaged or lost because of a **fault** on the communication channel, an **error** is said to have occurred. If a communication is to be reliable, the entities involved in the communication should be checking each message for errors, and if an error is detected, have an agreed upon procedure whereby the damaged or lost message can be recovered.

Errors can occur on the communication channel connecting computers through any one of a variety of faults, including:

Signal Attenuation. As a signal is transmitted through a medium, it must overcome resistances in the medium and as such may become weakened to the point where is it no longer possible to determine the meaning of the signal. For example, consider the problems involved in trying to determine what someone is saying if they stand several hundred metres away and talk in a whisper.

Signal attenuation can be overcome by either boosting the initial signal strength (i.e., getting the person to talk louder) or including amplifiers in the communication channel. There is a trade-off here: the signal cannot be boosted to the point where it is unintelligible or distorted to the receiver (consider the effect of yelling in someone's ear).

Noise. Changes in signal strength due to external interference or anomalies within the communication medium are known as **noise**. There are a number of different categories of noise:

- **white** (or **thermal**) noise, is a background interference caused by the motion of electrons in the communication medium due to temperature. Unless the communication can take place at absolute zero, the interference associated with thermal noise cannot be eliminated.
- **intermodulation noise** occurs when a number of communications share the same channel (as in a full duplex communication, see section 1.2, above) and one frequency interferes with another. For example, in Figure 1.10, a station's transmission hardware is faulty, producing signals at the wrong frequency, causing intermodulation noise. In this case, station B is to transmit signals at frequency Y, but instead, signals are produced at frequency X, thereby interfering with the transmission from station A, also transmitting at frequency X.

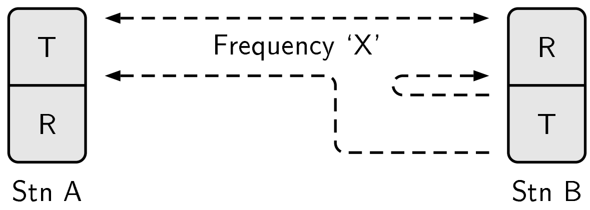


Figure 1.10: Intermodulation noise causing a frequency shift

- signals that are sent electrically across wires can be subject to **electromagnetic interference** or **impulse noise** caused by events such as lightning or power surges.

In addition to the above, a communication can be curtailed if the transmission medium is physically destroyed (for example, in a fire).

The recovery from an error typically involves the **retransmission** of the message, that is, the transmitter begins retransmitting from the first message that the receiver missed. In many protocols, there is a limit to the number of times the transmitter will retransmit the same message. Should this limit be exceeded, the transmission is terminated, since the flow of information approaches zero as the entire communication is taken up with retransmissions of the same message and the receiver is not obtaining any useful information.

Similarly, recovery from an error may make it necessary to have mechanisms to ensure that both entities can remain *in-step* with each other and that the transmitter does not flood the receiver with too much information. For example, during a conversation, the person speaking (the transmitter) often watches the person listening for cues to indicate that what has been said is understood. If the speaker is talking too fast, the person listening may request that the speaker slow down. This is known as **flow control**.

Computers can also use flow control to limit the amount of information flowing across a channel. For example, simple terminal-host communications often use the control characters X-OFF and X-ON for flow control. A person typing X-OFF (CTRL-S) will cause the transmitting host to stop sending, thereby allowing the person to read what is on the terminal's screen, and typing X-ON (CTRL-Q) which restarts the output. More advanced protocols use various control sequences to achieve the same effect; these protocols will be discussed in subsequent Chapters.

Frames (or packets) sent from a source to a destination can be lost or damaged, either through faults on the communication channel or the destination entity losing frames because of processing overheads. The fact that frames can be lost or damaged means that the protocol should support mechanisms that permit the destination to determine if the incoming frame is in error or out-of-sequence (implying that a frame has been lost), such as:

- out-of-sequence frames can be identified if each frame is sent with additional control information known as a **sequence number**. For example, if each frame is sent with a unique number and both the source and destination agree upon the numbering scheme (typically a series of increasing numbers: `0`, `1`, `2`, and so on), then if frames `0`, `1`, and `2` are transmitted, but only `0` and `2` are received, the destination 'knows' that frame `1` must have been lost.

- both the source and the destination must use the same algorithms for determining the next sequence number and agree upon the layout of the control information within the frame.

Once an error has been detected by the destination, it is standard practice to attempt to recover from it. In Figure 1.11, a typical recovery is shown; one that entails having the destination send some form of error indication to the source, which may then retransmit the message.

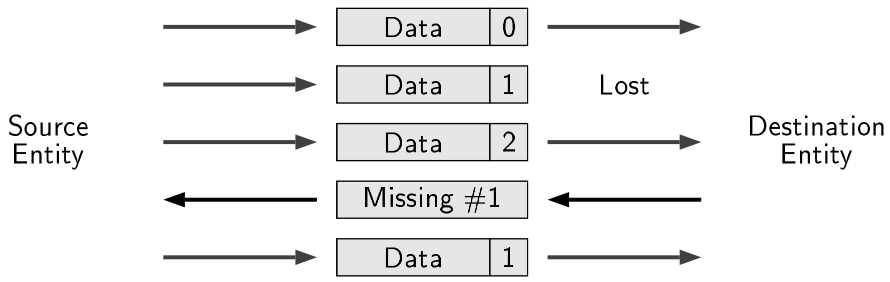


Figure 1.11: Error recovery using sequence numbers

Although the mechanism for recovery may seem simple enough, there are other problems which can occur, including:

- how does the protocol handle the situation in which the error indication message is lost?

- what procedures should be taken if the message sent from the source is always received in error?

Techniques for overcoming these and other problems will be discussed in subsequent Chapters.

1.3.4 Channel Utilization

In a communication, the information flowing between the entities must utilize some type of communication channel. There are many different media which can be used as a communication channel, for example, when two people talk on a telephone, the communication channel is the telephone system. Ideally, communication channels should be used as efficiently as possible, maximizing the amount of information passing from one entity to the other (the amount of useful information that flows between the transmitter and the receiver is known as **throughput**).

As an example of channel utilization (or throughput), consider the delivery of letters to someone's home by a letter carrier, where the channel is the person carrying the letters. The letter carrier could deliver one letter at a time, rushing back and forth between the post office and the house (clearly an example of poor channel utilization and throughput). On the other hand, as an example of efficient channel utilization, the letter carrier could deliver all the letters to the house once, thereby minimizing the number of trips required and hence maximizing throughput.

Throughput can be maximized in a number of ways. For example, in a frame, the ratio of control information to message should be as low as possible (that is, the number of bytes of control information should be much less that the number of bytes of message), otherwise the channel is carrying more control information than useful information. Throughput can also be enhanced by increasing the speed of the channel (that is, the number of bits (or bytes) that are sent in a given period of time).

A third technique involves *how* the information in a frame is sent between the communicating entities. Information is typically sent between computers as a stream of bytes (for example, frames are constructed out of a series of bytes). The channel between the two computers can either be **serial**, which means that one bit at-a-time is sent, or **parallel**, meaning that all eight bits making up the byte are sent at once; this is shown in Figure 1.12.

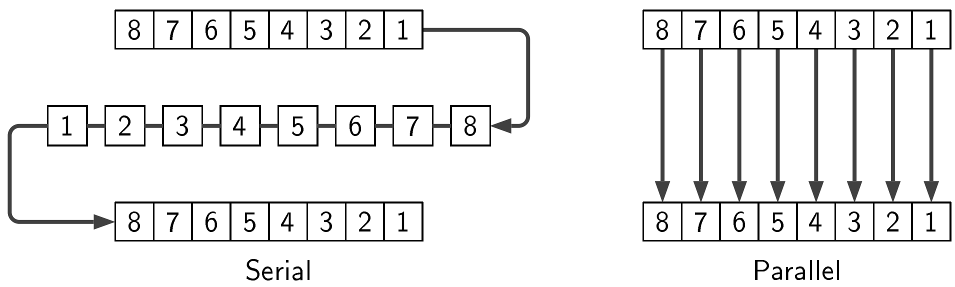


Figure 1.12: Serial vs. parallel communications

The obvious advantage of sending bytes in parallel is that it is eight times quicker than sending them serially. The trade-off is the cost of setting up a communication channel consisting of one path (for serial) or eight paths (for parallel): eight times as many connections and paths are need for parallel. One usually finds parallel connections *within* the computer or between closely coupled computers; however, over distances of more than a few metres, most communications are serial.

Within the computer, bytes are moved in parallel. Before a byte can be transmitted serially, it must be converted from parallel to serial using a hardware **parallel-to-serial converter**. Similarly, the stream of serial bits must be assembled into a (parallel) byte using a **serial-to-parallel converter**.

1.3.5 Synchronization and Timing

In any communication, certain steps must occur before others. For example, in a telephone conversation, before the conversation can take place, one person must dial the telephone number of the other person, and before the person being called can answer the telephone, the telephone must ring. Once the called person answers the telephone, the conversation can proceed.

Protocols often require the entities involved in a communication to be in-step or **synchronized**. That is, one event occurs which is followed by another. Continuing with the telephone analogy, once the communication has been established, synchronization is maintained, typically with one person making a series of statements, to which the other responds. At the end of the telephone conversation, both parties agree that the conversation is over and hang-up.

Synchronization in a computer can occur at any number of levels. Two communicating processes must exchange synchronization information (such as sequence numbers, as discussed in section 1.3.3), to ensure that no messages are lost. Synchronization is achieved by the use of an external timing device, typically a clock.

At lower layers, synchronization must be maintained by the communicating computers at the frame, byte, and bit levels. Since the computers are operating independently, the receiver must 'know' when the next bit is to arrive. This can be achieved by having each computer maintain a clock that is *in sync* with the other computer's clock. For example, at each 'tick' of its clock, the transmitter sends a bit; while at each 'tick' of the receiver's clock, a bit is read from the channel.

The term **synchronous communication** is applied to those communication systems that can send a stream of bytes with only periodic timing information (typically attached to the start of the frame); while the term **asynchronous communication** is applied to those communication systems that require each transmitted byte to be prefixed with timing information.

Communicating entities can also use timing to ensure that the communication is still in progress. For example, if an entity does not receive a response to a message within a given time period, the entity can take action to determine the cause of the inactivity. The inactivity could be due to a variety of reasons, including a break in the communication channel (giving the false impression that the other computer is inactive), or the other computer may have crashed (thereby halting any communication activities on the channel). Many protocols require special frames to be sent periodically, to allow each entity involved in the communication to determine that the other entities are still active.

Examples of synchronization and timing will appear throughout the remainder of the book.

1.3.6 Transparency

In any communication, the underlying facilities supporting the communication should be **transparent** to the entities involved in the communication. For example, when using the telephone, the people involved in the communication should not be concerned with the details of how the telephone system actually places the call.

The same arguments can be applied to entities within a computer communication system, for similar reasons:

- application programmers should not be concerned with the 'hows' and 'whys' of the underlying communication system, as long as the facilities exist to get information from one machine to another. For example, the channel could be serial (or parallel), or the bytes could be sent synchronously or asynchronously: these issues should be transparent to both the programmer and the application.
- if an application is written for a specific communication system, it can become very difficult to transfer the software to another computer using a different communication system. Therefore, by layering the software and making the underlying layers transparent to the application software, it should be possible to transfer the software to another computer as long as the **interface** between the application and the lower layers is the same for both systems.

The term transparency can also be applied to the type of information that is transmitted. A **transparent** frame is one that can contain all possible characters (both control and non-control characters) as part of its message; while a **non-transparent** frame is one that can only contain non-control characters (i.e., printable characters only).

1.4 Data Terminal Equipment

In its simplest form, a computer can be considered to be a device, which, when given **input**, **processes** it, and produces **output**. If a computer is to communicate with other computers, it also requires some type of **connection**. A computer that serves as a point of collection, use, or transfer of data, is referred to as **Data Terminal Equipment**, or, more commonly, a **DTE**. A DTE is represented diagrammatically in Figure 1.13. Examples of DTEs include terminals, point-of-sale terminals, line printers, and computers.

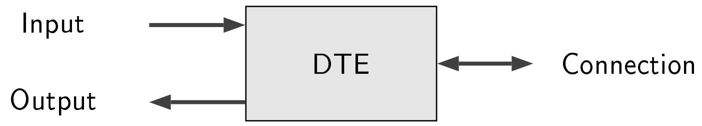


Figure 1.13: A DTE (Data Terminal Equipment)

1.4.1 Input

To the vast majority of people who use computers, input takes place through the keyboard. Most keyboards are equipped with more than the standard typewriter '`QWERTY`' keys; for example, 'function' and cursor control keys are also available.

Over the past decade, a variety of specialized input tools have been developed, including optical scanners (to interpret the 'bar-codes' associated with items such as supermarket goods), voice-to-digital units (to allow people to talk to the computer rather than type), magnetic strip decoders (to read the magnetic material found on the back of most charge-cards), and pointing tools such as the mouse and track-ball (to allow a user to select an object on the screen).

Although most DTEs support either ASCII or EBCDIC internally, input to the DTE need not be either. The PC keyboard is an example of an input device which produces its own character code which must be converted to another character code (typically ASCII), before being processed. Similarly, a voice input device may produce 8-bit data values, but an individual byte probably has no direct equivalent in either ASCII or EBCDIC.

1.4.2 Output

For the most part, DTE output is visual; originally textual either on hardcopy (paper) or softcopy (cathode ray tube or CRT). Today, output can be a bit-mapped image including **icons** (still-pictures on the screen), **facsimile** (fax documents), and animation. Monochrome CRTs are being superseded by colour and liquid crystal display (LCD) terminals.

It is worth noting that a DTE need not be associated with input. For example, a line printer (a DTE) produces output and is connected to a computer; however, the line printer need not support input.

Similarly, not all output need be printable text. Consider the following examples:

- both ASCII and EBCDIC support control characters which can be used to control the output text stream. For example, certain escape sequences are intended as control signals to the DTE rather than for the user.
- some DTEs support voice output for tasks that require people to listen rather than read. For example, voice output is necessary for people with sight disabilities.
- output can be an *action* such as unlocking a door after a person has typed the password associated with the lock. That is, the DTE's output is used to control an external device.

1.4.3 Connections

By definition, a DTE must have at least one connection to another computer (a DTE), otherwise communication cannot be achieved. Connections are typically physical, consisting of insulated wires or glass (optical fibre cable). In longer distance communications, the DTEs need not be connected physically, for example, information can be passed through the air (microwave communications), or through space (satellite communications).

It is worth noting that although some DTEs do not support input and output directly, input and output can occur across the connection. For example, some highly specialized computers do not communicate with humans directly. Instead, all input and output is passed to the specialized computer from an external processor: humans communicate with the specialized machine via the external processor. In this situation, the specialized computer does support input and output, but only through the connection.

1.4.4 More DTE Terminology

Probably the most common type of DTE interaction that users experience involves a terminal connected to a remote host (both the terminal and the host are examples of DTEs). Terminals usually operate in one of two modes:

Block mode. A block mode terminal is one in which the information entered at the terminal by the user is not sent to the remote DTE until the terminal receives an explicit signal from the user. Examples of block mode terminals include airline reservation systems in which information is accepted from a client and verified before being sent to the remote host for processing. Block mode terminals are typically CRTs.

Character mode. Character mode terminals accept single characters entered by the user and forward them to the remote host. In some situations, the character mode terminal will buffer one line of characters before sending it to the remote host.

The characters entered by a user can be displayed from either the local *or* the remote DTE:

- A **remote echo** occurs when a character is entered at the user's local DTE and the remote DTE is responsible for outputting the character. Figure 1.14 illustrates the cycle of a remote echo: the local DTE accepts the character and transmits the character to the remote DTE; the remote DTE performs whatever processing is required on the character and returns the character to the local DTE for output.

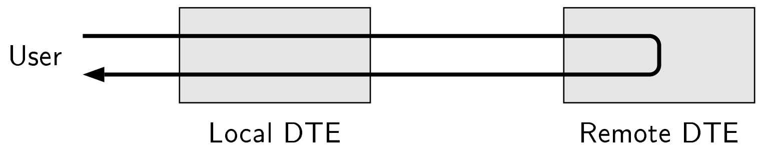


Figure 1.14: Remote echoing

Remote echoing is often used in applications, such as screen editors, that do not expect the user to end each command with a delimiter such as a carriage return. Remote echoing is sometimes referred to as **echoplexing**.

- A **local echo** occurs when the user's local DTE is responsible for echoing each character entered by the user. The local DTE is still responsible for forwarding each character to the remote DTE (data sent from the remote DTE is displayed on the user's local DTE), as shown in Figure 1.15.

  Local echo is often used by block mode terminals; when the user types ENTER, the information is transmitted to the remote DTE.

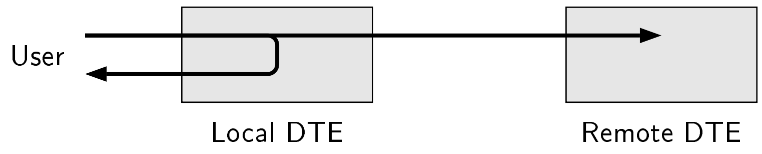


Figure 1.15: Local echoing (Local DTE echos each character before transmission)

1.5 Summary

This Chapter has introduced some of the terminology and concepts associated with data communications. There are a number of points that should be remembered when dealing with data communications:

- almost all aspects of data communications have a 'human parallel'; when stuck trying to unravel a problem in data communications, think of it in 'human' terms.
- the only reason society functions is through rules: the same is true with data communications. Protocol standards exist to be adhered to, otherwise there is no guarantee that the implementation or any applications that use it will succeed.

1.6 Exercises

1. Describe the medium and suggest the type of channel (either simplex or duplex) for the following communications:

   (a) A person talking.

   (b) A pigeon carrying a message.

   (c) Television.

2. Which of the following communications could be described as half-duplex? As full-duplex?

   (a) Two polite people talking.

   (b) Two angry people yelling.

   (c) Sending a message by smoke signals.

   Of those that are half-duplex, what is the line turnaround signal (if any)?

3. Assuming that most, if not all, human speech falls into the 'half-duplex' category, identify some gestures and phrases that people use to indicate that they have finished talking (i.e., the point at which the line can be turned around).

4. Describe the steps required to convert a byte stored in parallel to serial. Consider the hardware required for such an operation.

5. Repeat the previous question, but convert the byte from serial to parallel.

6. Experiment with several computers to determine if they support X-OFF (CTRL-S) and X-ON (CTRL-Q). What happens if CTRL-S is typed followed by a command, and finally CTRL-Q?

7. Examine several computers to determine their character codes (i.e., ASCII or EBCDIC). Try implementing the following C program to display the computer's different characters:

```
#include <stdio.h>
main()
{
unsigned char ch;
for(ch = 0; ch < 256; ch++)
     printf("%d %c\n", ch, ch);
}
```

   Now, explain why the above program runs forever, printing the characters and their numeric equivalents.

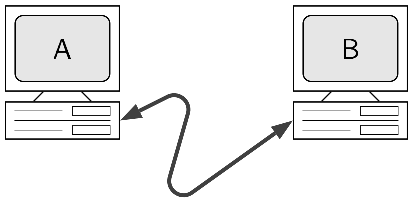
8. Pictographic languages such as Chinese do not work well with ASCII. Suggest a means whereby Chinese symbols (each representing a 'word') could be supported by ASCII.

9. A DTE displays every character typed by a user *twice*. Explain what is happening and how to correct it.

10. A DTE doesn't display anything that is typed, but the remote DTE responds when a carriage return is pressed. Explain what is happening and how to correct it.

Part 2

Computer communications over an unshared channel



Most computer communications occur between pairs of DTEs. For example, one widespread application of computer communications involves the transfer of information from an individual's terminal to a remote computer. This type of communication is commonly known as a **point-to-point** communication.

The communication channel over which the communication takes place can be **shared** by other communicating entities or it can be **unshared**, dedicated to the two computers (i.e., one at either end of the channel). Part 2 examines two approaches to unshared point-to-point communications: **asynchronous** and **synchronous**.

As shown in Chapter 1, all communications are based upon time. Although the term 'asynchronous' communication implies that the communication is done without regard to time, this is not the case. As shown in Chapter 2, 'asynchronous' refers to the fact that each byte is transmitted without regard to the time the previous byte was sent.

In synchronous communications, bytes are sent in blocks or messages, with no gaps between each byte sent, resulting in both benefits and problems that must be addressed. Synchronous communications are discussed in Chapter 3.

Chapter 2

Asynchronous Serial Communications

2.1 Introduction

An asynchronous communication is one that can occur without regard to time. Asynchronous communications are typified by terminal-host communications in which the user can send data from the terminal to the host periodically at random, unpredictable intervals. Due to the widespread popularity and availability of asynchronous communications (for example, most PCs are equipped with at least one asynchronous communications adapter), this Chapter examines asynchronous communications in detail.

2.2 General Concepts

In Chapter 1, it was shown that data is stored in **binary** form. For example, the ASCII letter 'S' is represented by the 7-bit binary pattern:

```
101 0011
```

When information is transmitted on a communication channel between two devices, it is not sent as a letter (such as 'S'), but rather in a form that can be easily manipulated by the computers. Since the information is already stored in binary within the device, such as a computer, data is, not surprisingly, transmitted as a series of `0`s and `1`s. For example, the transmission of the letter 'S' is represented pictorially in Figure 2.1.

When data is transferred between devices, it is usually done serially, since it is cheaper to use one wire instead of seven or eight (depending upon the number of bits in the byte) that would be required in a parallel transmission. The rightmost (low-order) bit is sent first, followed by the remaining bits, working from right-to-left.

It is generally accepted that a transmitted '1' is called a **mark**, while a transmitted '0' is called a **space**. These terms come from the days of telegraphy when a '1' on the communication channel was

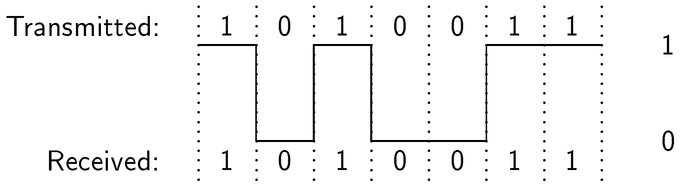


Figure 2.1: Transmission of the letter 'S'

represented by a flow of current which resulted in the equipment making a 'mark' on the receiver's output device (a moving strip of paper), while the lack of current left a 'space' on the paper.

The receiving device must read the data bits from the communication channel at the same rate at which they are being transmitted, otherwise information will be lost. For example, Figure 2.2 depicts what might result if the byte 'S' is received at twice the rate of transmission; the receiver receives *two* bytes instead of one.

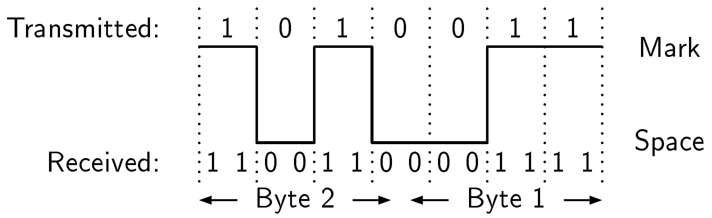


Figure 2.2: Receiving at twice the rate of transmission

Similarly, if the transmitter is transmitting faster than the receiver can accept, there may also be a loss of information. In Figure 2.3, the transmitter is sending data at twice the rate it is being received (the letter 'S' is sent once again). The receiver receives only four bits rather than the seven that make up the byte. The value of the third bit could be either a mark, 1, or a space, 0, depending upon the exact moment the channel is sampled (channel sampling is discussed further in section 2.5, below).

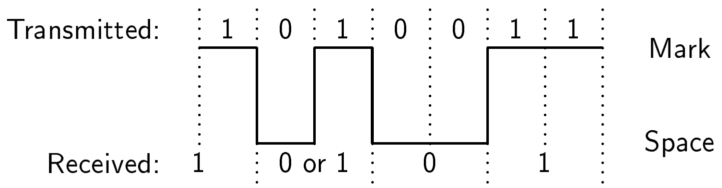


Figure 2.3: Receiving at half the rate of transmission

As the previous examples illustrate, in any communication, both the transmitting and receiving devices must agree to a single, common channel speed known as the **line speed**.

Asynchronous line speeds can be expressed in terms of **bits-per-second** (or **bps**) and typically range from 50 bps to 9600 bps. Some commonly used line speeds are 1200 bps, 2400 bps, and 9600 bps. As well as agreeing upon the line speed, both devicess must agree upon the number of bits in a byte (this is usually a choice between 7-bit and 8-bit, as these are the most common character code widths).

Since the channel can be in only one of two states, mark or space, when there is nothing to transmit the channel must always be left in one of these states. All devices, when they are not transmitting, leave the channel in the mark state.

This then raises another issue: if the channel idles in the mark state, how does the receiving device determine where the first data bit occurs? In the worst case, if a device transmits the binary pattern '`1111111`', the receiving device will not be able to distinguish between the marks representing the idle channel and the marks representing the data!

The solution to this problem is to have the first bit sent as a 'non-data' space. That is, before the first data bit is sent, the channel changes state from mark to space. After the space, the bits making up the byte are transmitted; Figure 2.4 shows the transmission of the byte 'S'.

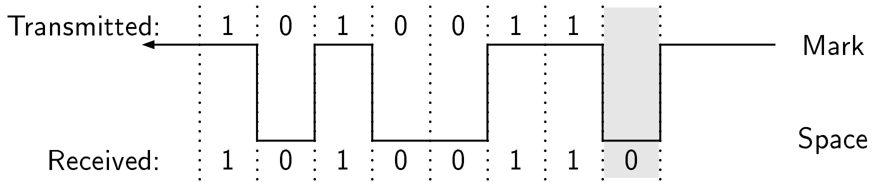


Figure 2.4: Start of transmission is denoted by the initial mark-to-space transition

The first bit, indicating the start of data, is called the **start bit**. The start bit signals the receiving device that a data byte is about to follow. The width of the start bit is the same as each of the data bits (that is, it takes the same length of time to send the start bit as it does to sent a data bit). Upon receipt of the start bit, the two devices are **synchronized** for the length of time it takes to send the data byte.

Once a data byte is received, the receiving device must process it. Processing the character is not an instantaneous operation; at a minimum, the receiving device must convert the incoming serial data byte into a parallel byte for storing (this is known as a **serial-to-parallel** conversion). If the transmitting device continues to transmit while the serial-to-parallel conversion is taking place, one or more bytes may be lost unless the receiving device is given time to process each byte. To give the receiving device the necessary processing time, one or more **stop bits** are appended to the end of each data byte.

The stop bit(s) are simply the length of time the transmitting device must wait before the next byte can be transmitted. The three most common stop bit times are: 1 bit, $1\frac{1}{2}$ bits, and 2 bits. Figure 2.5 shows the transmission of the byte 'S' with a start bit, and two stop bits.

The start and stop bits are said to **frame** the transmitted data byte.

Errors

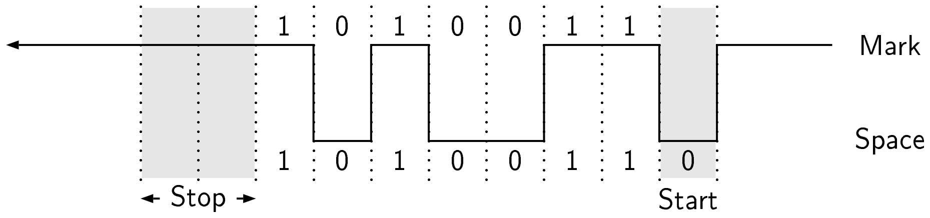


Figure 2.5: The byte 'S' sent with a start bit and two stop bits

On occasion, one or more of the bits in the transmission may be changed from a 1 to a 0 (or vice versa) due to noise or other interference on the channel. In Figure 2.6, the byte 'S' is changed into the byte 'Q' because of noise on the line (the second bit is inverted from a 1 to a 0).

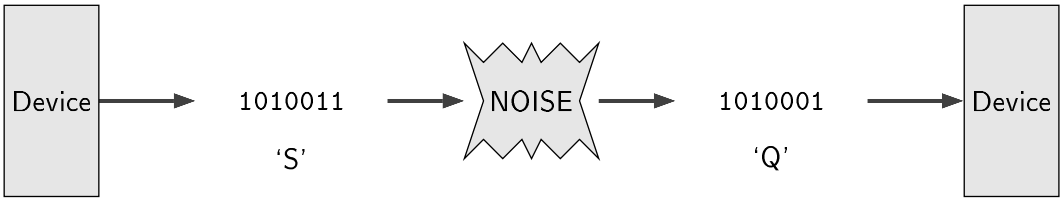


Figure 2.6: Bit inversion due to noise

Even with the start and stop framing bits, the receiving device has no way of determining from the information received that the byte is correct. What is needed is a mechanism whereby the receiving device can detect that an error has occurred in the transmission of the information.

One possible error detection technique is to transmit each byte *twice* and to have the receiver examine both bytes. If the bytes are different, the receiver has detected an error (but cannot determine which byte is the correct one). Interestingly enough, if both bytes are the same, the receiver still has no indication of whether the bytes are correct, since the same error might have occurred to both bytes, resulting in the same bit pattern.

Retransmitting the entire byte presents another problem: using the channel to send each byte twice means that only half as much information is being transmitted. A much less costly form of **error detection** can be achieved by summing the bits and determining whether the sum is **odd** or **even**. The result of the sum (i.e., odd or even), can be represented as a single bit and transmitted within the frame. This extra bit is known as the **parity** bit and is transmitted after the last data bit and before the stop bit(s); in Figure 2.7, the byte 'S' is transmitted with a start bit, a parity bit, and two stop bits.

The two best known types of parity are:

Odd parity: the sum of all the bits, including the parity bit, is **odd**. As an example, in Figure 2.8, the byte 'S' (bit pattern 1010011) is transmitted and the parity bit is set; that is, it has a value of 1, since the sum of the data bits is even (there are *four* data bits set to 1), but the total of all the bits must be odd.

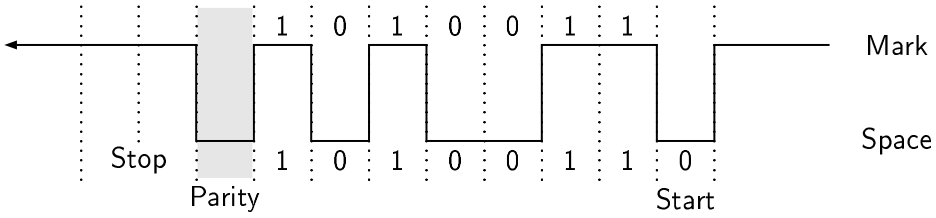


Figure 2.7: The parity bit follows the last data bit and precedes the stop bit(s)

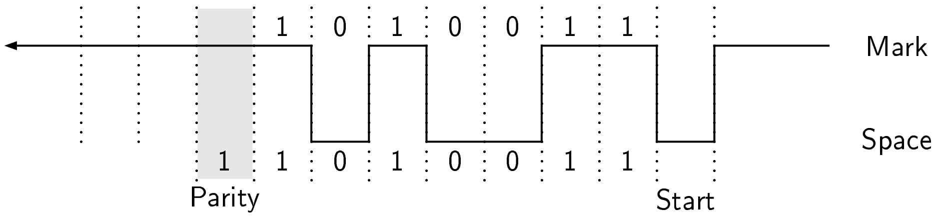


Figure 2.8: Parity bit is set since the sum of the data bits is not odd

However, as shown in Figure 2.9, if the byte 'L' (bit pattern `1001100`) is transmitted, the parity bit is cleared; that is, it has a value of `0`, since the sum of the data bits is already odd (there are *three* data bits set to `1`).

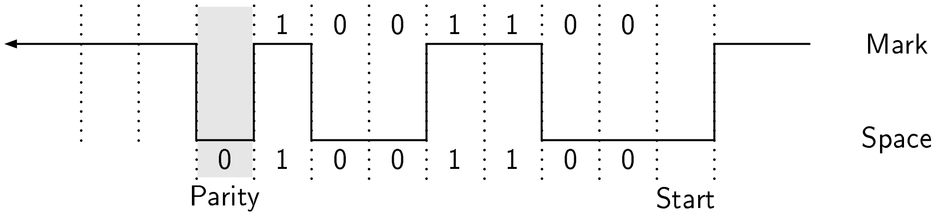


Figure 2.9: Parity bit is cleared since the sum of the data bits is already odd

Even parity: the sum of all the bits, including the parity bit is **even**. For example, in Figure 2.10, the byte 'S' is transmitted, requiring the parity bit to have a value of `0`, since the sum of the data bits is already even.

If the byte 'L' (bit pattern `1001100`) is transmitted, the parity bit would have a value of `1`, since the sum of the data bits is odd, but, as shown in Figure 2.11, the total of all bits must be even.

The receiving device can detect an error when the sum of the bits (data plus parity) is wrong (i.e., odd instead of even or even instead of odd). This type of error is known as a **parity error**. Note once again that the error has been detected but it has not been corrected. If the error is to be

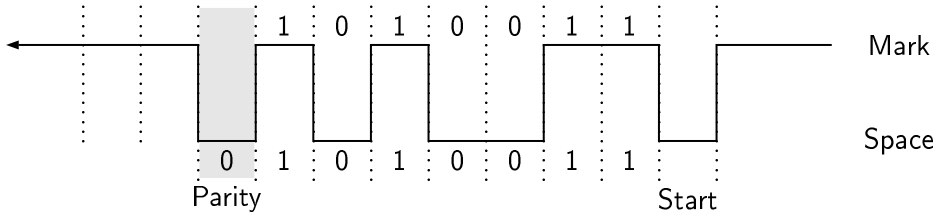


Figure 2.10: Parity bit is cleared since the sum of the data bits is already even

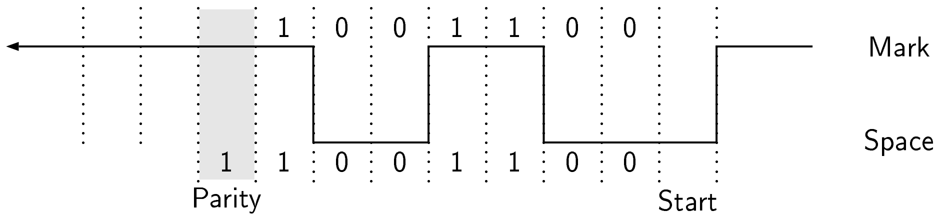


Figure 2.11: Parity bit is set since the sum of the data bits is not even

corrected, the receiving device must inform the transmitting device that the last byte received was in error.

If the receiving device is to detect the parity error, it must support the same parity the transmitting device is using (i.e., both must be odd or both must be even). Failure to observe this will result in all bytes being flagged as having parity errors. As an example, consider the transmission of the byte 'S' with even parity, where there are a total of four bits set to 1 (the parity bit is therefore zero). If the transmitted byte is received by a device expecting odd parity, the byte will be received in error, because an even number of bits is set.

Now, if interference occurs on the connection between the two devices, the receiving device can *detect* the error (however, the error cannot be *corrected* since it is not possible to determine which bit(s) are in error). In Figure 2.12, the byte 'S' is transmitted with odd parity, but noise changes the second bit from a 1 to a 0.

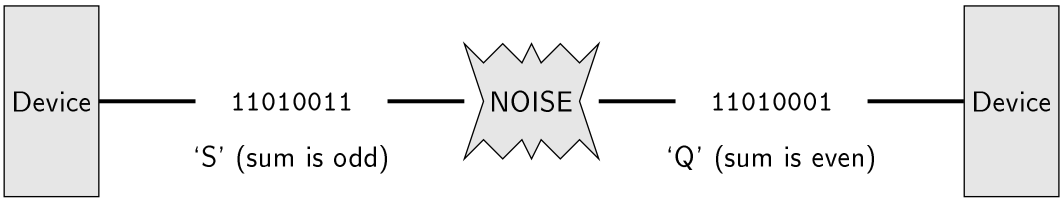


Figure 2.12: Error detection using the parity bit

Upon receipt of the byte 'Q', the receiver can immediately determine that the data was not received correctly because the parity is wrong (the sum of the bits is *even* when it should be *odd*).

Does parity error detection still work if the parity bit is changed? Yes, since parity error detection works by summing all the bits, data *and* parity. For example, in Figure 2.13, although the byte 'S' that is received is valid, the parity is incorrect, indicating that an error has occurred somewhere in the frame. Since there is no indication as to where the error has occurred, the entire frame is taken to be in error.

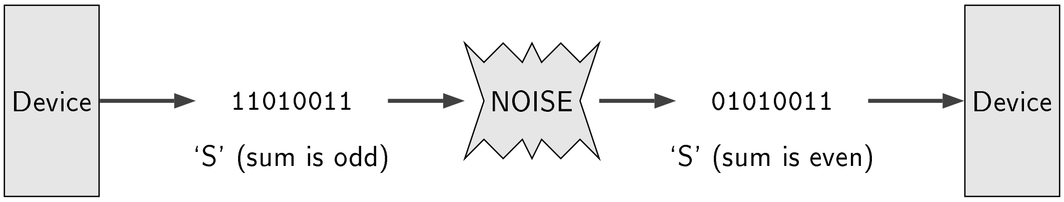


Figure 2.13: Error detection using parity bit (parity bit inverted)

Parity error detection does not work in all cases, as illustrated in Figure 2.14, in which the byte 'S' is affected by noise which inverts bits 2 and 3, producing the byte 'U' (bit pattern `1010101`); note that odd parity is being used. The sum of the bits is *still* odd, which means that the error goes undetected since the parity is correct. Parity checking (odd or even) is not foolproof: it can only detect *odd* numbers of errors.

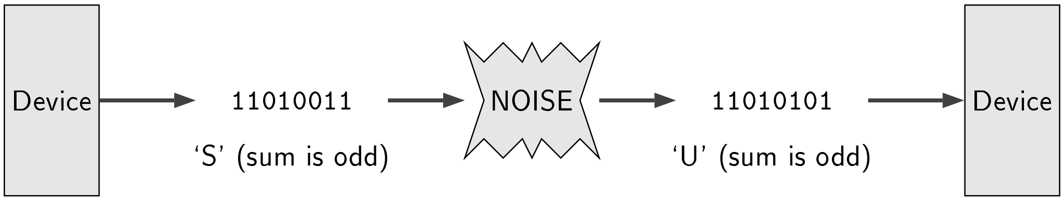


Figure 2.14: An example of parity failing to detect an error

In addition to odd and even parity, many asynchronous systems also support the following parities (in all cases, the start and stop bits are still sent):

None: the parity bit is neither generated nor transmitted. This reduces the number of bits in the frame, which can be useful when transferring large amounts of data. However, it also means that the receiver cannot detect if an error has occurred in the transmission of the data.

Mark parity: the parity bit, regardless of the data bits, always has a value of `1` (hence the name, mark parity). Figure 2.15, illustrates the transmission of the byte 'S' with mark parity.

If the parity bit received is *not* a mark, then a parity error is detected; otherwise, the byte is assumed to be error-free.

Space parity: the parity bit, regardless of the data bits, always has a value of `0` (hence the name, space parity). The transmission of the byte 'L' with space parity is illustrated in Figure 2.16.

If the parity bit received is *not* a space, then a parity error is detected; otherwise, the byte is assumed to be error-free.

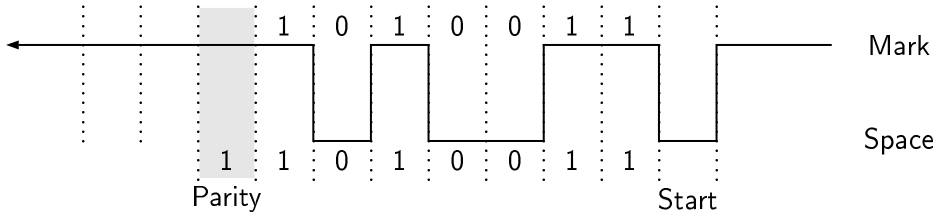


Figure 2.15: In mark parity, the parity bit is always set

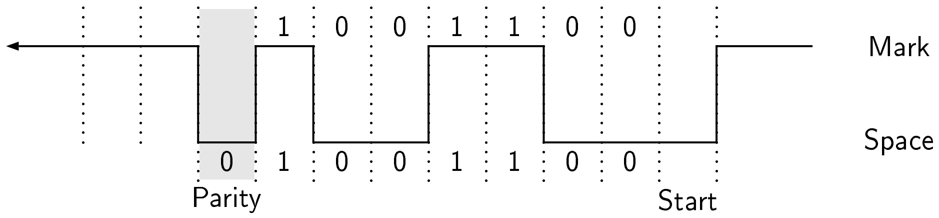


Figure 2.16: In space parity, the parity bit is always cleared

2.3 Other Errors

There are two other common types of errors that can occur on an asynchronous channel:

- An **overrun error** occurs when the receiving device does not process the incoming data bytes quickly enough. In Figure 2.17, three bytes have been sent by the transmitting device; however, before the receiving device has had time to receive and process the second byte (it is still processing the first byte), the third byte arrives, overwriting the second byte.

Transmitting Device		Receiving Device
First byte sent	$\longrightarrow$	Receives first byte
Second byte sent	$\longrightarrow$	Processing first byte
Third byte sent	$\longrightarrow$	Receives third byte

Figure 2.17: An example of an overrun error

  Overruns can be avoided by reducing the processing load on the receiving device or reducing the line speed.

- A **framing error** occurs when the receiving device cannot detect the end of the frame (i.e., the stop bits cannot be detected). There are two reasons why a device can miss the stop bit(s):

  – noise may have caused the stop bits to be inverted, prompting the receiving device to detect one or more spaces instead of marks.

  – noise may have caused the start bit to be inverted, making the device start assembling the byte with the wrong bit.

Regardless of the cause of the framing error, there is no mechanism available to allow the receiving device to determine the correct value of the transmitted byte (other than the re-transmission of the byte).

Further explanation of both overrun and framing errors is given in section 2.5, below.

2.4 The Break Condition

In addition to the characters found on a terminal's keyboard, many devices with keyboards have a key marked BREAK. The break key does not have an equivalent ASCII (or EBCDIC) character code; rather, it is intended to cause the user's device to generate a **break condition** on the channel. A break can be used for a number of reasons; in many applications it is intended to signal the receiving device that attention is required.

As shown in Figure 2.18, a break condition is indicated by holding the channel in the space state for *at least* the amount of time required to send an entire frame (i.e., a start bit, the data bits, the parity bit, and the stop bits).

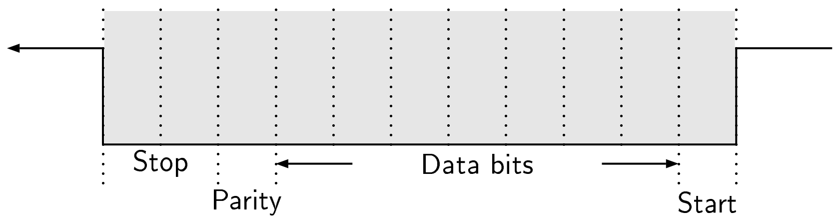


Figure 2.18: In a break condition, all the frame bits are cleared

How the receiving device handles the break condition is determined by a higher level of protocol. For example, software may be designed to treat the reception of a break as an indication that a process being executed is to be aborted.

2.5 Asynchronous Communications Hardware

Data within a device is usually stored and moved in a parallel fashion (the start and stop framing bits do not exist); however, the data sent across the channel is generally sent serially. The device is therefore required to convert the internal data to a form that can be transmitted on the channel (in serial form, with start, stop, and parity bits). Similarly, the serial data received from the channel must be changed into parallel for use within the device.

The conversion of the data for transmission and reception is *often* done in hardware by a device known as a Universal Asynchronous Receiver/Transmitter or **UART**. Figure 2.19, represents the major UART connections.

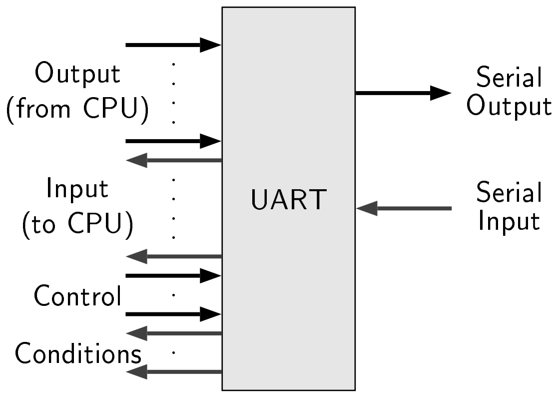


Figure 2.19: The basic UART connections

The UART is responsible for generating the start, stop, and parity bits, as well as for removing them. Channel conditions such as the break condition, framing and overrun errors are signalled by the UART back to the processor. The processor can send control signals to the UART, including line speed, word size, parity, and the number of stop bits.

UART Transmission

When transmitting data, the UART must convert the internal parallel byte into a stream of serial bits using **parallel-to-serial** conversion. The byte to be transmitted is supplied to the UART by the device's processor. The bits are stored in a **shift register** (often referred to as the **Transmit Shift Register**). The start bit is sent first, followed by the bits in the shift register. As shown in Figure 2.20, the shift register **clocks** them onto the channel, one-at-a-time, at the line speed.

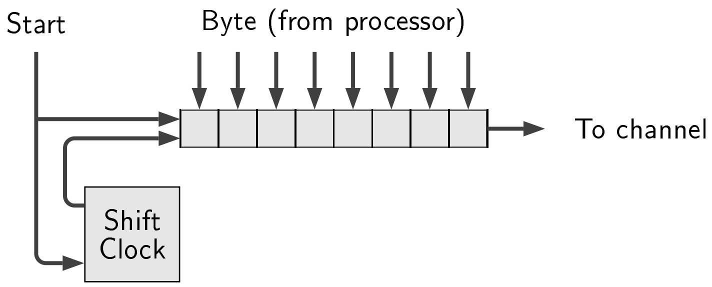


Figure 2.20: The UART Shift Register

Odd and even parity are calculated by exclusive or'ing the data bits as they are shifted onto the channel. The parity bit is initialized to `0` for even parity and `1` for odd parity.

Each bit that is sent is included in the parity generation. Once the final data bit is included in the parity, the parity bit can be clocked onto the channel for transmission. If mark or space parity is selected, a `1` or `0`, respectively, is put onto the channel after the final bit, regardless of the value in the parity generation hardware.

The parity bit is followed by one or more stop bits. Stop bits are 'generated' by the UART entering an idle state in which the channel idles with marks and blocks transmission for at least the time it would take to transmit the required number of stop bits.

It is important to note that the line speed indicates the *total* number of bits the UART can transmit in one second. Therefore, when determining the number of bytes that can be transmitted in one second, it is necessary to include the number of data bits as well as the number of start, stop, and parity bits. For example, a 1200 bps line transmitting 7-bit bytes, with parity (odd or even), one start and one stop bit (a total of 10 bits) could transmit a maximum of 120 bytes per second.

Once the UART has sent a byte, the processor can be signalled that the UART is ready to transmit another byte.

If data is supplied by the processor to the UART only when the shift register is empty, noticeable transmission delays may result. This is because transferring a byte to the UART is not an instantaneous operation since the line speed is considerably less than the speed of the processor and the UART.

The periods when the channel is idle can be reduced if a second register, a **holding register** (usually referred to as the **Transmit Holding Register**), is placed between the processor and the UART. The relationship between the two registers is shown in Figure 2.21.

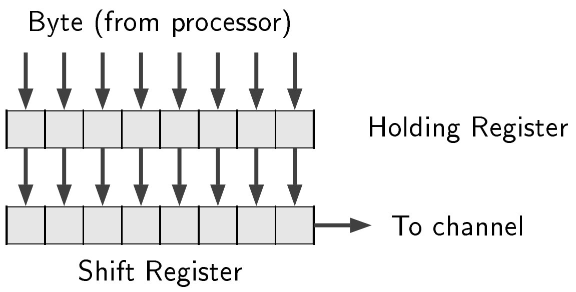


Figure 2.21: UART Transmit Holding Register

Delays are now reduced because the UART can be transmitting a byte from the shift register while the processor fills the holding register. This technique, whereby two registers are used in a transmission is known as **double buffering**. In addition to accepting bytes from the processor, most UARTs can signal the processor when the Transmit Holding Register is empty and when the Transmit Shift Register is empty.

UART Reception

The UART is in an idle receive state while the channel is in an idle (i.e., mark) state. Upon detection of a channel state change (i.e., the channel changes from a mark to a space), the UART determines the center of the start bit and then reads bits from the channel at regular intervals using a **sampling clock**. The bits are assembled in a shift register; each time the sampling clock 'ticks', the register is shifted to the right and the next bit from the channel is read. When all of the bits have been read from the channel (the number of bits read is determined by the word size), the resulting byte is supplied to the processor; see Figure 2.22.

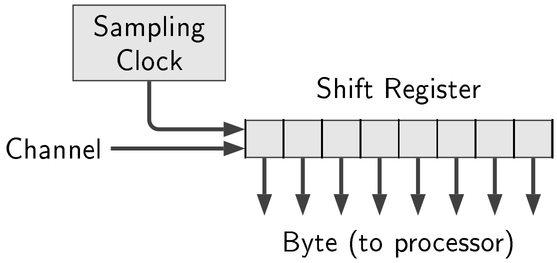


Figure 2.22: UART Reception Shift Register

The UART also scans for the start and stop bits, and if required, calculates and performs a parity check. How parity is handled depends upon the parity chosen:

None. The UART ignores parity checking altogether.

Mark or **Space.** The channel can be sampled after the last data bit is read. The value of the bit can be compared to that expected by the receiving UART. If the parity bit received is the same as that expected, the data is assumed to be correct, otherwise a parity error is signalled.

Odd or **Even.** Both the incoming data bits *and* the parity bit are used to determine whether a parity error has occurred. The UART initializes its parity check circuit to `0` (even parity) or `1` (odd parity) and proceeds to generate parity by exclusive or'ing the incoming data bits with the parity. If no errors are detected, the value of the parity calculated by the UART should be the same as that received by the UART. At this point, the UART can explicitly compare the parity bit received with that calculated and signal the DTE accordingly.

Alternatively, the UART can include the incoming parity bit in the parity calculation. If the parity in the frame received is the same as that calculated by the UART (i.e., both `0` or both `1`), the result of this final exclusive or'ing is `0`. However, if the values are different, the final exclusive or'ing will result in a value of `1`. This result can be made available to the DTE; for example, a zero could indicate that no errors were detected, while a one would signal a parity error.

Once a complete byte has been assembled in the Shift Register, the UART can signal the processor that a byte is available. At this point, the processor can copy the byte from the Shift Register into a memory location.

If the assembled byte is kept in the Shift Register and the processor does not read the Shift Register before the first bit of the next byte arrives, an **overrun** error will occur. Since there is only one Shift Register, either the incoming byte must be discarded or the byte in the Shift Register must be overwritten. Ideally, no bytes should be lost (neither the incoming byte nor the byte in the Shift Register), meaning that the byte in the Shift Register should be read by the processor as quickly as possible. The reception technique is similar to that used in transmission: a second register is placed between the processor and the shift register (i.e., double buffering). As soon as a byte is assembled in the shift register, it is copied (in parallel) into the holding register; see Figure 2.23.

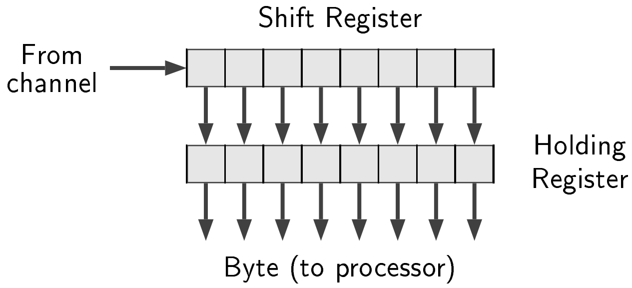


Figure 2.23: UART Reception Holding Register

The addition of the holding register does not eliminate the possibility of an overrun occurring, it merely reduces the probability. For example, a byte could be copied into the holding register and the processor signalled; if the processor did not remove the byte from the holding register before a second byte was copied into it, an overrun error would still occur.

The other errors discussed in section 2.2 are also handled by the UART's reception hardware. If the parity calculated as the byte is received does not agree with the parity supplied with the byte, the UART should signal the processor that a **parity** error has occurred. If a stop bit (i.e., a mark) is not detected after the last bit (data or parity, depending on whether parity is required), the UART should signal the processor that a **framing** error has occurred.

The UART should also be able to detect a **break condition**. Specifically, after the line has been in the space state for at least the start bit, data bits, parity bit, and stop bits. Often, a UART will signal a framing error when detecting a break because the framing error condition can be indistinguishable from the break condition. For example, an inverted (missing) stop bit on a null character sent with even parity is equivalent to a break condition.

The value of the byte assembled by the UART clearly depends upon the bits that are sampled from the channel. Although the signal diagrams have shown the rising and falling edges of a transmitted character to be sharp and square, in reality the edges may be much less distinct. In fact, over time the signal may become **attenuated** or weakened, so that it is extremely difficult to determine the value of the bit, as shown in Figure 2.24.

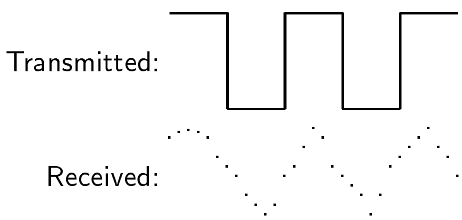


Figure 2.24: Signal attenuation

Since the value of a bit may become harder to determine at its edges, UARTs are normally designed to sample the line as close to the *centre* of each bit as possible.

The centre of each bit is found by first locating the center of the start bit. Once the center of the start bit has been found, the UART can sample the line once per **bit time** (i.e., the inverse of the line speed; for example, a 1200 bps line would have a bit time of $\frac{1}{1200}$ of a second), to find the centre of the remaining bits in the transmission.

The algorithm to find the centre of the start bit is as follows:

1. Wait for a mark-to-space transition;
2. When a transition has been found, sample the line at 16 times the bit rate;
3. After 8 'ticks' (i.e., one-half a bit time), sample the line:
    - if a space (0) is found, a start bit is assumed and the line can be sampled once per bit time.
    - if a mark (1) is found, a false start bit was found, and the line sampling can be abandoned.

For example, if the letter 'S' is sent, the sampling of the channel occurs as shown in Figure 2.25.

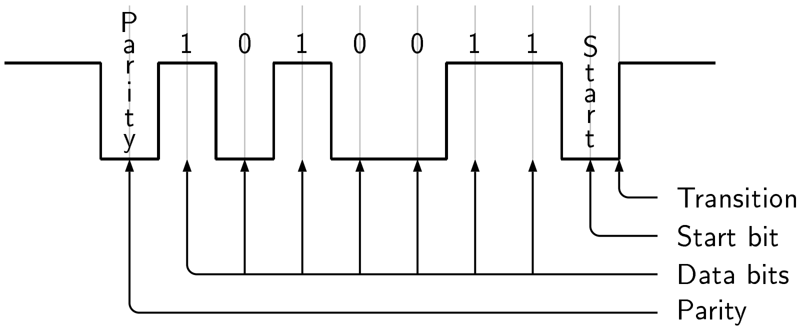


Figure 2.25: The channel is sampled at the centre of each bit

Since the UART may not start its sampling clock at the exact edge of the start bit's mark-to-space transition, using a 16 times counter could prove to be inaccurate in determining the centre of the start bit. The error can be made smaller by sampling at 32 or even 64 times the bit rate.

The underlying assumption in sampling the channel is that both the transmitter and receiver agree to a common bit rate (or line speed) that does not vary over the period of the transmission of the frame. However, should one of the clocks be slower than the other, **clock drift** may occur, possibly resulting in the misinterpretation of the byte or an overrun error. For example, in Figure 2.26, the receiver samples the line at a rate faster than the transmitter sends (the byte 'S' is transmitted with 7-bit, even parity).

Because the receiver is sampling the line at a rate faster then the transmitter sends, the value of the byte received by the DTE will not be the same as was transmitted. Depending upon the degree of signal attenuation, the DTE could receive one of a number of possible byte values, for example (the leftmost bit is the parity bit):

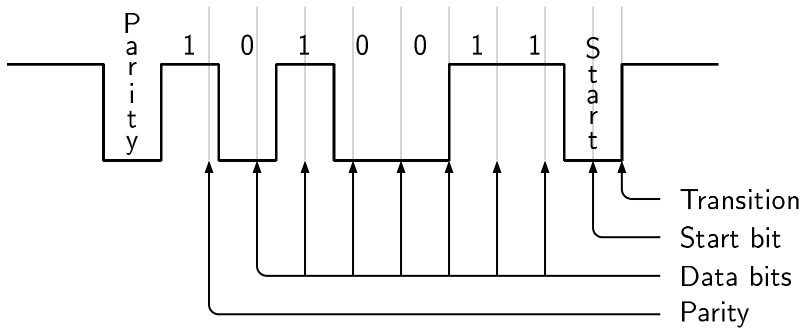


Figure 2.26: Clock drift: the centre of the bit is not found

`10100111`. The third bit received (sampled at the space/mark transition between the second and third bits transmitted) is read as a `1`. A parity error will occur.

`10100011`. The third bit is read as a `0`, meaning that the parity bit and the data bits are assumed to be correct.

In each of the above cases, the parity bit may be misinterpreted as the start bit of the next byte. Regardless of the value of the byte received by the DTE, it is wrong, meaning that one or both UART clocks need to be reinitialized and the data retransmitted.

2.6 RS-232 and Physical Connections

RS-232-C (ITU-T V.24) defines the electrical, physical, and mechanical properties of a widely used (and often abused) EIA standard. Almost all existing DTEs support RS-232-C; for example, the serial communication ports supplied with most PCs usually support RS-232-C.

Physical Characteristics

An RS-232-C connector, shown in Figure 2.27, has 25 **pin**s and is known as a **D-connector** because of its shape (one side is slightly longer than the other, making it look like a 'D').

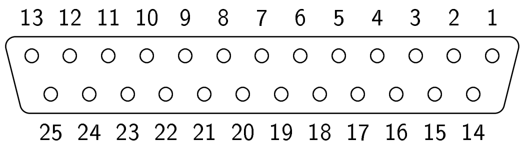


Figure 2.27: The 25-pin RS-232-C (ITU-T V.24) D-connector

Not all 25 pins are necessary to connect two devices together; for example, 9 pin D-connectors are also available. Of the 25 pins, only three are required to support a communication:

Pin 2. Serial data is transmitted on this pin. (On a 9-pin connector, the transmit pin is pin 2.)

Pin 3. Serial data is received on this pin. (On a 9-pin connector, the receive pin is pin 3.)

Pin 7. Used to establish signal ground. (On a 9-pin connector, signal ground is pin 5.)

There are two 'genders' of RS-232-C connector: male (the connector with pins), and female (the connector with holes). The above illustration of a 'D' connector has female numbering (the male is reversed, so that when the connectors are coupled, the numbering corresponds correctly). The RS-232-C standard expects all DTEs to have male connectors, meaning that the cable connecting two DTEs should have female connectors. However, since RS-232-C is a (very) loose standard, it is possible to find DTEs with female connectors. When the cable's connector is the same gender as that of the DTE, a device known as a **gender mender** is required; the gender mender is simply a very short RS-232-C cable with the same gender of connector at either end.

The maximum recommended separation between two DTEs connected by an RS-232-C cable is 50 feet (15 metres); while the maximum recommended bit rate is 9600 bps. As with most aspects of RS-232-C, these maximums are often exceeded; for example, it is possible to run 9600 bps at distances up to 250 feet (75 metres). An RS-232-C cable connects each pin at one end of the cable to the same pin at the other end of the cable (i.e., pin P connects to pin P).

Interconnecting DTEs

Before a pair of DTEs can communicate, they must be connected. In the case of RS-232-C, the standard RS-232-C cable described above will *not* work. The reason for this is fairly straightforward: as shown in Figure 2.28, if a DTE is connected to another DTE, both transmit pins will connect through to one another and both receive pins will do the same (assume a 25-pin connector).

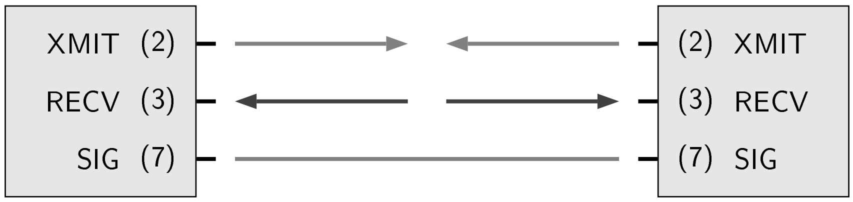


Figure 2.28: The result of directly connecting a pair of DTEs

To allow one DTE to connect to another (without modifying the pin connections on the RS-232-C port), something must be done to the cable to ensure that the communications can take place. The solution is to *swap* the transmit and receive lines, so that whatever is sent on a DTE's pin 2 appears on the other DTE's pin 3. A cable wired in this fashion is known as a **null-modem** cable. The minimum null modem wiring is shown in Figure 2.29.

Signal Levels

The typical voltage range for RS-232-C is ±15 volts, with a positive voltage (in the range +3 volts to +15 volts) indicating that the line is in a space condition and a negative voltage (in the range

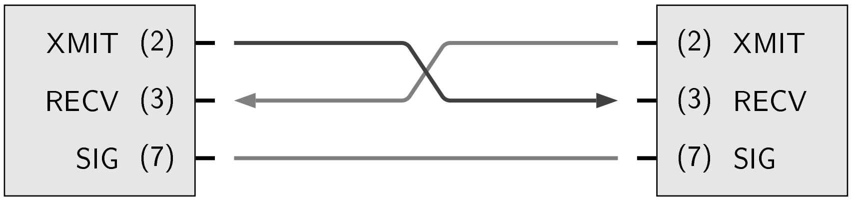


Figure 2.29: The minimum wiring for a Null modem

−3 to −15 volts) indicating the mark condition; see Figure 2.30.

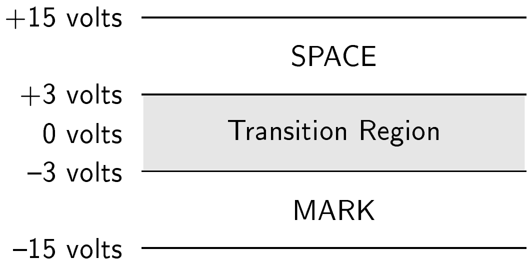


Figure 2.30: RS-232-C signal levels

Data Encoding

The simplest type of encoding is known as **Non-Return to Zero – Level** (or **NRZ-L**) in which each bit value is associated with a *constant* signal level, either positive or negative (hence the name NRZ-L, since during the transmission of a bit, the signal level remains constant and does not return to zero). For example, a mark may be associated with a negative voltage; while a space, with a positive voltage. The only transition that occurs is when the signal changes from a mark to a space (or vice versa). NRZ-L can be implemented with little difficulty. RS-232-C uses NRZ-L for its signal encoding.

For example, the character 'S' transmitted using NRZ-L would appear as shown in Figure 2.31. Remember, the rightmost bit is being transmitted first in this and subsequent examples; all of the examples in this Chapter use NRZ-L.

2.7 Summary

Asynchronous serial communications are a widespread and commonly used method of communicating between devices. A serial communication is one in which data bytes are sent, serially, across a communications channel in a frame. Each frame consists of a start bit, the data bits, an optional parity bit, and one or more stop bits. The signals on the channel are NRZ-L (Non-Return to Zero-Level) and the physical connections are defined by RS-232.

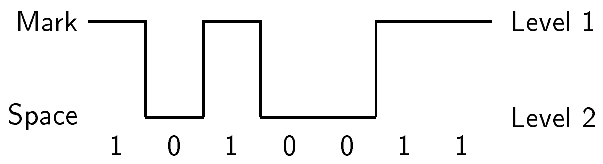


Figure 2.31: Nonreturn to zero – level (NRZ-L)

2.8 Exercises

1. What would be the value of the missing bit in order to get the specified parity (the missing bit is indicated by a '?'):

1 0 ? 0 1 1 0 0	Even
0 0 0 1 0 ? 0 1	Mark
1 0 0 1 1 1 1 ?	Odd
0 0 0 0 ? 0 0 0	Space
1 ? 1 1 1 1 1 0	None

2. Does the width of each bit get larger or smaller as transmission speeds increase? Does the time between the transmission of asynchronous characters increase or decrease as transmission speeds decrease? Explain the result.

3. Describe the two types of overrun that a UART can experience.

4. Calculate the number of data bytes sent in one minute if a UART is to send at 9600 bps with odd parity and two stop bits. Does the number change if even parity is used? Does the number change if one stop bit is sent?

5. Show that the recommended divisors for 300 and 1200 bits per second actually produce the required line speeds.

6. Show, by means of a series of examples, that parity checking (even or odd) will only detect an *odd* number of errors.

7. Explain the conditions whereby mark parity will be equivalent to even (or odd) parity and space parity will be equivalent to even (or odd) parity.

8. In section 2.5, the number of samples on the line performed by the UART each second was calculated for a 9600 bps channel. Calculate line speeds for 2400, 2000, and 1200 bps channels; do they all work out exactly using the 1.8432 MHz clock?

9. Parity calculations were performed in this Chapter by exclusive or'ing the outgoing data bits. Show that the same result can be achieving using exclusive nor'ing.

Chapter 3

Synchronous Communications

3.1 Introduction

Asynchronous communications are intended for **low-volume** data transfers, consisting of several bytes a second, typically between a user typing at a DTE connected to a remote DTE. However, if the transfer between the two DTEs is a **large-volume** transfer, consisting of thousands or even millions of bytes (as could easily happen if a file is being transferred), asynchronous communications may prove to be a less desirable method of data transfer. Consider the following:

Transmission overheads. Each byte that is sent requires at least two, and possibly as many as four, extra control bits. These control bits are necessary since they ensure that the byte is properly framed (the start and stop bits), and offer a degree of error detection (the parity bit). With the inclusion of these control bits, each asynchronous frame transmitted has an overhead of between 20 per cent (a 10-bit frame consisting of an 8-bit byte and 2 control bits: start and stop) to over 36 per cent (an 11-bit frame consisting of a 7-bit byte and 4 control bits: start, parity, and two stop). Therefore, for every million bits that are sent, between 200,000 and 360,000 bits are transmitted as control information.

Error detection and correction. The only means available to the receiving DTE to determine whether the byte has been received correctly is the parity bit. Although the parity bit can help in detecting errors, it can only detect an odd number of bit changes. The parity bit is adequate for low-volume transfers in which each byte is echoed by the remote DTE, since the user can tell immediately whether what has just been typed is echoed correctly. However, echoing each byte in a high-volume transfer halves the throughput.

One way in which DTEs can transfer less control information and more data is to use **synchronous communications**. Synchronous communications are intended to:

- minimize transmission overheads by reducing the amount of control information sent with each **message** (i.e., sequence of data bytes), and
- support better error handling, thereby making error detection and correction more efficient.

The remainder of this Chapter examines how the above two points can be achieved using synchronous communications.

3.2 Reducing Transmission Overheads

Ideally, when sending a message between a pair of DTEs, nothing other than data should be transmitted. For example, in Figure 3.1, three data bytes are transmitted.

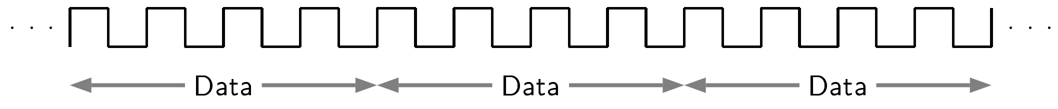


Figure 3.1: Transmitting 'pure' data

However, in Chapter 2, it was shown that two problems must be overcome if data is to be sent between DTEs:

Frame delimiting. The channel idles in the mark state, and any byte sent with one or more leading marks (1s), will not be detected correctly by the receiving DTE.

Timing. The transmitting and receiving DTEs must remain 'in-step' to ensure that each bit is received as transmitted.

In an asynchronous communication, these problems are overcome through the use of additional control bits (i.e., the start and stop bits) and by restricting the number of bits that can be transmitted (i.e., the byte size plus the control bits).

Since synchronous communications permit the transmission of data without embedded control characters, techniques must exist to overcome the problems of frame delimiting and timing.

3.3 Frame Delimiting

In an asynchronous communication, all frames are prefixed with a start bit to distinguish the data from the channel's idle state. A similar solution can be applied to sending a message using synchronous communications: before the first data byte is sent, the transmitting DTE forwards an indication (such as a special byte) that a message is about to follow. For example, see Figure 3.2 (note, in this and subsequent examples, all transmissions occur from *left* to *right*).

Upon receipt of the **start-of-message** indicator, the receiving DTE can then begin removing the message from the channel. If the start-of-message indicator is a byte, there is no need to support special hardware for the transmission and reception of the start-of-message indicator.

A similar argument can be applied to ending the message: the receiving DTE must be able to detect the **end-of-message**, otherwise the DTE will continue receiving indefinitely. Three ways in which a transmitter can indicate to the receiver where the end-of-message occurs include:

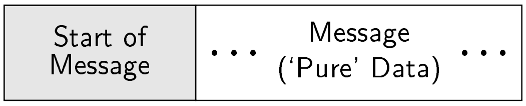


Figure 3.2: Prefacing a message with a start-of-message indicator

- The first, and simplest, is to have all messages consist of the same number of bytes. By using a **fixed message size**, the receiving DTE always expects a message to consist of say, N data bytes. Once the start-of-message indication is detected, the receiving DTE samples the channel for N bytes of data. The transmitting DTE always sends N bytes of data after the start-of-message indication is sent.

  The problem with using a fixed message size is that the message to be sent may consist of *less* than N bytes, requiring the message to be padded with extra bytes that are not part of the message itself. A mechanism must be available whereby the transmitting DTE can inform the receiving DTE of the number of data bytes actually sent.

- A second message delimiting technique involves the transmitter sending the receiver a byte containing a count of the number of bytes in the message. The **byte-count** is usually sent immediately after the start-of-message indication, as shown in Figure 3.3.

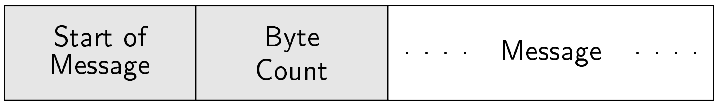


Figure 3.3: Prefacing a message with a byte count

  The byte-count can be used by the receiving DTE to count the number of incoming bytes, when the count reaches zero, the receiving DTE can stop receiving data:

```
receive(Start_Byte);
receive(Count_Byte);

for (i = 0; i != Count_Byte; i++)
     receive(Message[i]);
```

  Although the algorithm for using a byte-count is quite simple, the drawback is that if the byte-count is incorrect (for example, its value may have been changed because of a noisy channel), the receiver will either receive too many or too few bytes.

- A third approach is to transmit a special end-of-message indicator after the last byte of the message. The incoming byte stream is always scanned for the end-of-message indicator; once this has been detected, the receiving DTE has received the entire message. For example:

```
receive(Start_Byte);
```

```
receive(Data_Byte);
i = 0;
while (Data_Byte != END_OF_DATA)
{
     Message[i++] = Data_Byte;
     receive(Data_Byte);
}
```

Messages sent with byte-counts often have an end-of-message indicator appended to the last byte of the message.

The start-of-message, message, end-of-message sequence is known as a **frame** and is shown in Figure 3.4.

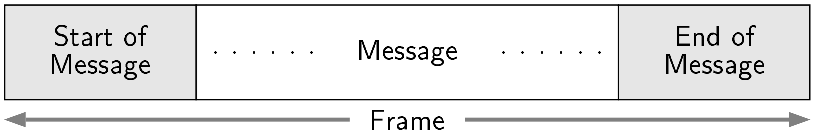


Figure 3.4: A frame encapsulating a message

Should the end-of-message indicator be lost (for example, due to noise), the receiving DTE will receive more bytes than were originally sent. The solution (which can also be used in the byte-count method), is to limit the number of bytes in a frame that can be sent (and hence received). For example, assume that a frame cannot contain more than N bytes and that the frame is delimited by the bytes `START` and `END`, then the receive data algorithm could be written as follows:

1. Wait for `START` byte.
2. Set `byte_count` to zero.
3. Check each data byte as it arrives:

`END` byte:
    Message with valid number of bytes received. Terminate loop.
`default`:
    Check `byte_count`:
    `0 .. N-1`:
        Store byte in buffer, increase `byte_count` by 1, repeat from step 3.
    `N`:
        Error, missing `END` byte. Terminate reception process.

The above algorithm ensures that a receiving DTE does not continue receiving indefinitely (and possibly run out of buffer space).

In an asynchronous communication a frame consists of a single byte; the number of bytes in a synchronous frame depends upon several factors, including:

- the number of bytes needed to represent the control sequences and other framing information. Ideally, this should be very small in relation to the total frame size.
- the amount of storage available at the receiving DTE. The data bytes within the frame must not exceed the receiving DTE's storage capacity.
- the amount of information that must be retransmitted should an error be detected in a frame. For example, if a file of one-million bytes is transmitted as a single frame, and if one byte is found to be in error, the entire one-million bytes must be retransmitted. However, transmitting the file as a series of one-thousand byte frames means that the amount of information to be retransmitted is considerably less.
- the time required to transmit a frame must not exceed the length of time the hardware can remain in synchronization. If synchronization is lost, the contents of the frame will be lost, potentially requiring another transmission (preferably using a smaller frame size).

3.4 Timing

Information that is transmitted serially between DTEs is transmitted a bit-at-a-time. To ensure that there is a high probability that each transmitted bit is received correctly, both the transmitting DTE and receiving DTE must be synchronized.

In Chapter 2, it was shown that a UART's clock is designed to remain in step with the remote UART only for the time it takes to send a single byte. The transmitter's and receiver's clocks are synchronized by the start bit: once the center of the start bit is found, the channel can be read once every bit-time. As soon as the receiving DTE has assembled the byte, the two DTEs are no longer synchronized, meaning that the next byte sent requires that a start bit be prefixed.

If 'pure' data is sent between the communicating DTEs, the DTEs must still remain in-step to ensure that each byte sent is the same as the byte that is received. However, a synchronous communication, unlike an asynchronous communication, requires that the DTEs remain in-step for tens, hundreds, or possibly even thousands of bytes. Keeping the DTEs synchronized is not a trivial matter, since clocks can drift and result in the loss of synchronization.

As with so many other aspects of data communications, there are a number of techniques available which permit the DTEs to transmit a frame of 'pure' data at the same time staying 'in-step':

Special synchronization sequences. Special synchronization byte sequences can be scattered throughout the frame. These characters are detected by the hardware of the receiving DTE which can realign its clock to correct for any clock drift that may have occurred. As shown in Figure 3.5, frames are often prefixed by the synchronization sequences to ensure that the clocks are properly aligned before the transmission begins.

Since the synchronization sequence is being embedded within the frame of 'pure' data, it is necessary to distinguish between synchronization sequences that are used to realign the clocks and data bytes that happen to have the same value as the synchronization sequence (an event that could occur if a binary file is transmitted). Protocols have been designed to overcome this problem and are discussed in subsequent sections.

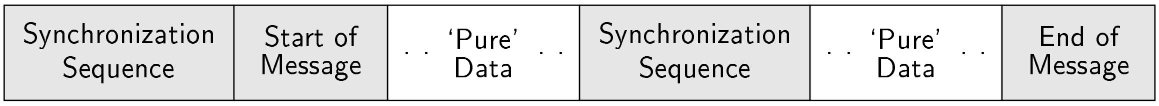


Figure 3.5: Achieving synchronization using synchronization sequences

Separate timing channel. Another solution to the problems associated with the synchronization of DTEs is to transmit on two channels: one reserved for the data bits, the other for a clocking signal. With this method, as illustrated in Figure 3.6, the receiving DTE knows exactly when to sample the channel since the transmitting DTE indicates the center of the data bit with a timing pulse.

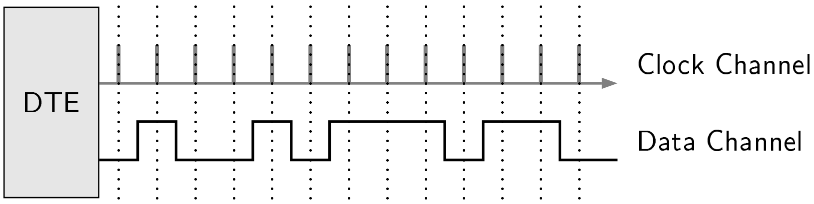


Figure 3.6: Data transmission using a separate timing channel

There is another reason for having one of the DTEs supply the clocking signal: the 'other' DTE may not support a clock:

- If the transmitting DTE is supplied with the clock, it sends each data bit on one channel and a timing pulse on the other. The receiving DTE detects the timing pulse and reads the data channel to determine the value of the bit.
- However, if the transmitting DTE is not supplied with a clock, the receiving DTE must supply the timing signal. In this situation, the transmitting DTE waits for a timing pulse and then sends a bit: the receiving DTE can sample the channel after the timing pulse has been sent, as shown in Figure 3.7.

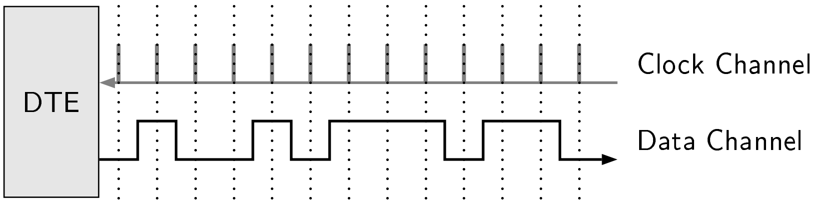


Figure 3.7: Data transmission using an external clock channel

Encoding the timing into the data. A third approach to keeping the communicating DTEs synchronized is to encode the clocking information directly into each bit as it is transmitted.

In this way, as the signal arrives, the receiving DTE extracts both the value of the bit and the clock signal.

One of the best known examples of encoding the clock into the data stream is **Manchester encoding**, in which the middle of each bit contains the clock signal as well as an indication as to the value of the bit; see Figure 3.8.

Figure 3.8: Manchester encoding

In Manchester encoding, a low-to-high transition in the middle of the bit interval means space (or 0), while a high-to-low transition indicates a mark (or 1). Figure 3.9 shows the encoding of the byte 'S' using Manchester encoding.

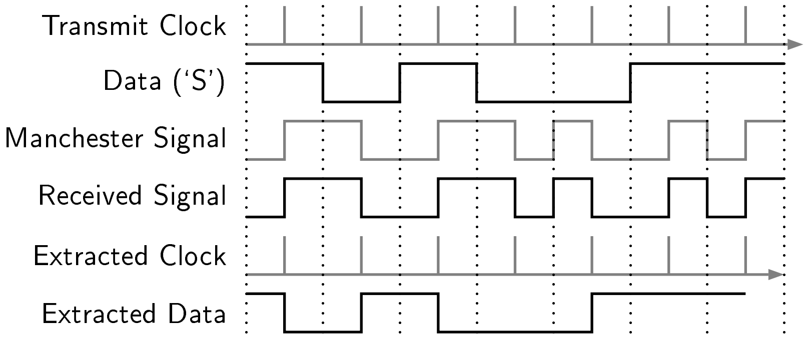


Figure 3.9: The Manchester encoding of the byte 'S'

The extracted data is half a signal behind the transmitted data because the value of each bit is determined at the point where the clock signal is extracted (i.e., the centre of the bit).

3.5 Improving Error Detection

In a full duplex asynchronous communication, error detection is typically the responsibility of the user. In remote echoing, if the character typed is not the same as the character echoed, the user can deduce that an error has occurred, forcing them to retype the character. Should the receiving DTE detect a parity error, an error message can be returned to the user, requesting that the character be re-entered.

However, as Figure 3.10 illustrates, even if each byte is echoed when received, there is no guarantee that the transmitting DTE can determine that an error has occurred simply by checking the echoed byte and its parity.

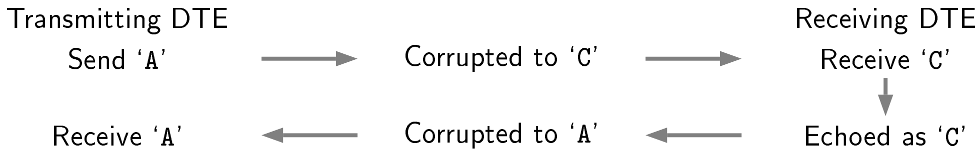


Figure 3.10: A limitation of error detection using echoing

Should the transmitting DTE determine that an error has occurred, it must inform the receiving DTE that the last byte sent was in error and the correct byte follows. For example, the character '%' could be sent by the transmitter when the echoed character is not the same as was originally sent; a possible scenario is shown in Figure 3.11.

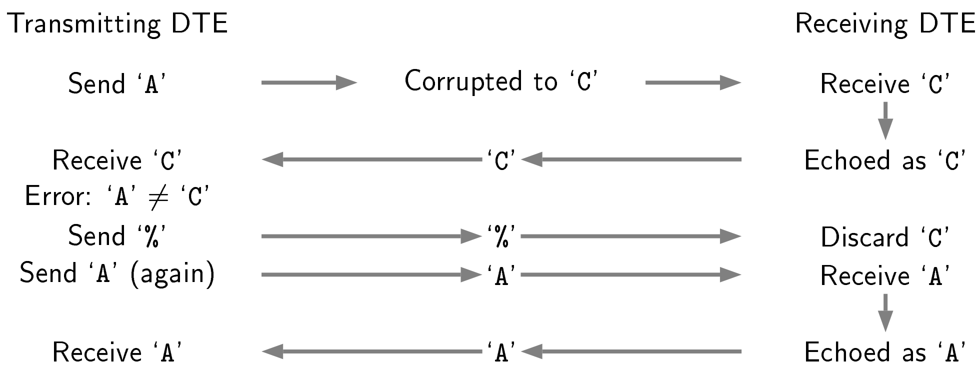


Figure 3.11: An attempt at error correction using echoing

This solution simply leads to other problems:

- a mechanism must be devised whereby it is possible to send a '%' as data (that is, the receiving DTE must be able to distinguish a data '%' from a retransmission request '%'),
- if an error occurs during a retransmission (especially if the '%' is corrupted) a protocol must be devised which allows the transmitter to inform the receiver to discard the corrupted '%', and
- if, for some reason, the '%' is lost, the receiver will accept the retransmitted byte but not discard the original corrupted byte.

Admittedly, it is possible to write software to handle the above and other problems, but in the end, the overheads (such as echoing each byte), will make the solution unnecessarily clumsy and expensive. In addition, the parity check is costly in terms of the amount of information that must be sent and the level of detection that is achieved. For example, if a message of 200 bytes is sent, 200 parity bits are also sent.

Fortunately, a number of techniques exist which are more accurate in their ability to detect errors, and are less costly in terms of the amount of information transmitted when applied to large volumes of data.

3.5.1 Longitudinal Redundancy Check (LRC)

The parity check is used to check an individual byte by summing the bits *across* the byte. A second type of parity check, known as a **Longitudinal Redundancy Check** or **LRC**, can be applied to a 'column' of bits within a message. That is, whereas parity is applied to a single byte, the LRC is applied to *all* of the bytes.

The LRC is a byte, initially set to `0xFF`. Prior to transmission, each data byte is exclusive-or'ed (XOR) with the LRC. (The output from an exclusive-or is `0` if the bits are the same and `1` if the bits are different.)

Once all of the data bytes have been transmitted, the LRC is sent. For example, consider the transmission of a message (`msg`) of length N:

```
LRC = 0xFF;
for (i = 0; i < N; i++)
{
   LRC ^= msg[i];  /* Equivalent to: LRC = LRC ^ msg[i];  ^ denotes XOR */
   xmit(msg[i]);
}
xmit(LRC);
```

For example, if the above algorithm were applied to the following five bytes:

```
01101100
10101111
01110101
11100010
00010111
```

the LRC result would be as shown in Figure 3.12.

0	1	1	0	1	1	0	0	Byte 1
1	0	1	0	1	1	1	1	Byte 2
0	1	1	1	0	1	0	1	Byte 3
1	1	1	0	0	0	1	0	Byte 4
0	0	0	1	0	1	1	1	Byte 5
1	0	1	1	1	1	0	0	Check character (LRC)

Figure 3.12: The LRC after XOR'ing five data bytes

The receiver follows a similar algorithm, initially setting the LRC to `0xFF`. As each byte is received, it is exclusive-or'ed into the receiver's LRC:

```
LRC = 0xFF;
for(i = 0; i < N; i++)
{
   recv(&msg[i]);
   LRC ^= msg[i];
}
recv(&last);
LRC ^= last;
```

Assuming that no errors have occurred during the communication, the transmitter's LRC will be the same as the receiver's: meaning that when the receiver exclusive-or's the transmitter's LRC, the result should be zero. Figure 3.13 shows the results of transmitting the five data bytes followed by the LRC byte.

Transmitter			Receiver
LRC = 11111111			LRC = 11111111
send(01101100)	→	recv(01101100)	
LRC = 10010011			LRC = 10010011
send(10101111)	→	recv(10101111)	
LRC = 00111100			LRC = 00111100
send(01110101)	→	recv(01110101)	
LRC = 01001001			LRC = 01001001
send(11100010)	→	recv(11100010)	
LRC = 10101011			LRC = 10101011
send(00010111)	→	recv(00010111)	
LRC = 10111100			LRC = 10111100
send(LRC)	→	recv(10111100)	
			LRC = 00000000

Figure 3.13: The transmission of a message and its LRC

When the last byte is received and the LRC is zero, it means that no errors were detected. Since the LRC algorithm, like that of the parity check, uses exclusive-or'ing to calculate the check information, it should not be surprising that the LRC is no more accurate than the parity check. For example, if an odd number of bits within a column are inverted due to noise, then the error can be detected; however, an even number of bit changes cannot be detected. This problem is illustrated in Figure 3.14.

In some manuals the term **vertical redundancy check** or **VRC** can be found in addition to LRC. VRC is simply another term for parity check. The terms VRC and LRC refer to the error checking used with magnetic computer tapes. In tape lexicon, vertical refers to information stored across the width of the tape (i.e., data bytes stored in parallel with a parity bit), while longitudinal refers to information stored along the length of the tape. In Figure 3.15, a vertical redundancy check is applied across the tape and the longitudinal redundancy check is applied along the length of the tape.

Transmitter			Receiver	
LRC = 11111111				LRC = 11111111
	send(01101100)	⟶	recv(01101100)	
LRC = 10010011				LRC = 10010011
	send(10101111)	NOISE	recv(101011$\boxed{0}$1)	
LRC = 00111100				LRC = 001111$\boxed{1}$0
	send(01110101)	⟶	recv(01110101)	
LRC = 01001001				LRC = 010010$\boxed{1}$1
	send(11100010)	⟶	recv(11100010)	
LRC = 10101011				LRC = 101010$\boxed{0}$1
	send(00010111)	NOISE	recv(000101$\boxed{0}$1)	
LRC = 10111100				LRC = 101111$\boxed{0}$0
	send(LRC)	⟶	recv(10111100)	
				LRC = 00000000

Figure 3.14: An error that is undetectable by the LRC

0	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	Bit 7
1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	Bit 6
0	1	0	1	0	0	1	0	0	0	1	0	0	0	0	1	Bit 5
0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	Bit 4
0	0	0	1	1	0	1	0	1	0	1	0	0	1	1	0	Bit 3
1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	1	Bit 2
1	1	1	0	1	0	0	0	1	0	1	0	1	0	1	1	Bit 1
0	1	1	1	0	0	0	1	0	0	0	1	1	0	0	1	VRC (Odd)
⇑ LRC																

Figure 3.15: VRC and LRC encoding

Error detection can increase by two to four orders of magnitude over either VRC alone or LRC alone if the two techniques are used together (that is, each byte is sent with a VRC and is included in an LRC). In Figure 3.16, five 8-bit bytes are sent (with hardware generated *odd* parity), followed by the LRC.

Although the LRC includes all the data bits, it does *not* include the hardware generated parity check bit. In the above example, the LRC's parity check is 0 (because the LRC is sent as odd parity and it has an odd number of bits set) rather than 1 (as it would be if the parity bits were included in the LRC).

However, even when used together, combinations of errors can still cause the error detection to fail. Figure 3.17 shows the effects of an even number of bit changes in the same columns of an even number of bytes. In this example, neither parity nor LRC errors are detected, even though the

1	0	1	1	0	1	1	0	0	Byte 1
1	1	0	1	0	1	1	1	1	Byte 2
0	0	1	1	1	0	1	0	1	Byte 3
1	1	1	1	0	0	0	1	0	Byte 4
1	0	0	0	1	0	1	1	1	Byte 5
0	1	0	1	1	1	1	0	0	Check character (LRC)
⇑									
Parity check (hardware generated)									

Figure 3.16: The inclusion of a parity check with an LRC

message has been received in error.

Transmitter		Receiver
LRC = 11111111		LRC = 11111111
send(01101100) (Parity = 1)	⟶	recv(01101100) (Parity = 1)
LRC = 10010011		LRC = 10010011
send(10101111) (Parity = 1)	NOISE	recv(10$\boxed{0}$011$\boxed{0}$1) (Parity = 1)
LRC = 00111100		LRC = 00$\boxed{0}$111$\boxed{1}$0
send(01110101) (Parity = 0)	⟶	recv(01110101) (Parity = 0)
LRC = 01001001		LRC = 01$\boxed{1}$010$\boxed{1}$1
send(11100010) (Parity = 1)	⟶	recv(11100010) (Parity = 1)
LRC = 10101011		LRC = 10$\boxed{0}$010$\boxed{0}$1
send(00010111) (Parity = 1)	NOISE	recv(00$\boxed{1}$101$\boxed{0}$1) (Parity = 1)
LRC = 10111100		LRC = 10$\boxed{1}$111$\boxed{0}$0
send(LRC) (Parity = 0)	⟶	recv(10111100) (Parity = 0)
		LRC = 00000000

Figure 3.17: An error undetectable by both LRC and VRC

3.5.2 Cyclic Redundancy Check (CRC)

In certain applications, sending blocks of information with an LRC, VRC, or both, may provide sufficient error detection, especially if the channel is known to be reliable and reasonably error free. However, there are situations which call for as near-to-perfect error detection as can possibly be applied. Consider the electronic transfer of funds between bank branches. If, during the transfer of funds, the channel is subject to noise, undetected errors could be disastrous for the bank and its customers, since the amounts received might not correspond to the amounts sent. Although LRC and VRC can detect odd numbers of errors, it is possible for errors to go undetected: what is needed is an error detection technique that is several orders of magnitude more effective than LRC and VRC.

The technique used in many applications requiring better error detection is a **cyclic redundancy check** or **CRC**.

CRC Fundamentals

The principles associated with the other error detection techniques discussed earlier in the book are also applicable to the cyclic redundancy check algorithm. That is, the transmitting entity calculates the error check using the data from the message, appends a check sequence to the end of the message, and then transmits the message and the check sequence to the receiving entity in a frame. The receiving entity performs the same calculation on the incoming message and determines whether the message is valid by examining the result of the calculation. The cyclic redundancy check is particularly attractive since it offers greater error detection accuracy through the use of polynomial division.

Basically, the cyclic redundancy check algorithm treats the message as a single bit stream in which each bit is taken as a coefficient of a polynomial. In general, a message k bits long has k terms and is a polynomial of order $k-1$. A message with terms $m^{k-1} + m^{k-2} + \cdots + m^2 + m^1 + m^0$ can be written as the polynomial:

$$M(x) = m_{k-1}x^{k-1} + m_{k-2}x^{k-2} + \cdots + m_2x^2 + m_1x^1 + m_0$$

For example, the message `101101001` can be written as the polynomial:

$$1x^8 + 0x^7 + 1x^6 + 1x^5 + 0x^4 + 1x^3 + 0x^2 + 0x^1 + 1x^0$$

and simplified to $x^8 + x^6 + x^5 + x^3 + x^0$.

As with LRC and VRC, the message is transmitted with the check bits (commonly referred to as the CRC). Together, the message and the check bits make a unique cyclic code word. The objective of the CRC algorithm is to ensure that the vast majority of errors that can possibly occur are detected by the receiving entity. By dividing the message polynomial by a **generator polynomial**, a set of check bits unique to the message can be obtained.

The generator polynomial, $G(x)$, is a polynomial of degree g, which must be less than the degree of the message polynomial, $M(x)$. $G(x)$ is always odd (i.e., the lowest order term has a value of 1) and must have a value greater than one. Unless otherwise indicated, all calculations are performed on binary digits using modulo 2 addition without carries ('$\oplus$' denotes modulo 2 addition).

The transmission algorithm is as follows:

1. The message $M(x)$ is multiplied by x^g, (i.e., the message is shifted left by g bit positions; these bit positions are cleared).

2. The result of the multiplication is divided by the generator polynomial, $G(x)$, giving a quotient, $Q(x)$, and a remainder, $R(x)$:

$$\frac{x^g \times M(x)}{G(x)} = Q(x) \oplus \frac{R(x)}{G(x)}$$

The remainder $R(x)$ is always less than the divisor $G(x)$ since the maximum number of bits in the remainder is g.

3. The remainder, $R(x)$, is added to the shifted message (i.e., the lower g bits), producing the frame to be transmitted, $T(x)$:

$$T(x) = x^g \times M(x) \oplus R(x)$$

The receiving entity receives the frame $T(x)$, and proceeds to divide the message by the generator polynomial, $G(x)$. Since the transmitted message includes the remainder from the original division, the frame $T(x)$ should be exactly divisible by the generator polynomial $G(x)$:

$$\frac{T(x)}{G(x)} = \frac{x^g \times M(x) \oplus R(x)}{G(x)}$$

substituting $Q(x) \oplus \frac{R(x)}{G(x)}$ for $x^g \times \frac{M(x)}{G(x)}$, one finds:

$$\frac{T(x)}{G(x)} = Q(x) \oplus \frac{R(x)}{G(x)} \oplus \frac{R(x)}{G(x)}$$

Since any number exclusive-or'ed with itself is zero (such as $\frac{R(x)}{G(x)}$), a successful transmission will result in a remainder of zero.

Example

Consider the transmission of the message `101101001` using the generator polynomial `101001` ($x^5 + x^3 + 1$). Following the algorithm described above, one finds:

1. The generator polynomial $G(x)$ has $g = 5$, meaning that the message $M(x)$ must be shifted left by five:

$$2^5 \times \texttt{101101001} = \texttt{10110100100000}$$

2. The shifted message is then divided by the generator polynomial:

```
                        1 0 0 1 0 1 0 1 0   (Quotient)
1 0 1 0 0 1 | 1 0 1 1 0 1 0 0 1 0 0 0 0 0
            ⊕ 1 0 1 0 0 1
                    1 0 0 0 0 1
                  ⊕ 1 0 1 0 0 1
                        1 0 0 0 0 0
                      ⊕ 1 0 1 0 0 1
                            1 0 0 1 0 0
                          ⊕ 1 0 1 0 0 1
                                1 1 0 1 0   (Remainder)
```

The quotient, $Q(x)$ is discarded.

3. The remainder $R(x)$, `11010`, is added to the shifted message, producing $T(x)$, consisting of $M(x)$ and $R(x)$. $T(x)$ is therefore `10110100111010`.

The receiving entity accepts the transmitted frame and performs the division using the same generator polynomial:

```
                          1 0 0 1 0 1 0 1 0   (Quotient)
            ----------------------------------
1 0 1 0 0 1 | 1 0 1 1 0 1 0 0 1 1 1 0 1 0
            ⊕ 1 0 1 0 0 1
            -------------
                    1 0 0 0 0 1
                  ⊕ 1 0 1 0 0 1
                  -------------
                        1 0 0 0 1 1
                      ⊕ 1 0 1 0 0 1
                      -------------
                            1 0 1 0 0 1
                          ⊕ 1 0 1 0 0 1
                          -------------
                                      0   (Remainder)
```

The result of the division is zero, indicating that no errors were detected in frame.

However, should one or more bits become inverted (i.e., `0` to `1` or vice versa), the division should result in a non-zero remainder. For example, if the frame received was `101`$\boxed{\texttt{01}}$`100111010` (rather than `101`$\boxed{\texttt{10}}$`100111010`), the division would proceed as follows:

```
                          1 0 0 1 0 1 0 1 0   (Quotient)
            ----------------------------------
1 0 1 0 0 1 | 1 0 1 0 1 1 0 0 1 0 0 0 0 0
            ⊕ 1 0 1 0 0 1
            -------------
                  1 0 0 0 1 1
                ⊕ 1 0 1 0 0 1
                -------------
                      1 0 1 0 1 0
                    ⊕ 1 0 1 0 0 1
                    -------------
                                1 1 1 0   (Remainder)
```

The receiving entity can reject the frame since the remainder after the division is non-zero.

CRC Generators

All communication channels can be subject to some form of fault which leads to one or more bits in the transmitted frame becoming inverted. The CRC generator polynomial must be designed to detect invalid frames.

It is possible to represent the error itself as a polynomial that is exclusive-or'ed to the frame (if the bits in error are represented as coefficients, the corresponding bits in the frame will be inverted). Using the example from above in which the transmitted frame, `10110100111010`, is corrupted to `10101100111010`, the bits in error are as follows:

Transmitted frame	10110100111010
Error bits:	00011000000000
Received frame:	10101100111010

In this example, the error polynomial $E(x)$ is $x^{10} + x^9$. Error bits corresponding to polynomials containing $G(x)$ as a factor will be undetected. The question is how to develop a generator polynomial that will detect errors within the transmitted frame.

The CRC generating polynomial is chosen to detect the following four types of error:

Single-Bit Errors. If the transmitted frame has a single bit in error, the error polynomial $E(x)$ contains a single term, x^e, where e has a value less than the total number of bits in the frame $T(x)$. By having more than one term in the generator polynomial $G(x)$, x^e cannot be divided evenly, meaning that all single-bit errors will be detected.

Double-Bit Errors. A double bit error, like the single bit error, can be represented by the error polynomial. In this case, the polynomial consists of two terms, $E(x) = x^i + x^j$; as before, i and j are both less than the number of bits in the frame, $T(x)$. $E(x)$ can be expressed as $E(x) = x^i \times (x^{j-i} + 1)$ if $i < j$. By ensuring that the generator polynomial has a factor with three terms, neither x^i nor $x^{j-i} + 1)$ are divisible by the generator, meaning that all double-bit errors will be detected.

Odd Number of Errors. If the frame is received with an odd number of errors, $E(x)$ will have an odd number of terms. Since no polynomial with an odd number of terms has $x + 1$ as a factor, if $G(x)$ has $x+1$ as a factor, all frames with an odd number of errors will be detected.

Error Bursts. An error burst is a sequence of at least two incorrect bits separated by a series of other bits that may be in error. Error bursts less than or equal to g bits in length (i.e., the degree of the generator polynomial) will be detected as errors. If the burst is exactly $g + 1$ bits long (i.e., equal to the number of bits in the generator polynomial), the error will be detected if $E(x)$ does not equal $G(x)$. The probability of the error going undetected is $\frac{1}{2}^{(r-1)}$ (where r is the number of bits in the remainder), since the first and last bits of $E(x)$ are the same as $G(x)$, meaning that the remaining $r - 1$ bits must be equal. Finally, if the burst exceeds g, the probability of the error going undetected can be shown to be $\frac{1}{2}^{r}$.

The number of bits in the CRC is typically a function of the character (i.e., byte) size used by the communicating entities since the supporting hardware handles quantities of this size. For example, in eight-bit communications, the size of the remainder is typically sixteen or even thirty-two bits in length, meaning that the generator polynomial can be seventeen or thirty-three bits long.

There are a number of CRC generators available for use with synchronous communication protocols, some of the better known include (note, the X in CRC-X refers to the number of bits transmitted in the remainder):

CRC-12. A 12-bit CRC intended for use with synchronous protocols supporting 6-bit bytes. The generator polynomial is $x^{12} + x^{11} + x^3 + x^2 + x + 1$. The initial value of the CRC is zero. If, after the end-of-message is detected, the receiver's CRC result is zero, it is assumed that message was received correctly.

CRC-16. CRC-16 is a 16-bit CRC used by IBM for the transmission of 8-bit character codes. The generator polynomial is $x^{16} + x^{15} + x^2 + 1$. The initial value of the CRC is zero. The message is assumed to be correctly received if the receiver's CRC result is zero.

CRC-CCITT. A 16-bit CRC developed by IBM for their SDLC protocol and also used by the ISO HDLC protocol (both protocols are discussed later); it employs the following generator polynomial: $x^{16} + x^{12} + x^5 + 1$. The CRC is initialized to `0xFFFF` and the CRC bits are inverted upon transmission. The received message is deemed correct if, after the end of the message, the receiver's CRC has a value of `0xF0B8`.

CRC-32. A 32-bit CRC intended for use primarily in local area networks and some military applications. The generator polynomial is:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

The initial value of the CRC is `0xFFFFFFFF`. As with the CRC-CCITT, the transmitter inverts the CRC bits prior to transmission. The receiving entity's CRC should equal `0xDEBB20E3` if the message is received correctly.

Although the generator polynomials are different, the basic algorithm is the same for each.

CRC Implementations

The first observation that one must make when considering the implementation of a CRC generation routine (in either hardware or software) is *how* is it possible to perform division on a message that could be thousands of bits long, when most machines can only perform 32-bit arithmetic (or less in many cases).

Interestingly enough, the entire division process can take place in a shift register and can be performed on each bit as it is being transmitted serially. For example, Figure 3.18 shows a possible shift register design for CRC-16 calculations (polynomial generator: $x^{16} + x^{15} + x^2 + 1$).

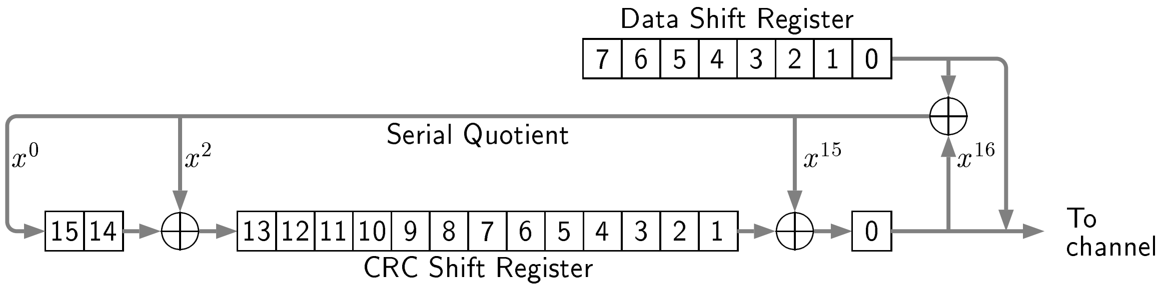


Figure 3.18: A shift-register implementation of CRC-16

Prior to the first byte being transmitted, the CRC shift register is initialized to `0x0000`; the CRC shift register contains the intermediate remainder. The shift register emulates polynomial division by performing modulo-2 subtraction on the terms x^0, x^2, and x^{15}. The result of the exclusive-or between the low-order data bit and the high-order bit of the remainder (x^{16}) is either a `0` or a `1`: a `0` value causes the remainder to rotate unchanged, whereas a `1` inverts the value of each divisor polynomial term; equivalent to obtaining a new intermediate remainder value by including bits from the dividend. Each bit from the Data Shift Register is put onto the channel for transmission as well as being fed into the CRC circuit. When the last data bit has been included in the intermediate remainder, the bits in the shift register are clocked onto the channel.

CRC Utilities

CRCs can be calculated in software or hardware. Two CRC-16 utilities available from the author's web site show how CRCs can be calculated in software; the first is a software emulation of the CRC shift register circuit described above, while the second performs a table lookup:

Emulation. The shift register shown above can be emulated in software by looping through each bit in the data byte prior to the transmission of the data.

`Crc_calc()` (found in the file `crcsr.c`) contains an implementation of the shift register written in C. The software is optimized, upon entry to `crc_calc()`, an intermediate value of the remainder is produced by exclusive-or'ing the data `byte` with the `crc`. The low-order bit in the new intermediate remainder is right-shifted. Whenever a carry is detected, the polynomial `POLY` is exclusive-or'ed into the remainder (note that `POLY` is simply the octal representation of $x^{15} + x^2 + x^0$).

The intermediate remainder is returned by `crc_calc()`.

Table Lookup. By studying the emulation software, a number of patterns emerge that can be used to reduce the time associated with division by repeated shifting. These patterns allow the creation of a table that can produce the next intermediate remainder from the current remainder and the data byte. A second, faster version of `crc_calc()` using table lookup can be found in `crctbl.c`.

The table is based upon the observation that each bit in the new intermediate remainder is always made of the same combinations of data bits and the previous value of the remainder. For example, bit `15` of the new remainder always contains the data bits exclusive-or'ed with the low-order bits of the old intermediate remainder. Similarly, bits `1` through `5` of the new remainder are the shifted values of bits `9` through `13` of the old remainder. The table `crc_tbl` is an array of 256 integers, each of which represents the different intermediate remainder results associated with the subscript. The table is created in `gen_tbl()`.

Upon entry into `crc_calc()`, the byte to be included in the CRC is exclusive-or'ed with the lower 8 bits of the CRC, producing an index into the table. The `crc` is then shifted right and exclusive-or'ed with the bit pattern from `crc_tbl`; this is the new intermediate CRC.

Prior to the first call to `crc_calc()`, `gen_tbl()` must be called to generate the array `crc_tbl`.

Not surprisingly, both methods produce the same results for the same messages; however, table lookup is the faster of the two methods.

3.6 Summary

This Chapter has examined synchronous communications and its uses; notably, the transfer of large volumes of data with the minimum of overheads. Many of the issues involved in synchronous communications have been discussed, including:

- synchronous communications involve the transmission of 'pure' data (i.e., no start and stop bits); to achieve this, special frames are needed with some form of delimiter.
- the receiving DTE can verify that the transmitted frame has been received correctly through the use of a series of one or more check bytes that are generated by the transmitter. Techniques such as the longitudinal redundancy check (LRC) can be used, but they offer limited checking capabilities on the data within the frame. However, more accurate frame checking is possible using cyclic redundancy checks or CRCs. A CRC treats the entire message as a long dividend, and divides the message by a CRC generator; the remainder is transmitted as part of the frame. A number of different CRC standards exist; CRCs can be generated in hardware and software.

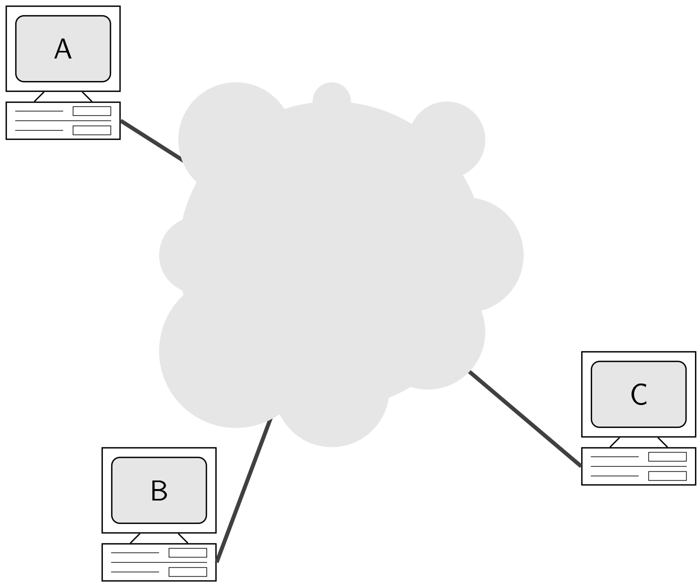
The combination of being able to send 'pure' data *and* having a more advanced error detection makes synchronous-type communication more appropriate that asynchronous communications to high-speed applications. In subsequent Chapters, it will be shown that few computer networks use asynchronous communications.

3.7 Exercises

1. Show that synchronous communications offer better throughput than asynchronous.
2. If synchronous communications are so much more efficient that asynchronous, would it make sense to replace all asynchronous terminals with synchronous terminals? Consider the overheads.
3. Compare the overheads involved in the calculation of a message's CRC using the software shift-register implementation against the table lookup method.

Part 3

Computer communications over a shared channel



A shared channel is one in which two or more computers are attached. Unlike an unshared channel, users of a shared channel require a means of **identifying** the intended destination of each message that is sent on the channel. The additional information (i.e., **control information**) can be handled in several different ways.[1]

Multidrop

One approach is to have a centralized computer that **polls** all other computers on the shared channel, asking whether each one has anything to send. If there is something to send, the message is sent to the central computer for subsequent processing. This type of channel, in which there is a single controlling computer and several secondary ones is often referred to as a **multidrop line**.

[1]The grey 'blob' through which all connections from the computers enter in the above figure is referred to as a *network cloud*. A cloud represents any network (or networks), depending upon its interpretation.

Multiplexing

The centralized polling computer can be eliminated through the use of a **multiplexed** or shared channel. At either end of the channel is a device known as a **multiplexer** to which other devices are connected. The different approaches to multiplexing are discussed in Chapter 4.

Local Area Networks

Prior to the development of the microprocessor, most processing was centralized in nature, requiring users to communicate with a single mainframe to which all equipment was attached and upon which their information was stored. The obvious advantage of having a single, centralized computer is that the sharing of resources (such as equipment or information) is a relatively simple procedure controlled by the operating system. For example, a laser printer on a single, centralized computer is accessible to all users through a series of commands which place the object to be printed on the computer's laser printer queue. However, the one great disadvantage of having a single centralized computer is that should the central computer fail, it is impossible for a user to access any of the resources associated with the computer.

The microprocessor has changed the way in which the computer can be used. Now, instead of having all resources available to all users on a single computer, the resources necessary to perform certain tasks may reside on one or more distinct computers. One clear advantage in using a number of computers to perform the tasks that were originally performed by a single centralized computer is that should one of the computers fail, users on the other computers may continue their processing.

However, in a decentralized system consisting of many small, independent computers, it may not be cost effective to supply each computer with the same set of devices (such as laser printers), since the device may remain idle for long periods. Instead, only a handful of computers may be connected to certain, costly devices. For example, a problem clearly arises when people whose computers are not connected to a laser printer want to print information on a laser printer. One solution is to have the users carry their information on a diskette to a computer attached to a laser printer, or wheel the laser printer on a cart from office to office as and when it is required. Of course, neither of these approaches are completely satisfactory since they can result in a great deal of wasted time on the part of the user.

A similar problem arises when users want to share information. If the information resides on one computer, 'sharing' may be reduced to copying and distributing a diskette. This in turn leads to other problems, such as which user will manage the information and how will changes to the information be handled.

Ideally, what is needed is a mechanism whereby individual computers can communicate with other computers (such as those attached to the laser printers or those with information to be shared) at speeds approaching or exceeding the time to access a disk. To achieve these speeds, **local computer networks** are often used to interconnect locally distributed computers. A local computer network (or more commonly, **local area network**) is similar to a wide area network in that messages are still sent in packets and the stations on the network are identified using an address; however, a general definition of a local area network requires that the network conform to a broad set of requirements, including:

- support data rates several orders of magnitude greater than wide area networks (typically 10 to 100 megabits per second);
- span distances of several kilometres (often within a single organization);
- support several hundred independent devices;
- offer simplicity of functionality and performance;
- have low error rates, high reliability, and the minimum of dependence upon any centralized components or control;
- permit efficient use of shared resources (including the network itself);
- maintain stability under high load;
- allow fair access to the system by all devices;
- permit ease of re-configuration and maintenance;
- be low cost.

Local area networks (or **LAN**s), like wide area networks, can be discussed from a variety of different angles, such as topologies, access methods, packet formats, and performance. However, unlike most wide area networks which exhibit similar characteristics, there is no single 'generic' local area network. For example, the most popular LAN topologies are the bus, ring, and star. Accessing methods are dictated, in part, by topology; and the performance of different LANs varies greatly depending upon the amount of traffic on the network (the **network load**).

Chapters 5 and Chapter 6 are devoted to the most widespread local area network topologies, notably bus and ring local area networks, respectively. In each cases, design issues and existing networks are discussed.

The last Chapter of Part 3, discusses network monitoring tools for both asynchronous communications and local area networks. Network monitoring tools, such as an Ethernet **sniffer**, are extremely valuable in tracking down software or hardware communication problems.

Chapter 4

Multiplexing

4.1 Introduction

When people work at a terminal they often pause, look around, think, scratch their heads, turn the page of the material they happen to be typing, or whatever: the end result being that the channel connecting the terminal to the central computer is idle a considerable amount of the time. Even data entry clerks who can type hundreds of words a minute and are working 'flat-out' cannot keep the channel busy for any more than a fraction of its total capacity. As an example, the number of 10 character 'words' that a typist would have to type each minute to keep a 9600 bits per second line full is:

$$\frac{9600\ bits\ per\ second \times 60\ seconds\ per\ minute}{10\ bits\ per\ byte \times 10\ bytes\ per\ word} = 5760\ words\ per\ minute$$

From this simple example, it is clear that a channel devoted to a single terminal is typically idle much more than it is active. If the channel could be *shared* amongst several terminals, the number of individual cables from user's DTEs to the central computer could be reduced dramatically, thereby cutting down on the expense of laying and maintaining the cables. The sharing of a single communication channel amongst several DTEs is known as **multiplexing**.

In addition to reducing cabling costs, multiplexing can also reduce the costs associated with telephone access of remote DTEs. For example, consider the costs to a company that has five telephone lines dedicated to data traffic; it may be possible to cut costs by up to 80 per cent if the five telephone lines can be replaced by a single, multiplexed telephone line.

4.2 Background

Multiprocessing DTEs such as timesharing computers are designed to support literally dozens of remote DTEs (PCs, terminals, and so on). Each remote DTE connection is made through a **port**, an interface to the central computer. Most operating systems allow the various ports to be configured (by either the system manager or the operating system) to specific terminal types (for example, dial in, asynchronous, synchronous).

A typical set of DTE connections in a multiprocessing environment might consist of the configuration shown in Figure 4.1, where three interactive terminals (DTE 1, DTE 2, and DTE 3) are connected by a series of individual asynchronous channels to ports E, L, and S, respectively, on a shared central processor (Host DTE).

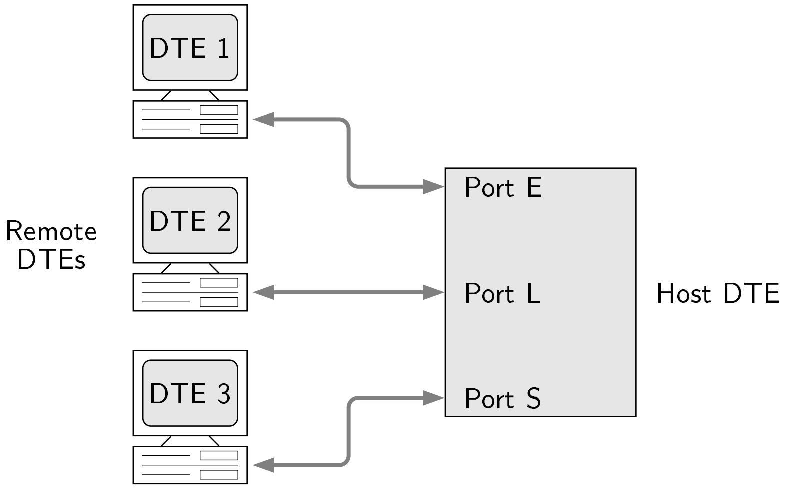


Figure 4.1: Remote DTEs connected to dedicated ports

If the individual channels connecting the interactive terminals to the ports on the host are only partially in use, it may be possible to have some or all of the terminals multiplex their communications onto a single channel; for example, see Figure 4.2.

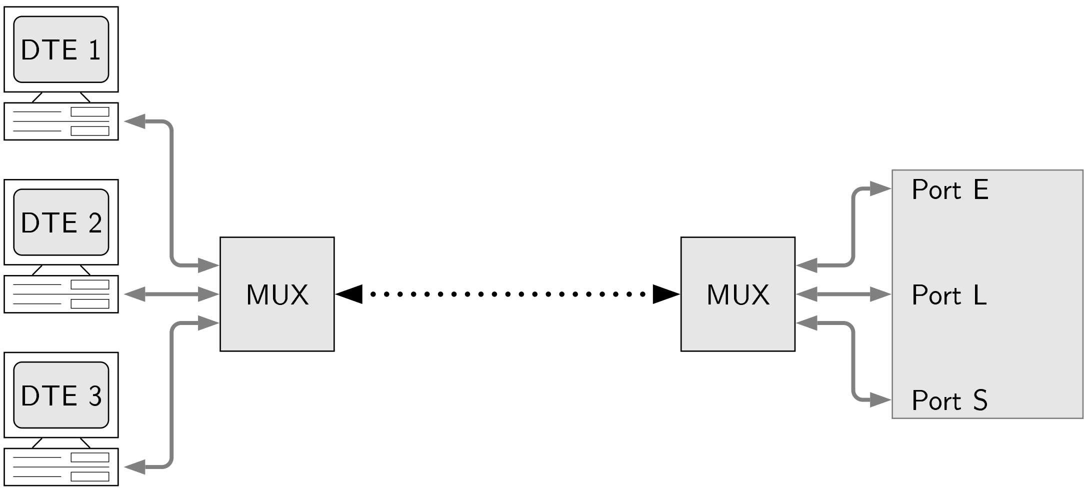


Figure 4.2: Remote DTEs sharing a channel by multiplexing

The equipment controlling the multiplexing is known as a **multiplexer** (or **mux**) and performs two operations:

- the multiplexer takes information from the various DTEs and puts the information onto the multiplexed channel. This is known as **multiplexing**.
- the multiplexer takes information from the multiplexed channel and supplies it to the intended destination DTE. This is known as **demultiplexing**.

The operation of the multiplexer should be transparent to the DTEs which are being multiplexed. In this situation, transparency refers to the effect of the multiplexer on the communications. For example, there should be no detectable difference in communication speeds regardless of whether the multiplex is or is not present. Similarly, the data that is sent should not be altered in any way by the multiplexer.

The multiplexed channel is typically full-duplex, thereby allowing some DTEs to be transmitting information while others are receiving. Having said this, the connections between a multiplexer and its multiplexed DTEs can be any or all of full-duplex, half-duplex, or even simplex. It is also possible to have the multiplexer support a mixture of synchronous and asynchronous DTEs. In some cases, the connection between the DTEs and the multiplexer is asynchronous and the information sent between the multiplexers is synchronous. In short, the characteristics of the multiplexed channel should not have any bearing on the information that is being multiplexed. As with all other communicating devices, for the communications to succeed between a pair of multiplexers, both must agree to a common protocol.

4.2.1 Multiplexer Internals

A multiplexer is another example of the input-processing-output cycle. The multiplexing task consists of:

1. checking a connection for data,
2. formatting the data according to the protocol used by the channel, and
3. transmitting the data on the channel.

The demultiplexing algorithm is somewhat similar:

1. read the data from the channel,
2. determine the destination of the data, and
3. forward the data to the specific connection.

Physically, the multiplexer consists of two parts: connections to external devices and a connection to the remote multiplexer. The connections used depend upon the distance from the multiplexer to the external devices and the remote multiplexer; meaning that multiplexers usually support both direct connections and modems.

4.3 Multiplexing Techniques

The multiplexer is responsible for ensuring that the information supplied by one DTE arrives at the correct destination DTE. There are three multiplexing techniques: **frequency division multiplexing**, **time division multiplexing**, and **statistical multiplexing**.

4.3.1 Frequency Division Multiplexing

Frequency division multiplexing or **FDM**, involves dividing the multiplexed channel into a number of unique frequencies, with each frequency assigned to a pair of communicating entities. FDM can be achieved only if the available bandwidth on the multiplexed channel exceeds the bandwidth needs of all the communicating entities.

Whenever a multiplexer receives data for transmission, the data is transmitted by the multiplexer on the frequency allocated to the transmitting entity. The receiving multiplexer forwards the information received on a specific frequency to the destination associated with that frequency.

The following example illustrates how a frequency division multiplexer could connect DTEs 1, 2, and 3, with ports E, L, and S on a central host, respectively. The frequency allocation is given in Table 4.1.

Table 4.1: Frequency allocation for Figure 4.3

DTE-port pair	Frequency (Hz)
1 and E	10000 – 14000
2 and L	5000 – 9000
3 and S	0 – 4000

The 1000 Hz separation between the channels is known as the **guard band** and is used to ensure that one set of signals do not interfere with another. Diagrammatically, the connections and their frequencies are shown in Figure 4.3.

The advantage of FDM is that each DTE is assigned a unique frequency that can be treated as an unshared channel. However, FDM is not widely used in data communications because of the costs involved in having hardware that can transmit and receive signals on a variety of frequencies. An everyday example of FDM is cable television, in which many signals are 'stacked up' and transmitted simultaneously over the cable. The user selects a viewing channel by tuning to that channel's frequency.

4.3.2 Time Division Multiplexing

Time division multiplexing or **TDM**, requires the multiplexer to timeshare the channel between the various DTEs involved in the communication. That is, at a specific moment, the remote multiplexer will send a byte from, say, DTE 1; at the next instance a byte from DTE 2 will be sent, and so on until all DTEs have been polled; the cycle is then repeated.

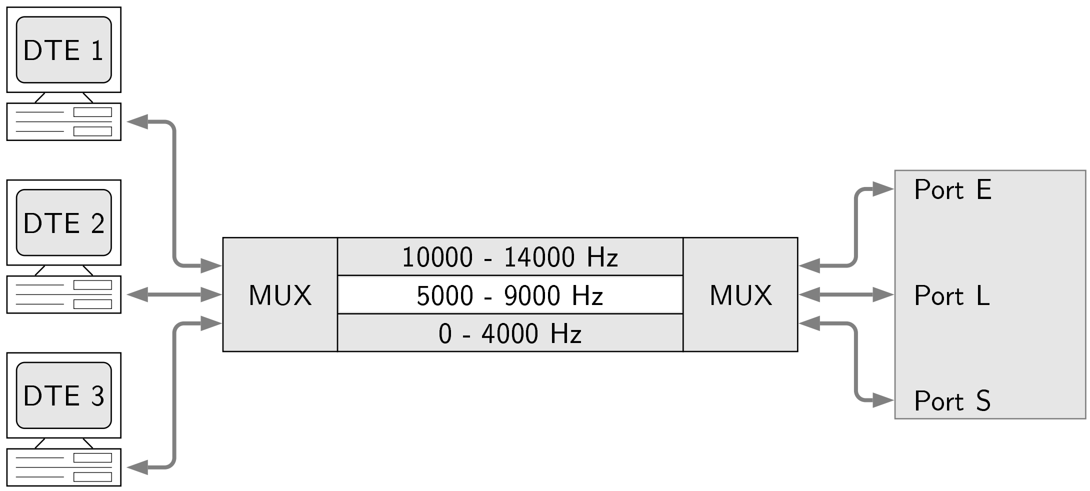


Figure 4.3: Frequency-division multiplexing

In Figure 4.4, DTEs 1, 2, and 3 are in communication with ports E, L, and S, respectively. The multiplexed channel is full duplex and shared amongst all the communicating DTEs (the identifier in each box in the multiplexed channel between the two multiplexers should be read as "data from" the specific port or DTE; that is, it contains data rather than the identifier).

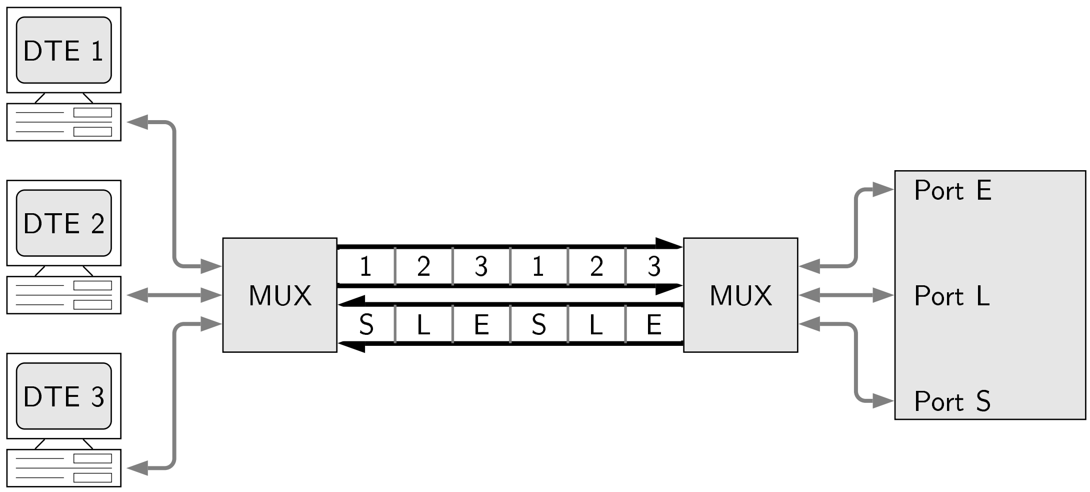


Figure 4.4: Time-division multiplexing

In Figure 4.4, the topmost channel contains data from DTEs 1, 2, and 3. The rightmost multiplexer is about to receive a byte from DTE 3, this is followed by a byte from DTE 2. On the lower channel, data is sent from ports E, L, and S; the leftmost multiplexer is about to receive a byte from port S.

To avoid the situation in which information arrives at the wrong DTE, both multiplexers must be synchronized. That is, the bytes must be sent in an agreed upon order and each byte that is received must be for the specific DTE (or port) to which it was intended. Synchronization can be achieved in a number of ways. A common approach is to use a special bit pattern that is used to indicate the start of a new cycle; for example, if N DTEs are being multiplexed, the $(N+1)^{th}$ byte to be transmitted would be a special, synchronizing bit pattern.

A time division multiplexer polls each DTE to determine if there is information to be sent. However, a problem arises if one of the DTEs has nothing to send: what does the multiplexer transmit? Something *must* be sent because not sending a byte will mean that the time allotted to one DTE may be used by another, potentially resulting in the information arriving at the wrong destination. For example, if DTE 2 has nothing to send to port L, data from DTE 1 might be sent to port L instead.

This problem can be overcome in several ways, including:

- having a reserved bit pattern (for example, the *NUL* character), that is sent whenever a DTE (or port) has nothing available for transmission.
- transmitting *nine* bits between the multiplexers, eight bits for data, and the ninth to signal whether the byte contains data or is empty.

4.3.3 Statistical Multiplexing

Statistical multiplexing attempts to overcome the problem of idling DTEs by sending only information from a DTE when information is available. For example, if three DTEs are sharing a channel and only one DTE is active, then as much of the channel as possible should be given over to the active DTE. (In the case of the time division multiplexing example in the previous section, two of the three time-slots would be empty because of the inactive DTEs.)

Since the multiplexers must be able to determine the intended destination of each byte that is received, an identifier uniquely identifying either the source (i.e., the sending port or DTE) or the destination (i.e., the receiving port or DTE) of the byte, must be included with each byte. This has two implications:

- the bandwidth is reduced because the identifier is sent with each byte. If the identifier is a byte (a reasonable choice since the multiplexed channel is probably byte-oriented), the bandwidth would be halved.
- the multiplexer must know the destination of the byte. Each byte must be transmitted with an identifier which can indicate either source of the byte or the intended destination of the byte.

  If the identifier is the source's address, the remote multiplexer must map the source address into a destination connection. Similarly, if the identifier is a destination address, the local multiplexer must map the local device's identifier into the destination address. Either of these approaches can be achieved through the use of mapping tables.

In addition to the reasons already noted, statistical multiplexing differs from time division multiplexing in that if none of the DTEs are active, the multiplexed channel will be idle.

Figure 4.5 shows how a pair of statistical multiplexers can function. As bytes are sent, they are prefixed with an identifier that allows the receiving multiplexer to determine the destination of the byte. In this example, each data byte is sent as two bytes: the address of the destination (displayed in **bold**) and the data:

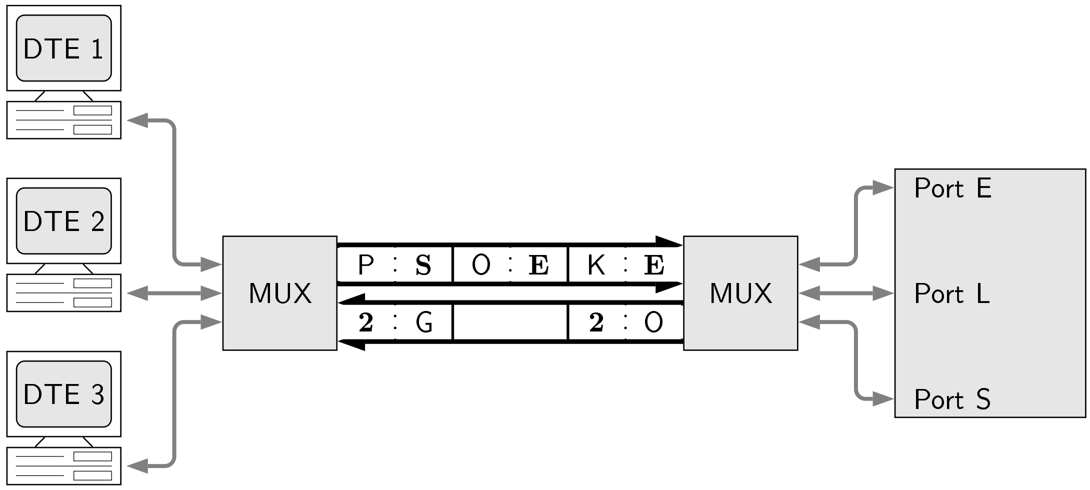


Figure 4.5: Statistical multiplexing

In Figure 4.5, the topmost channel (flowing from left to right) contains two bytes destined for port **E** (K followed by O) and one byte for port **S** (a P). The lower channel has two bytes for DTE 2 (G followed by O); additionally, there is a period during which nothing is being transmitted on the channel.

If all devices connected to a multiplexer transmit simultaneously for a sustained period, there may not be sufficient bandwidth to handle all of the traffic. To handle these situations, the multiplexers may simply discard the additional input or buffers may be used to hold the data until transmission can take place.

4.4 Port Selectors

In most multiprocessing systems, there are more users than there are possible connections to the central computer; and in some cases, there may be more remote DTEs than there are ports to the computer. From the system manager's point of view this is quite reasonable, since few users want to be on the computer twenty-four hours a day (there are exceptions), and the manufacturer of the computer may place a limit on the maximum number of ports that the computer can support.

In those situations where there are more remote DTEs than there are ports on the central host, a device known as a **port selector**, (or **front end**, or **switch**, or **terminal concentrator**), is employed to manage the connections from the remote DTEs to host's ports. The port selector supports two sets of connections: those to the remote DTEs and those to the host's ports. The port selector is typically connected to *all* possible remote DTEs and to all ports on the central host.

When a user on a remote DTE wants to initiate a communicate with the central host, some form of signal is sent by the user to the port selector; for example, a series of characters (typically one or more carriage returns), or a break indication, or the detection of a carrier signal generated when the DTE is powered on. If a free port on the central host is available, the port selector makes a *logical* connection between the remote DTE and the free port. Thereafter, all communications between the remote DTE and the host's port are handled by the port selector mapping the data from the DTE to the port or vice versa. In Figure 4.6, a central host has 2 ports (A and B), and there are four possible remote terminals (DTE 1 through DTE 4); the port selector has mapped DTE 1 to port A and DTE 3 to port B.

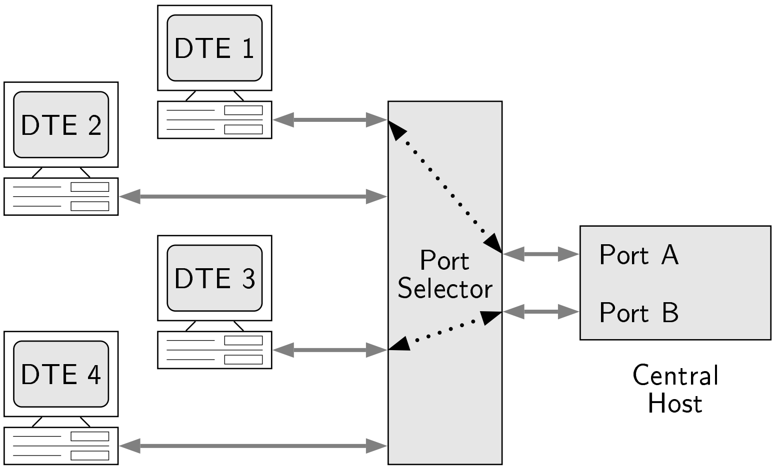


Figure 4.6: A port selector

When the communication finishes, the logical connection is broken and another DTE can use the port. Over a period of time, a DTE might be involved in any number of communications with the central host; however, each communication may be using a different port. If all ports are in use, the port selector ignores the incoming requests (possibly issuing a diagnostic message to the user at the remote DTE).

Port selectors can offer multiplexing capabilities as well. That is, one or more remote DTEs might share a single channel from a remote site; the port selector would demultiplex the channel and make logical connections to free ports using the same techniques as for the directly connected remote DTEs.

4.4.1 Other Multiplexer Applications

Until now the examples of multiplexing showed one multiplexer connected to a group of remote DTEs, while the second multiplexer is connected to a central host. In some situations it may be necessary to have the several (rather than one) central host. For example, in Figure 4.7, a pair of multiplexers support two central hosts and several remote DTEs.

The connections between the various DTEs can be:

Static. If the connections are static, the multiplexers 'know' the destination of each byte that is sent. This can be implemented using statically allocated mapping tables, meaning that, for

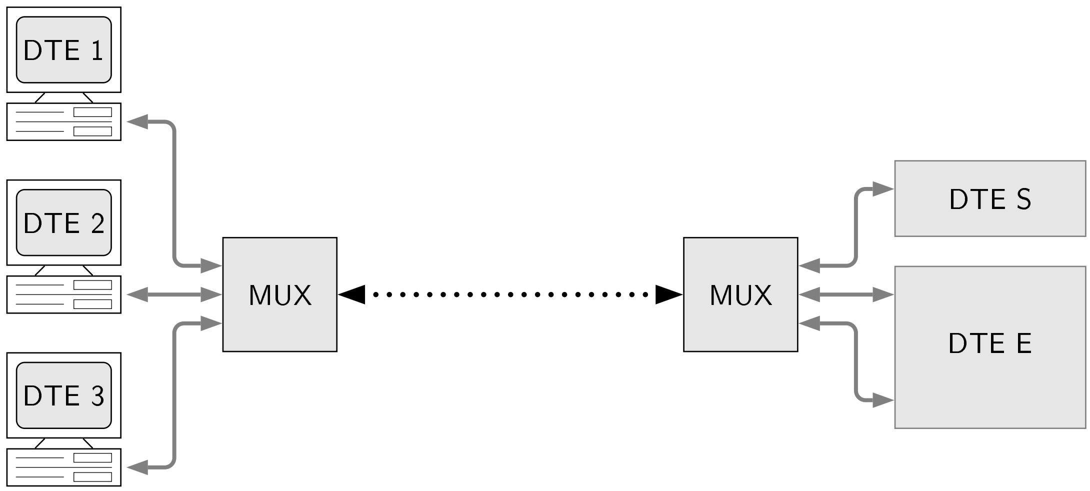


Figure 4.7: Multiplexing with multiple central hosts

example, each byte from DTE 2 is always sent to DTE S (and vice versa).

Dynamic. Static connections force users to 'seek-out' terminals that connect to specific DTEs. By making the connections dynamic using some form of a port selector, users can choose their intended destination DTE.

When the remote user is allowed to select one of several possible central DTEs, it is necessary for the user to indicate the required destination to one of the multiplexers. Once the destination is known, the information can be used by the multiplexers to establish a path from the user's DTE to the central DTE. If dynamic connections are allowed, the multiplexers may be required to support a protocol whereby the connection information (i.e., the source and destination addresses of the devices) is exchanged.

4.5 Summary

This Chapter has examined another method of reducing line charges, notably the sharing of a channel amongst several DTEs or processes through the use of a multiplexer. The basic function of a multiplexer is to accept data from a number of devices and transmit it on a single, shared channel, while simultaneously receiving data from the channel and routing the information to the correct destination device.

There are a number of different multiplexing techniques:

Frequency Division. Frequency division multiplexing requires each pair of communicating devices to be given their own frequency on the multiplexed channel. The multiplexers put data onto the channel using the frequency associated with the devices. The frequency at which data is removed indicates the destination of the data.

Time Division. In time division multiplexing, data from each device is sent at a precise time interval on the multiplexed channel. If the device has nothing to send, a signal is sent indicating that at this moment, the device associated with this slot has nothing to send. The receiving multiplexer is synchronized with the transmitter to ensure that the received data is supplied to the correct destination device.

Statistical. Statistical multiplexing attempts to reduce the amount of wasted bandwidth associated with time division multiplexing by transmitting information only when a device has information to send. To allow the receiving multiplexer determine the intended destination of the data, an identifier is sent with every data byte.

4.6 Exercises

1. (A) Describe how device polling works. Compare device polling with multiplexing.
2. (A) Two approaches to handling idle connections were described for time division multiplexing. Describe the advantages and disadvantages of each.

Chapter 5

Bus Local Area Networks

5.1 Introduction

A bus local area network is a network in which all computing devices (or **stations**) share a single common communications channel. Stations communicate by passing messages in **packets** across the network. To distinguish between the different stations, each station on the bus is assigned a unique address. In Figure 5.1, a bus network interconnects five stations.

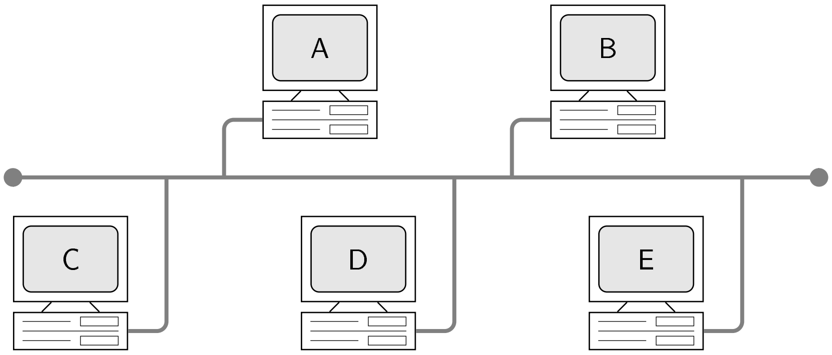


Figure 5.1: A bus network of five stations

A typical transmission scenario involves one station transmitting a packet (consisting of at least a message *and* a destination address) one bit-at-a-time onto the network. The bits propagate past all stations; the stations are responsible for reassembling the bits into packets. The packet's destination address is examined by each station: the station with the address that matches the destination address in the packet keeps a copy of the packet. Those stations with addresses that do not match the packet's destination address ignore (discard) their copy of the packet. A bus network is also known as a **broadcast** network since each packet that is transmitted is received by all stations on the network. Packets are not physically removed from the network by any of the stations; instead, each end of the bus has a **terminator** which, using resistors, electrically removes the packet.

The bus is a resource that is shared by all stations on the network, and as such, must support mechanisms which:

- prevent one station from monopolizing the network by, for example, transmitting a packet of a size that prevents other stations from transmitting messages, and
- resolve conflicts should two (or more) stations attempt to access the network simultaneously.

The first of these mechanisms (preventing a station from monopolizing the network) is usually achieved by restricting the maximum size of the packet and enforcing this limit through hardware. However, controlling network access is a somewhat more complex issue and has resulted in a number of solutions.

5.1.1 Controlling Network Access

Controlling access to a bus network is analogous to controlling a group of people at a meeting, all of whom wish to speak. A number of different approaches are possible, including:

- Have one person 'chair' the meeting. The 'chair' is responsible for determining when a person can speak. Once a person has spoken (or signalled the chair that there is nothing to be said), the chair can ask another person to speak.

  In a bus network, one station can be given the responsibility of controlling network access. This station polls the remaining stations on the network. If a polled station has data to send, the data will be transmitted to the intended destination, otherwise nothing will be sent. Once the transmission has completed or the controlling device has determined that the polled station has nothing to send (typically through some form of acknowledgement), the controlling device will poll another station on the network. The cycle of polling and transmitting will continue indefinitely, permitting each station access to the network.

  There are a number of limitations associated with using a single device to control access to the network. First, should the controlling device fail, none of the other stations could then access the network since the polling cycle has stopped. Second, although polling ensures that all stations have equal access to the network, it can result in poor throughput when one station has a large amount of data to send and the other stations have none (since each station must be polled, regardless of the amount of data to be sent). Third, a change in the status of the network (such as an existing station leaving the network or a new station joining), would require a mechanism allowing the polling device to determine that a change had occurred.

- An alternative to having a person act as the 'chair' is to allocate each person a limited time to speak. For example, each person could be permitted to speak for one minute, the first 'on-the-hour', the second at 'one-minute-past', the third at 'two-minutes-past', and so on. The cycle could be repeated once the last speaker has finished. A clock that could be seen by all possible speakers would be essential.

  A similar approach could be taken to control access to the network, with the exception that each station could only transmit during specific time periods, as dictated by a global clock (as opposed to a centralized device indicating which station could transmit). The underlying assumption is that all stations could synchronize their clocks. For example, at clock tick '1', station '1' would transmit, at clock tick '2', station '2' would transmit, and so on, until all stations had transmitted, at which point the cycle would repeat.

Many of the problems associated with device polling also occur when using synchronized clocks. For example, should a clock on one of the stations fail, the station may start to transmit out of turn; similarly, should one station have a large number of messages to transmit, while others do not, the network would remain idle unnecessarily.

- If the money were available, each person could be given a radio that could be tuned to different frequencies. By assigning each person a frequency on which to transmit, anyone could transmit whenever they felt the need, and those people interested in hearing what someone else had to say could tune into that person's frequency.

  Assigning each station a unique frequency band ensures that all stations have simultaneous access to the network. Protocols, such as those used in wide area networks, are required to permit the initial establishment of the call between two stations (using one frequency), and then proceeding with the call (possibly on another frequency).

  A number of networks have been implemented using this approach, but this type of network is costly since it requires that each station accessing the network use hardware that can switch between a number of different frequencies. Networks which support multiple frequencies (and multiple simultaneous transmissions) are known as **broadband** networks, while networks supporting a single frequency (and hence one transmitting station at a time) are known as **baseband** networks.

Although each of the above networks have been implemented in both commercial and experimental situations, the underlying requirement for having some form of centralized control (in the first two networks) and the costs involved (in the third), do not make them attractive as bus local area networks.

An alternate approach is to resolve the conflict in much the same way humans do should two (or more) people attempt to talk simultaneously: everyone stops and (ideally) one person is then allowed to proceed. In the case of humans, if a person is talking and simultaneously hears someone else talking, both should stop in order to determine who should proceed. This is resolved in a number of ways (the bigger person can win, the person with the louder voice, or more fairly, the person who has remained silent for the longest is allowed to talk first).

A similar algorithm can be applied to stations on the bus network: any station can transmit whenever it has a packet to send and the network is **idle** (that is, the network is not already in use: an idle network is indicated by the absence of a **carrier** signal). Should two stations simultaneously detect that the network is idle, they can both start transmitting: which will result in a **collision**. Collisions can be detected by the station transmitting its packet and simultaneously 'listening' to the channel; if the same data is received as is transmitted, there has not been a collision. However, if the data transmitted is not the same as what is received, a collision has been detected. The transmit-and-listen approach is used by many bus networks and is given the name **Collision Detection** (often abbreviated to **CD**).

A collision, once detected, must be resolved as quickly as possible since no useful information is transmitted during a collision. Collision resolution, whether it be between people talking simultaneously or between stations on a bus network, is not an instantaneous operation. Because of this, there are two main issues to be considered when attempting to minimize the time wasted resolving a collision:

1. how to avoid collisions in the first place, and
2. once a collision occurs, how to determine which station is to transmit next.

Once the network becomes idle, one or more stations may be waiting to transmit their messages. If all of the waiting stations transmit simultaneously (assuming that they all have detected the network being idle), their packets will become garbled, resulting in a collision. In an effort to minimize the possibility of collisions when the network becomes idle, while also attempting to diminish the length of time the network is idle, there are three classes of algorithm used to determine a station's action when the network is *busy* and the station has a packet to transmit:

Non-persistent. The station senses the network: if the network is busy, the station waits a random time period before sensing the network again. The assumption is that it is unlikely that two stations will wait the same random period, with the result that when the network becomes idle, only one of the stations will detect the idle state and start to transmit. However, the non-persistent algorithm can result in wasted bandwidth if a transmission stops and the network returns to the idle state while the stations with packets to transmit continue waiting for their random time period to expire; see Figure 5.2.

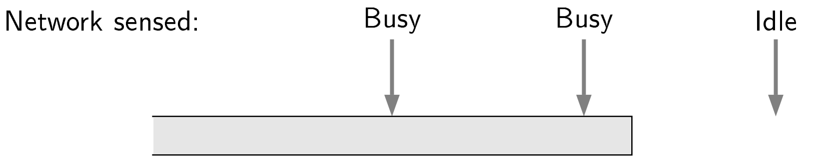


Figure 5.2: Sensing with the non-persistent algorithm can miss 'idle' periods

In the example shown in Figure 5.2, when the station samples the network, the network is found to be busy, resulting in the station waiting for a certain period of time. Meanwhile, the network becomes idle and remains so until the station senses the network again, at which point the network is found to be idle and the station can transmit.

1-persistent. A station with a message to transmit senses that the network is busy and continues to sense the network until it is idle, at which point the station starts to transmit its message. 1-persistent attempts to ensure that the idle time on the network is as short as possible.

p**-persistent.** As with the 1-persistent algorithm, the p-persistent algorithm requires that the station continue sensing the network until it becomes idle. Once idle, the station will transmit with a probability of p, otherwise it waits a predetermined length of time before sensing the line again. The p-persistent algorithm attempts to minimize the network's idle time while at the same time trying to minimize the chances of stations transmitting simultaneously.

Although all of the above algorithms require that the station sense the bus before transmitting, collisions can still occur should one or more stations detect an idle network and start to transmit.

A common form of collision resolution is to have each station 'backoff' for a random period of time before attempting to transmit again (somewhat like the non-persistent algorithm).

Bus networks that require their stations to sense the network before transmitting are commonly known as **carrier sense multiple access** (or **CSMA**) networks. The abbreviations CSMA and CD are usually combined to CSMA/CD when referring to networks that are carrier sensed, support multiple access and collision detection. There are a number of features that make CSMA/CD bus networks attractive as local area networks, including:

- the bus is **passive**. This means that control information such as polling packets are not present on the network: the only network traffic are the data packets sent between stations; and stations are required only to transmit their own packets.
- control on the bus is **distributed** amongst all the stations, meaning that the failure of one station does not mean that all communications cease.

5.2 Examples

At present, there are a number of bus local area networks that support CSMA/CD, two of which are discussed in this section: the **Ethernet** and the **IEEE 802.3 bus**. An example of a typical bus network controller (designed for both the Ethernet and the 802.3 bus and installed in many PCs) is described at the end of this section.

5.2.1 The Ethernet

Probably the best known CSMA/CD bus network is the Ethernet, a 1-persistent baseband network developed by Xerox in the mid-1970s. The Ethernet is based upon the research performed on the Aloha system at the University of Hawaii: a broadcast network consisting of various stations scattered throughout Hawaii communicating by radio with the University's central computer facilities.

The Ethernet has the following characteristics:

- a maximum of 1024 stations can be connected over a distance of up to 2.5 kilometres,
- signals are Manchester (phase) encoded (see section 3.4),
- the bus is a coaxial cable, and
- a data rate of 10 Mbits per second is supported.

Physical Organization

An Ethernet consists of a cable interconnecting a series of stations (devices which contain processors such as PCs, workstations, printers, or disks). Stations connect to the Ethernet by means of a **transceiver** cable attached to a **tap**; the tap makes the necessary connections between the transceiver cable and the network. In its simplest form, the Ethernet consists of a single **segment**; however, more complex layouts consisting of multiple segments are possible, as illustrated in Figure 5.3.

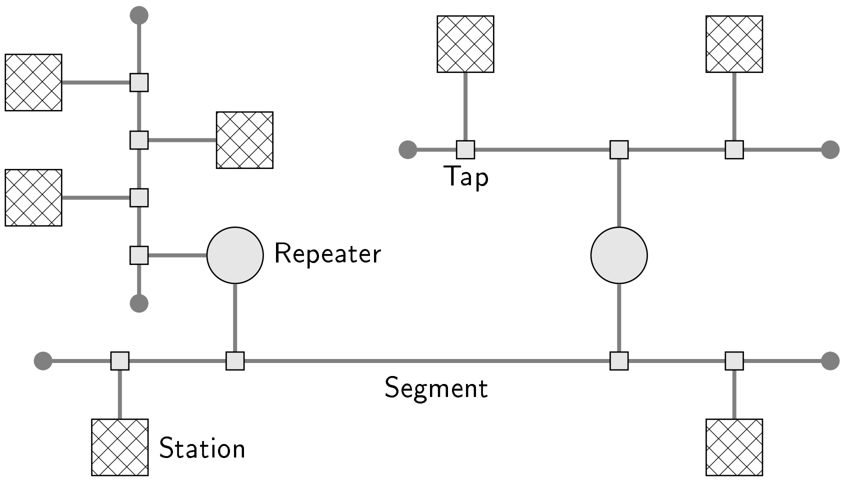


Figure 5.3: An Ethernet with multiple segments

The transceiver cable allows stations to be placed up to 50 metres away from the Ethernet. To avoid interference between station taps, stations must be a minimum of 2.5 metres apart (hence the maximum distance of 2.5 kilometres for 1024 stations). The maximum recommended segment length is 500 metres. The 50 ohm coaxial cable has a diameter of 0.4 inches.

The Ethernet topology is considered a loop-less 'unrooted tree', which means that a packet traverses the entire tree and that individual branches cannot be used independently.

The Ethernet Packet

All data is sent across the Ethernet in an **Ethernet packet**. The start-of-packet is indicated by a synchronizing pattern of 64-bits of alternating 1s and 0s, ending with two consecutive 1s. All bits are Manchester encoded. The packet structure, shown in Figure 5.4, consists of five fields (the numbers in parenthesis indicate the size of a field, expressed in bytes).

Preamble	DST (6)	SRC (6)	Type (2)	User Data (46 to 1500)	FCS (4)

Figure 5.4: The Ethernet packet structure

The fields in an Ethernet packet are defined as follows:

The DST and SRC Fields.
The fields DST and SRC are the destination (i.e., the station for which the packet is intended) and source (i.e., the station sending the packet) addresses of the packet, respectively. All stations on the network are assigned a globally unique 48-bit station address (supplied by

Xerox, thereby ensuring the uniqueness of the address). This address is inserted into the Ethernet packet as the source address (SRC).

The destination address (DST) is also 48-bits long and can be one of:

- a **unicast** address, identifying a single destination on the network. This address is unique to the station. A packet sent with a unicast address is discarded by all stations except the one with the address matching that in the DST field.

  The source address (SRC) is a unicast address.
- a **broadcast** address, identifying all stations on the network. These stations share the broadcast address. A packet sent with a broadcast address is kept by all stations.
- a **multicast** address, identifying a group of stations on the network. The number of stations sharing a multicast address is typically determined by the application and can vary over time. A packet sent with a multicast address is kept by those stations sharing the address.

Table 5.1 shows how addresses are distinguished by their high-order bit.

Table 5.1: Ethernet address structures

Identifier	High-order Bit	Remaining 47 Bits
Unicast	Zero	Zeros and ones
Broadcast	One	All ones
Multicast	One	Zeros and ones

All commercially available Ethernet hardware supports at least the station's unicast address and the broadcast address.

Multicast addresses are also supported to a limited degree by some Ethernet hardware:

- Digital (now Compaq) made the DEUNA (Digital Equipment UNIBUS Network Adaptor) hardware that could identify up to 10 distinct multicast addresses.
- Intel's 82586 and National Semiconductor's 83000 can filter up to 64 multicast addresses; however, the multicast address recognition algorithm used in these chips can result in multicast addresses being accepted by the hardware that are not supported by the station.
- 3COM's 3C5x9 series will either accept all packets with multicast addresses or ignore them. The 3C5x9 allows the receiving software access to the frame on a byte-by-byte basis, meaning that the destination address can be inspected; if the address is not supported, the entire frame can be rejected.

Finally, some Ethernet hardware permits stations to enter **promiscuous mode**, whereby all packets on the network, regardless of the value of the destination address, are made available to the station.

The Type Field.

By itself, the Ethernet offers only a point-to-point datagram service between *stations*; the destination address offers no indication of the intended destination process, application, or protocol. The two-byte Type field overcomes this limitation by allowing the Ethernet software to route the User Data to a specific higher-layer protocol, application, or process. For example, a station may support two services, each associated with its own Type value. An incoming packet would be routed to the service indicated by the value of the Type field.

The User Data **Field.**

The User Data field contains the data sent from one station to the station(s) specified by the destination address. The User Data field cannot contain less than 46 bytes, nor more than 1500 bytes. All data is transmitted transparently; that is, all possible byte values are considered valid.

The minimum packet size has been chosen to ensure that all collisions will be detected, regardless of the length of the Ethernet (i.e., up to its physical maximum of 2.5 kilometres). For example, in Figure 5.5, stations A and B each transmit a short packet simultaneously, the collision is undetected by the transmitting stations because the packets are too small.

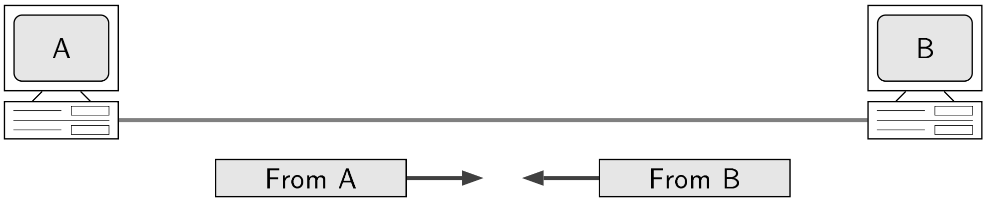


Figure 5.5: A collision cannot be detected since the packets are too small

Similarly, a station will miss the collision if its transmission stops before the station receives a bit from any other packet. This problem is illustrated in Figure 5.6.

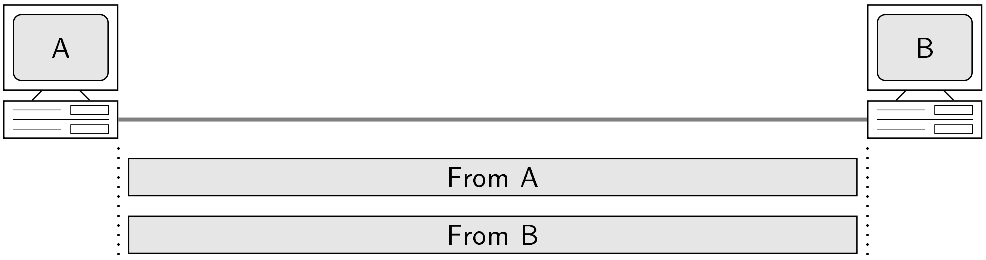


Figure 5.6: A collision cannot be detected since the packets are still too small

From Figure 5.6, one can conclude that a collision will be detected only if a station remains transmitting for the length of time it takes a bit to traverse *twice* the length of the Ethernet. The Ethernet transmits 10^7 bits a second (i.e., 10 megabits a second), or one bit every 10^{-7} of a second. Since the information travels at the speed of light (i.e., 10^8 metres a second), then the 'length' of one bit is 10 metres. Therefore, 500 bits fill twice the maximum length of an Ethernet (5 kilometres or 5000 metres). By rounding the number of bits to 512 (a power of 2), the minimum number of bytes in a packet would be $\frac{512}{8}$ or 64. Four fields within the packet have fixed sizes (the DST, SRC, Type, and the FCS) amounting to 18 bytes, meaning that the minimum size of the User Data field is $64 - 18$ or 46 bytes!

The maximum User Data field length ensures that a transmitting station does not monopolize the Ethernet for an undue period of time. Most Ethernet hardware supports **anti-jabber** features which cut off the transmitter if the packet's length exceeds the maximum number of bytes permitted (i.e., the total number of bytes from all fields).

The FCS Field.
The Frame Check Sequence (FCS) field is a 32-bit CRC value covering all bits in the packet (excluding the Preamble). The FCS transmitted is included in the FCS received by each station, once the Ethernet has gone idle (see below). If the calculated FCS is equal to a predefined value, the packet is assumed to be correct.

The end-of-packet is determined when no traffic is detected on the Ethernet. To ensure that each station's receiving hardware has time to examine the packet and supply it to a higher level of software, there is a minimum packet spacing of 9.6 microseconds.

When collisions occur, the stations involved in the collision are expected to wait random time periods before attempting to send their packets again. The use of a random delay is based upon the assumption that if the stations don't pick the same random value, the collision should be avoided. The Ethernet implements a **truncated binary exponential backoff** algorithm that each station uses if a collision occurs. The algorithm works as follows: each time a collision occurs, the station must wait a time period based upon a random number (see below) multiplied by the length of time it takes for a bit to travel from one end of the network to the other (51.2 μseconds) before attempting to transmit again. The random number is chosen as an integer in the range $0 \leq r \leq 2^k$, where r is the random number and k is the number of attempted transmissions or 10, whichever is less; the growth is **truncated** at $k = 10$ (hence the name of the algorithm).

The station aborts its transmission after 16 collisions.

5.2.2 IEEE 802.3

In the early 1980s, the IEEE set up a Local Area Network standards committee known as **IEEE 802**. Part of the mandate of the 802 committee has been to define data-link and physical layer specifications for local area networks. All networks are uniquely defined in terms of a physical layer and a **MAC** (or **Media-Access Control**) layer. The MAC layer definition overlays the physical layer and part of the data link layer.

One of the networks proposed by the 802 committee is 802.3, a CSMA/CD bus network based upon the Ethernet. The 802.3 standard differs from the Ethernet standard in three main areas: electrical connections, addressing, and the Type field.

802.3 allows both 16 and 48-bit addresses (the 10 Mbps standard requires 48-bit addresses). Unicast, multicast, and broadcast addresses are all supported. IEEE also assigns each station its own unique **global** address, which allows the station to be uniquely identified anywhere in the world.

The 802.3 packet structure is essentially the same as the Ethernet packet structure with the exception of the Type field (all sizes are expressed in bytes), as shown in Figure 5.7.

Preamble	DST (6)	SRC (6)	Length (2)	User Data (46 - 1500)	FCS (4)

Figure 5.7: The 802.3 packet structure

In 802.3, the Type field is replaced by a two-byte Length field. The Length field indicates the length of the User Data field, expressed in bytes. The User Data field can be from 46 to 1500 bytes long, although the Length field can have values less than 46. (If the message is less than 46 bytes long, it must be padded to the minimum length.)

Note that it is possible for both Ethernet and 802.3 packets to co-exist on either network. For example, Ethernet packets can be transmitted on an 802.3 network as long as the value in the Type field does not fall in the range of possible Length values. This is achieved by setting the value of the Type field to anything greater than the maximum Length value (i.e., greater than 1500).

The 802.3 protocol has a number of variants that are expressed in the form xyz, where x is the line speed expressed in megabits per second, y indicates the type of physical medium, and z is the maximum length of a segment, expressed in hundreds of metres. The original 10 megabit baseband 802.3 implementation on thickwire coaxial cable (500 metre segments) is written as **10Base5**. Other variants include **10Base2** (thinwire coaxial cable, more commonly known as **Cheapernet**), **1Base5** (1-megabit over twisted pair), and **10BaseT** (10-megabit over twisted-pair).

The 802.3 CSMA/CD network also implements the truncated binary exponential backoff algorithm to handle collisions.

Other 802 protocols will be discussed further in subsequent Chapters.

5.2.3 Network Interface Cards

Commercially available network interface cards (or NICs) for the Ethernet and 802.3, typically consists of three distinct chips; their relationship is shown in Figure 5.8.

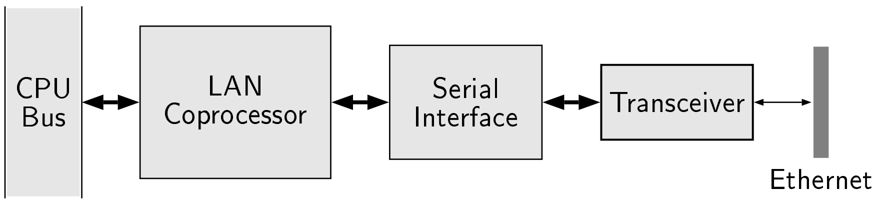


Figure 5.8: Typical 802.3/Ethernet hardware

The **transceiver** chip is responsible for transmitting and receiving the bits on and off the Ethernet. A transceiver such as the Intel 82502 performs additional functions, including:

- monitoring the number of bits the station has transmitted and cutting off access to the Ethernet if the number exceeds the maximum allowed in a packet (this is the anti-jabber function). The station is barred from transmitting for about 420 milliseconds; after which the transceiver again permits access to the network.

- comparing the bits read off the network with those that are being transmitted. Should the bit sent not equal the bit received, a collision has been detected. This information is relayed back to the serial interface.

All signals received by the transceiver are Manchester encoded; these signals are not decoded, but rather passed directly to the **serial interface**. The serial interface, such as the Intel 82501, extracts the clocking signal from the incoming data stream and generates the equivalent internal bit value. The serial interface is also responsible for encoding the outgoing data with a 10 MHz Manchester signal supplied to the transceiver. If a collision is detected (by the transceiver), the condition is forwarded to the **LAN Coprocessor**.

The LAN Coprocessor is a separate processor that implements the Ethernet protocol and performs the following functions:

- calculates the Frame Check Sequence (FCS) for outgoing and incoming frames.
- supports unicast, broadcast, and up to 64 multicast addresses.
- transmits Ethernet packets supplied from the CPU.
- filters and supplies Ethernet packets to the CPU.

In the Ethernet (802.3 **10Base5**; 10 Mbps, baseband, 500m segments) the LAN Coprocessor and Serial Interface can be on the DTE itself, connected to the Transceiver by a transceiver cable. The Transceiver physically taps the LAN. In the Cheapernet (802.3 **10Base2**; 10 Mbps, baseband, 200m segments, and **10BaseT**; 10 Mbps, baseband, twisted-pair), the three chips are typically on a single board with the Transceiver attached to a coaxial cable connector. The LAN connects to the Transceiver via a bayonet or **BNC** T-connector (BNC is an acronym of the device and its inventor: 'Bayonet, Neil-Concelman').

The Intel chipset described above supports the entire 802.3 CSMA/CD Media-Access Control (MAC) functions. There are many other Ethernet/IEEE 802.3 chipsets available from manufacturers, including Advanced Micro Devices (AMD), National Semiconductor, and Seeq Technology.

5.2.4 Programming the 3C509 NIC (Etherlink III)

The level of complexity surrounding the programming a NIC has declined dramatically over the past decade. Early NICs, such as Western Digital's WD8003E, required the programmer to manager the hardware queues and NIC memory. More recent versions, such as 3COM's 3C509, have reduced the programming requirements to that approaching the 8250 UART.

Window Set

All communications with a NIC takes place through one of its seven **windows**. Each window consists of eight 16-bit words; all windows can be read (to obtain status information) or written (to control or program the NIC). All windows are accessed through the NIC's `base_address`.

A window is accessed through the 8086's `in` or `out` instructions. Words `0x00` through `0x0c` are specific to each window, while word `0x0e` is common to all. When written, word `0x0e` contains a command to the NIC, while when read, word `0x0e` contains the window number and the NIC's status.

Initialization

The 3C509 can be configured as a PnP[1] device. This involves the software querying all possible 3C509s and then selecting one to be the "first device", and so forth until all NICs have been configured. The initialization process involves determining the `base_address` of the device (i.e., the device's address in the I/O port memory) and the `irq` (mapping into the NIC's interrupt vector).

As part of the initialization sequence, the software can read the NIC's PROM, which contains the NIC's assigned Ethernet/802.3 physical address. The address can be written to the NIC using one of the NIC's window set (described above).

Transmission and Reception

The NIC can be programmed for packet transmission, packet reception, or both. Transmission involves writing a header, consisting of two 16-bit words to the NIC (through window 1). The header is followed by the packet (destination address, source address, type field, and data bytes); this must be padded to the nearest doubleword. Once the packet is supplied to the NIC, it will attempt to transmit the packet (the NIC can calculate the CRC); upon successful transmission of the packet, an interrupt is generated.

When a packet is received, the NIC signals the Ethernet driver that a packet can be read. The driver reads the contents of the packet from the NIC (one, two, or four bytes at a time), starting with the destination address. If the driver chooses to discard the packet (for example, after reading an unsupported multicast address), it can do so by signalling the NIC.

The NIC can also cause interrupts when it detects a fault, such as 16 attempts at transmitting a packet or when receiving a runt packet. All NIC interrupts can be blocked under program control.

5.3 Summary

This Chapter has examined bus local area networks. Bus networks are named after their topology, in which all stations share a single, common communication channel. There are essentially two problems that the designers of bus networks must overcome: how to control access to the network and how to resolve collisions. Access control is handled by requiring each station to listen to (or sense) the line prior to transmitting. Collision resolution is handled first by having the stations determine that a collision has occurred and then cease their transmissions. The second step in collision resolution is an attempt to minimize the chances of it happening again. A common solution is have each station 'backoff' from transmitting for potentially longer and longer periods by picking random waiting periods. Many bus networks (such as the Ethernet and IEEE 802.3) implement

[1] Plug-and-play, or, more to the point, plug-and-pray.

such an algorithm, known as truncated binary exponential backoff. Much of the communication handling is now performed by VLSI chips.

The Ethernet and IEEE 802.3 networks are often referred to as CSMA/CD or Carrier Sensed Multiple Access with Collision Detection. That is, the network is sensed before transmission and if a collision is detected, transmission is halted.

5.4 Exercises

It is recommended that all of these exercises be done on a private network, separate from your campus network.

1. Design and implement a 3C509 driver that receives packets from any source.
2. Design and implement a 3C509 driver that can transmit a packet to a given address.
3. What happens to all machines on the network when a broadcast packet is transmitted?
4. Modify the 3C509 driver so that it will support a list of multicast addresses. The driver should be programmed to receive multicast packets, rejecting those that are not on its list.

Chapter 6

Ring Local Area Networks

6.1 Introduction

The CSMA/CD bus local area network, for all of its benefits, is by no means the only, nor necessarily the 'best' design for a local area network. In the worst case, it is possible that with the bus local area network, a station may never be able to transmit its data. Consider the following scenarios:

- two stations attempt to transmit simultaneously: a collision occurs. By some coincidence, both stations generate the same backoff values, resulting in a continuous series of collisions. Eventually, the collision limit is reached and the transmission of at least one of the packets is aborted.
- a heavily loaded network consisting of several hundred stations all trying to transmit large volumes of data simultaneously. Initially, all transmissions will result in collisions; however, as backoff delays begin to grow longer, some of the transmissions will be successful. Stations that constantly collide might never obtain an opportunity to send their data.

These situations, albeit extremely unlikely on a lightly loaded bus network, have the potential for seriously degrading a heavily loaded network. The source of the problem can be traced to the **non-deterministic** nature of the bus: access to the bus does not guarantee that a station can transmit.

6.2 The Token Bus

To ensure that every station on a bus network with data to transmit has a guaranteed opportunity to transmit implies some form of **deterministic** access to the network. Deterministic network access requires a certain degree of control that the bus network by itself does not offer. One possible approach is to have a station dedicated to polling all other stations for transmissions, thereby ensuring fair access to the network. However, in Chapter 5 it was shown that centralized polling has its limitations, an important one being the question of how to allow access to the network should the polling station fail.

A variation on polling is to allow each station to transmit in turn, but instead of having a single, centralized polling station, permission to access the network is passed *between* the stations themselves in the form of a **token**. Stations no longer compete for the network, instead they wait for the token, at which point they can transmit. Once the transmission has completed, the transmitting station is responsible for forwarding the token to another station to access the network. This cycle continues indefinitely.

At any time, the packet on the network contains either data or a token. Since stations may not transmit unless they have the token, collisions should be rare or non-existent.

Guaranteeing that deterministic network access using the token method described above means that after any station has had the opportunity to transmit, it must then wait until all other stations have been given the same opportunity. For example, in a network of four stations (A, B, C, and D), once station A has transmitted, it cannot transmit again until stations B, C, and D have been given permission to transmit. The question is: what type of mechanism is required to enforce this ordering?

As an analogy, consider the path of a memo throughout an office. The memo originates from the secretary, who passes it to the person named first on the list. The first person passes the memo to the second, the second to the third, and so on until it reaches the last person on the list, who is responsible for returning it to the secretary. The path the memo has taken could look something like that shown in Figure 6.1.

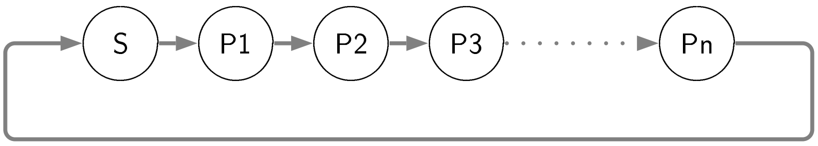


Figure 6.1: Path of a memo through an office

In the case of the memo, the 'last' person on the list is expected to return the memo to the first person (the secretary). Similarly, in the network of four stations, the 'last' station with the token must forward the token to the 'first' station. The path taken by the memo (or the token) is similar to the steps required to traverse a circularly linked list.

The bus network is not a circular list (it is an unrooted tree); however, it is possible, through software, to implement a circular linked list atop the bus. Quite simply, every station is followed by a unique 'downstream' station. Figure 6.2 is an example of how a circularly linked list could be organized on top of a bus network, while Table 6.1 shows the linking of the stations in Figure 6.2.

If each station 'knows' the station that follows it, then forwarding the token is a simple operation as illustrated by the following algorithm:

1. Wait for the permission-to-transmit token from the 'upstream' station.

2. Upon receipt of the permission token, transmit any pending messages that may exist.

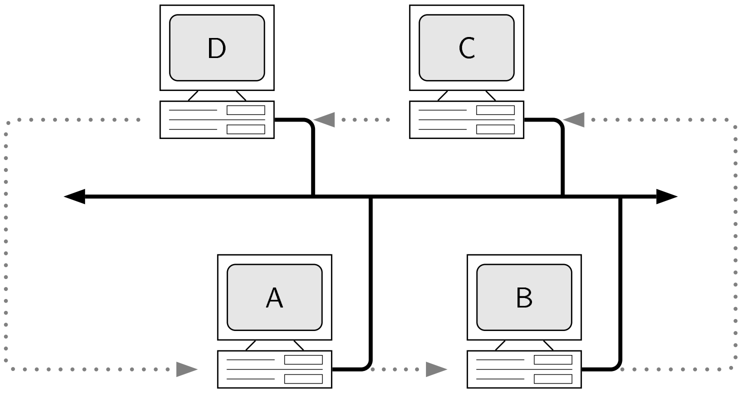


Figure 6.2: Imposing a linked list on a bus network

Table 6.1: Station linkage for Figure 6.2

Station		Downstream
A	⟶	B
B	⟶	C
C	⟶	D
D	⟶	A

3. Forward (i.e., transmit) the permission-to-transmit token to the 'downstream' station.
4. Repeat from step 1.

Access to the network is now deterministic, a station is guaranteed access to the network once all other stations on the network have had an opportunity to transmit. Note that the underlying bus network remains unchanged; it is still a broadcast network (any packet sent on the network is received by all other stations and discarded if the packet's destination address doesn't match that of the station). The bus now supports two types of packet:

Data packets. Packets containing information sent from one station to any other station(s). A data packet is simply a bus network packet carrying data.

Token packets. A packet containing an indication that the station specified in the destination address may access the network. A token packet is a bus network packet carrying token information from one station to another, specific, station.

The use of the token to permit network access ensures that there is an upper limit on the length of time a station must wait before it can transmit. For example, in the worst case (i.e., all stations transmit a data packet when the token is received), a station will have to wait a total time equivalent to:

$$(\text{Number of stations } -1) \times (\text{Time taken to send a data packet})$$

6.2.1 IEEE 802.4: Token Bus

The push for a token bus standard came, in part, from some of the issues that were raised in the Introduction to this Chapter: the bus is nondeterministic, meaning that stations on the network may be required to wait unduly long periods before being able to transmit a frame. The 802 standard committee recognized this and set about to develop another standard: IEEE Standard 802.4, Token Bus. Some of the main proponents of the Token Bus were companies such as General Motors and Boeing (both interested in factory and office automation), who were involved in the development of MAP (Manufacturing Automation Protocol) and TOP (Technical and Office Protocols), respectively.

The 802.4 protocol is *not* built atop the 802.3 CSMA/CD MAC layers. Instead, 802.4 defines its own physical and MAC layers. The physical layer uses a **broadband** backbone coaxial cable and provides three data rates (at separate frequencies): 1 Mbps, 5 Mbps, and 10 Mbps. The separate data rates are intended for equipment that support data, voice, or video. *Physically*, the 802.4 token bus is a broadcast network, *logically*, the stations on the bus are connected as part of a circularly linked-list. The 802.4 frame format is shown in Figure 6.3.

Preamble	Start Delimiter	Frame Control	Destination Address	Source Address	Data	Checksum	End Delimiter

Figure 6.3: The 802.4 frame format

The fields in the 802.4 frame are defined as follows:

Preamble. A clock synchronizing sequence; minimum size of 1 byte.

Start Delimiter and End Delimiter. Marks the start-of-frame (or end-of-frame) by encoding electrically invalid Manchester bit patterns into the byte. The two delimiters are not the same although both are one byte long.

Frame Control. Denotes the use of the current frame: the frame is either a data frame or a control frame. When signalling a data frame, the Frame Control field indicates the frame's priority (see below) and whether an acknowledgement is required by the transmitting station.

Destination and Source Address. The 802.4 frame supports an addressing scheme that is identical to that of the 802.3 CSMA/CD bus. Both 16 and 48-bit addresses are supported, although not simultaneously in the same network.

Data. The message field, up to 8182 bytes (when using 16-bit addresses) or 8174 bytes (when using 48-bit addresses) in length.

Checksum. The Checksum uses the 802.3 frame check sequence algorithm and generator polynomial; the Checksum includes the entire frame (between frame delimiters).

The Token Bus MAC layer is responsible for two main tasks: data transfer and network maintenance:

Data Transfer. When a token arrives at a station, the station has a certain amount of time available in which to transmit messages in 802.4 frames; this time is divided between various message queues. The Token Bus defines four levels of message priority (0, 2, 4, and 6, with 0 being the lowest, and 6 the highest); each level is associated with its own queue. Messages are transmitted from the highest priority queues first (i.e., from 6 down to 0). Once the messages in a queue have been sent, or the queue timer has expired, or if there was nothing in the queue to send, the next lower priority queue is given permission to start its transmissions. If the station has nothing to transmit, or the station has finished transmitting, or the station's time limit has expired, the token is forwarded to the station's successor: a control frame with the control frame field set to TOKEN.

The order of token passing is strictly defined: from high to low addresses. Each station maintains the address of its predecessor and its successor.

Network Maintenance. Network maintenance covers a wide variety of topics: the addition of new stations to the network, the removal of stations from the network, and the initialization of the network. The maintenance of the network is governed by the stations themselves, through the use of control frames. Some of the conditions that 802.4 can handle include:

Lost Token. A token bus cannot function if there is no token on the network. The CLAIM_TOKEN control frame code is used when a station determines that the token is missing and it is necessary to place a new token onto the network; for example:

1. When a station is first brought on-line, it monitors the network for traffic; if none is detected within a fixed period of time, the station assumes that it is the first station on the network. However, before putting a token onto the network, the station first broadcasts a CLAIM_TOKEN frame. If no responses to the frame are received, the station sets up a network of one (itself) and sends frames addressed to itself (thus generating network traffic).
2. All stations contain timers monitoring the network for the token; if a token is not detected within a certain period of time and a station's timer expires, the station broadcasts a CLAIM_TOKEN frame. If collisions occur during the bid for the token, various timers come into play, allowing one station to eventually win the bid. At this point, a new token is placed onto the network.

Adding stations. The physical presence of a station on the network does not necessarily mean that it will receive a token; the station must 'join' the network and become a successor of one station and the predecessor of another.

The SOLICIT_SUCCESSOR_1 control frame is sent periodically by any station that happens to have the token, thereby allowing other stations to participate in communications. In order that the strict highest-to-lowest sequencing of stations is maintained, the frame contains the sender's and successor's addresses; only stations waiting to join the network with addresses in that range are allowed to bid. If no bids are received within a certain period of time, the station with the token proceeds with its data transfer. If a bid is received from one station, it becomes the successor of the current station with the token. Should several stations bid simultaneously, their bids will collide, requiring the use of a RESOLVE_CONTENTION frame to indicate that the bids have failed. The station holding the token then attempts to add a new station by halving the range of possible addresses (i.e., from its address to the mid-point between its address and the address of its successor). This cycle continues until a successor is found.

Stations leaving the network. When a station Q is to leave the network, it sends a SET_SUCCESSOR frame containing the address of Q's successor (say, R) to its predecessor (P). The predecessor, P, makes its new successor R. Q is now removed from the network.

Missing stations. When the station currently holding the token passes the token to its successor, it monitors the bus for subsequent traffic. Ideally, there should be traffic generated by the successor; however, if nothing is heard, the original token holder sends the token again. If no response is heard the second time, the token holder assumes the worst and broadcasts a WHO_FOLLOWS frame, containing the address of the missing station. The missing station's successor is to respond with a SET_SUCCESSOR frame; the token holder updates its successor address and forwards the token to its new successor.

If no responses are forthcoming to the WHO_FOLLOWS, the token-holding station broadcasts a SOLICIT_SUCCESSOR_2 frame. Stations wanting to join the network can bid to become the successor.

One of the reasons behind the topology of the Token Bus is that it lends itself well to automated-factory assembly lines consisting of robots or other computer controlled equipment. The cabling is laid the length of the factory, connecting to the various pieces of equipment.

By completely changing the topology of the network (that is, by abandoning the bus structure), it is possible to achieve results similar to those of the token bus, with a somewhat less complicated protocol and, potentially higher throughput. The topology in question is the **ring**.

6.3 Ring Networks

The topology of a ring network is as the name suggests: the network forms a ring, interconnecting all stations; see Figure 6.4.

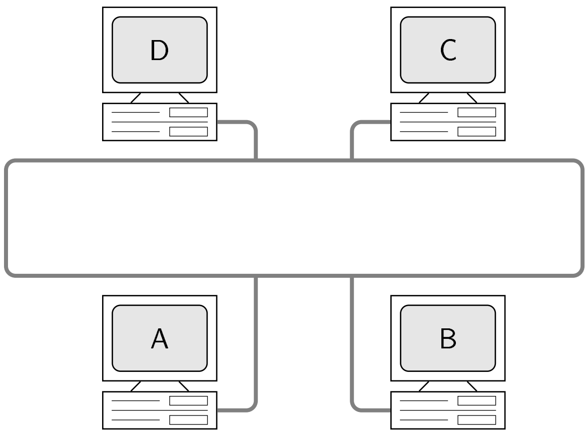


Figure 6.4: The basic structure of a ring network

Accessing a ring is similar to accessing a token bus: the station must wait until permission is granted, at which point, transmission can occur. However, there are several notable differences between the ring and the token bus:

Physical Topology. The token bus is a deterministic ring built atop a non-deterministic bus network; all transmissions are broadcast. The token, as it is passed from station to station, follows no particular direction: the physical location of a station has no bearing upon when it receives the token.

A ring network consists of a number of stations connected in a closed loop. In most rings, packets flow in one direction only; that is, out from the transmitting station, past all the other stations and back to the transmitting station, which is responsible for removing the packet.

Successor Identification. The successor station in a token bus must be explicitly identified in the token packet; whereas in a ring, the successor is the next station 'downstream' from the current station.

Reliability. Most token bus and ring networks are designed to continue functioning should a station fail. However, should the channel fail, the results can be quite different. The bus can, in theory at least, continue functioning in a degraded fashion as two separate buses.

A channel failure on a ring can result in the total cessation of communications since all information flows in one direction around the ring. Recent developments in ring architecture use *two* separate bi-directional channels: should one channel fail, packets can be routed through the other channel.

There are, broadly speaking, three different types of ring network algorithm: the token ring, the slotted ring, and register insertion.

6.4 The Token Ring

Accessing a token ring is similar to accessing a token bus: stations must wait until they receive permission before transmitting a message. The ring itself is in one of two states: **free** or **busy**; depending upon the value of the token in a variable length frame which is continuously circulating around the ring. Frames are made up of bits; at any moment there are a limited number of bits on the ring. The station hardware can sense and change the value of a single bit as it passes the station.

Token ring frames consist of a start-of-frame indication, followed by a control byte that contains a single bit, the token bit, indicating whether the ring is free or busy. The free token frame ends after the control byte with an end-of-frame indication.

A station detecting a free token (i.e., the token bit within the control byte is sensed by the ring physical layer hardware as free) is able to transmit a message. This is done by having the ring hardware first set the token bit within the frame to busy (the bit then continues on). At the end of the control byte, the destination and source addresses are sent. Each bit in the message is then transmitted, followed by a CRC and the end-of-frame indication. The frame circulates around the ring (typically only a few bits are on the ring at any one time), passing each station.

Upon detection of a busy token, each station takes a copy of the frame, assembling the frame out of the sequence of incoming bits. When the end-of-frame is detected, the receiving station can examine the destination address of the packet to determine if the packet should be kept or discarded.

The transmitting station is responsible for removing the bits off the ring (since only a few bits are on the ring at any one moment, if the bits aren't removed, they will interfere with the transmission of the remaining bits). When the transmitting station has completed its transmission, a new free token (consisting of the start-of-frame, control byte, and end-of-frame indication) must be put onto the ring.

The following series of Figures illustrate the cycle of events as station D transmits a message to station B. First, in Figure 6.5, station D waits for a free token.

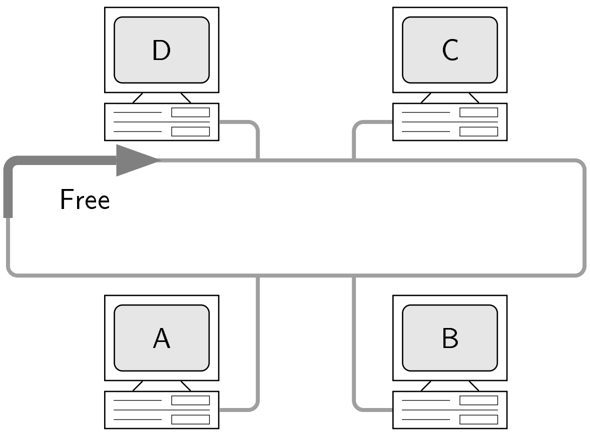


Figure 6.5: Station D awaits the Free token

Once the free token is detected, station D sets the token to Busy and starts to transmit the remainder of the frame (i.e., the destination address, the source address, the message, and any checksum). Note that during part of the transmission, some of the bits of the free token are still being removed from the ring by station D; see Figure 6.6.

The packet circulates past each station on the network; since the token bit indicates Busy, each station proceeds to assemble the remainder of the frame. When the destination address has been assembled, the station compares the frame's destination address with its own: if the destination and the station's address are the same, the station continues to accept the bits making up the frame. Upon detection of the end-of-frame indication, the CRC is checked and the message is forwarded to the station. In this example, only station B takes a copy of the frame since the destination address indicates B; see Figure 6.7.

If the number of bits in the packet exceeds the number of bits that can be circulating at any moment on the ring, the transmitting station (station D in this example) removes the bits at the same time it is transmitting. The transmitting station continues to transmit until the end-of-packet is sent, at which point the transmitting station can apply one of the following algorithms:

- the station can remove the entire packet from the network before reissuing the free token.

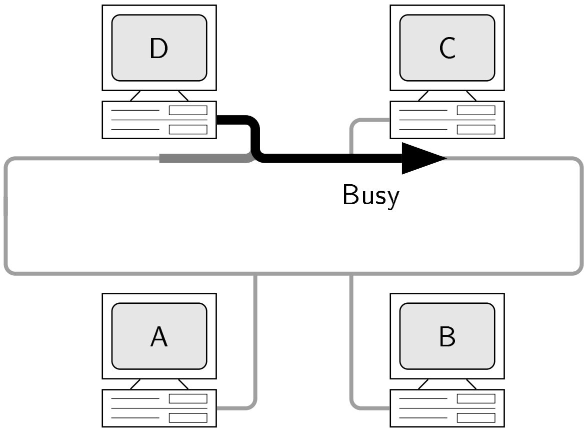


Figure 6.6: Station D changes the token from Free to Busy

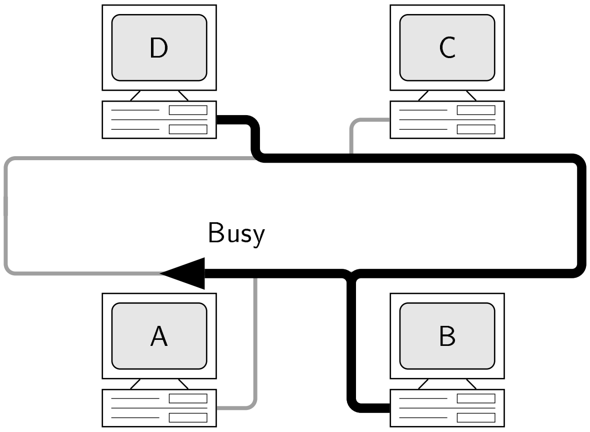


Figure 6.7: Station B keeps a copy of the frame

This is known as **single-frame operation**. Single-frame operation reduces the throughput of the network, since there are periods in which the network is devoid of any frame.

- the station issues a new free token as soon as the busy token is removed from the ring and the end-of-frame has been sent. This method, known as **single-token operation** can be used when the number of bits in the frame are less than the number of bits that can exist on the ring at any time.

- the station can start to transmit a free token as soon as the end-of-frame is sent, implying that several tokens can exist on the ring at any moment; this is known as **multiple-token operation**. This method ensures that the amount of network idle time is kept to a minimum.

In this example, station D issues a free token *before* the original frame is completely removed from the network (i.e., the network supports single-token operation); see Figure 6.8.

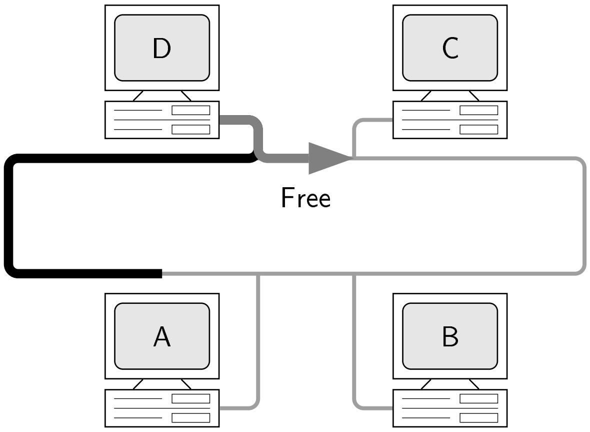


Figure 6.8: Station D is responsible for freeing the token

The above algorithm, which requires the transmitting station to remove the frame as it is being transmitted, is known as **source** removal. The alternate approach is to have the destination station remove the frame as it is received (this is known as **destination** removal). Although destination removal would suggest that greater throughput could be achieved, it does suffer a number of drawbacks, such as the following:

- the deterministic nature of the token ring can be lost. That is, instead of ensuring that each station will be able to transmit within a certain number of frame cycles, one station can continue to transmit without giving up the ring. For example, if station A transmits frames to station B and station B issues a free token, then station A would receive a free token before any downstream stations (i.e., D, C, or B) had a chance to transmit. Station A could therefore monopolize the network.
- support for broadcast and multicast communications would be cumbersome. For example, if the first station to receive a broadcast frame were to release it as a free token, no other stations would receive a copy of the frame. This problem can be overcome by requiring the transmitting station to perform source removal on broadcast and multicast frames only.

  A simpler solution is to use source removal for all transmissions, thereby eliminating these special cases.

6.4.1 Ring Errors

The objective of a ring network is to ensure that all stations have equal access to the ring. This is achieved using some form of circulating token. Should anything happen to the token to stop it from circulating, network access may become impossible. There are two error situations that can stop network access entirely:

Missing Token Situation. If the token is removed from the network or becomes so corrupted (through noise) that it is impossible to recognize, then all stations will be left waiting for a free token that never arrives. This is the **missing token** situation.

In addition to being lost because of noise, the missing token situation can arise if a station with a message to transmit removes the free token from the ring and *crashes* before putting the busy token onto the ring. Similarly, the missing token situation can occur if a frame has been sent and the transmitting station crashes after removing the busy token but before the free token is placed back onto the ring.

Circulating Busy Token Situation. The second error situation occurs when the token remains in the busy state. The token can remain busy either through being corrupted (i.e., changed from free to busy, typically because of noise), or by a transmitting station crashing after setting the token to busy, thereby being unable to put a free token back onto the network.

Should either one of these situations occur, the network will cease to function. To avoid the problem, some form of centralized control is needed. Most rings employ a station to *monitor* the network for either of these conditions.

6.4.2 The Monitor Station

The solution adopted by most ring network designers is to assign one station the task of monitoring the ring to ensure that if the network enters an error state (such as a missing token or the busy token circulating), the error can be corrected and a new token put onto the ring. This station is known as the **monitor station**; its relationship to the ring and the other stations on the ring is shown in Figure 6.9.

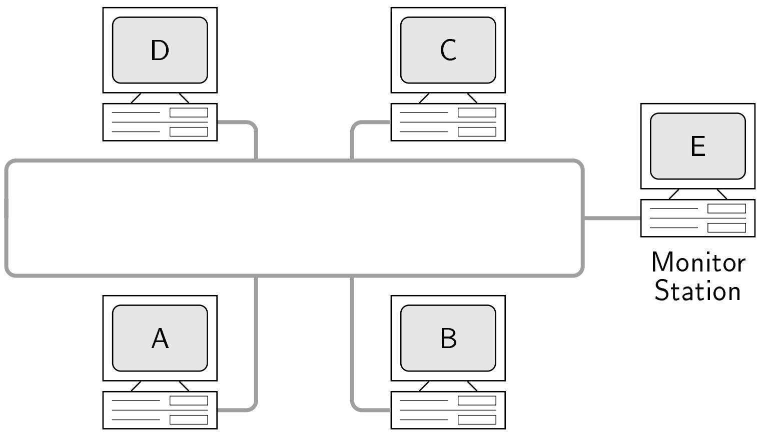


Figure 6.9: The ring monitor

Depending upon the network, the monitor can either be a device dedicated to monitoring the ring, or it can be a station that performs monitoring duties in addition to the transmission and reception of frames.

Regardless of how the monitor is implemented, it must still be able handle the two error situations described above. The monitor can detect the missing token error situation with little difficulty as the network is always being monitored for traffic. Tokens (busy or free) are expected at regular intervals, so that if one does not arrive within the designated period, the monitor can assume the worst and issue a new free token frame.

Detecting the circulating busy token is a somewhat more difficult task, requiring the monitor to determine whether the busy token has just been set to busy or is still busy from a previous cycle around the ring. The approach taken by ring network implementations is to add a bit to the control field which is set by the monitor each time a busy token bit is detected. The bit, often referred to as the **Monitor Passed** bit, signals that a busy token frame was detected by the monitor. The monitor is responsible for setting the Monitor Passed bit when it receives a frame with the token bit set to Busy and the Monitor Passed bit cleared; this is shown in Figure 6.10 (MP denotes 'Monitor Passed'). The transmitting station is responsible for issuing a free token frame with the Monitor Passed bit cleared.

Figure 6.10: The monitor station indicates 'monitor passed' if the token is Busy

There are two conditions that can arise when the monitor detects the busy token: the Monitor Passed bit is either set or cleared:

Set. If the Monitor Passed bit is set, it means that the station transmitting the packet did *not* remove the frame from the network. This is an error condition requiring the monitor to remove all bits from the ring and put a new free token frame onto the ring.

Cleared. If the Monitor Passed bit is clear, it means that this is the first time the packet has passed the monitor. The Monitor Passed bit is then set by the monitor.

Rings can be of varying sizes (i.e., total length or number of stations), and this can determine the number of bits that can be on the ring at any one time. In many rings, the monitor is also responsible for maintaining an **elastic buffer** which, acting like a queue, regulates the number of bits on the ring. For example, a free token may require 24 bits; if the ring has space for only 8 bits, then the elastic buffer must be holding a queue of 16 bits.

6.4.3 Other Ring Errors

Rings that allow at most a single token (free or busy) on the ring at any time (i.e., they operate in single-frame or single-token mode) must be protected from conditions in which more than one token exists on the ring. The **duplicate-token situation** can be caused by noise or some other error, and can result in, for example, two stations attempting to transmit simultaneously.

The solution to the duplicate-token situation requires each transmitting station to examine the source address of the frame *before* removing the frame from the ring. If the frame's source address is not the same as the station's address, then a duplicate-token situation has occurred. Once the duplicate-token situation is detected, there are a number of possible algorithms to handle the error:

- if all stations simply remove frames that do not have the correct source address, and do not place a new free token onto the ring, the monitor station will eventually detect a lost-token situation. Once the lost-token situation is detected, the monitor will reissue a new free token frame.
- the previous solution results in idle periods in which no busy frames can be transmitted until the monitor puts a free-token onto the ring. An alternative solution is to have the stations examine the frame's source address and have the station with the *lower* address value stop its transmission. (This can be achieved by having each station maintain a queue of the incoming bits.) The station with the higher address value continues to transmit its frame.

  The station with the lower address value forwards the bits already in its queue (starting with a start-of-frame indication). Subsequent bits are taken off the ring and stored in the queue for transmission. This ensures that the bits are received in the correct order by the transmitting station.

A second error that must be considered in any ring network that uses a monitor to control network access is: what happens if the monitor fails? Should this occur and an error situation such as the missing token arise, then network access will cease entirely.

Recovery from a monitor failure requires two distinct steps: first, the loss of the monitor must be detected, and second, once detected, the loss must be recovered from. Detection can be achieved in a number of ways, for example:

- the monitor can be required to periodically send 'monitor alive' messages to all stations on the network; if a 'monitor alive' message is not detected within an agreed upon time-limit, the monitor is assumed to have failed.
- since access to the ring is deterministic, each station knows that within a certain maximum time, a free token should be available; if the token is not found, the monitor is deemed to have failed.
- all stations can monitor the ring for traffic; if nothing is detected within a given period, the monitor is assumed to have failed.

Once the error has been detected, the remaining stations must the determine which station is to become the new monitor (assuming that a station can become a new monitor); otherwise the error must be signalled to the person managing the network and the monitor repaired manually.

In a typical recovery algorithm, each station that has detected the loss of the monitor announces this discovery (by means of a frame). As the frame circulates around the ring, each station that has discovered and announced the loss of the monitor examines the source address of the incoming frame. If the source address is *less* than the station's address, the frame is discarded: that is, the extra frame is handled as a duplicate-token situation. The station to receive a packet with its own address as the source address becomes the monitor.

The new monitor issues a new free token onto the ring and the network resumes activity.

6.4.4 Connecting to the Ring

In most ring networks, the ring does not pass physically through the stations connected to the ring. If the ring were to do so, the failure of a station would cause the entire ring to cease functioning. Instead, each station is connected to the ring through a **bypass relay** (shown in Figure 6.11). As long as the station is electrically active, the bypass relay remains open, allowing individual bits to be received by the station. Should the station fail, the bypass relay closes and the bits avoid the station, flowing past it.

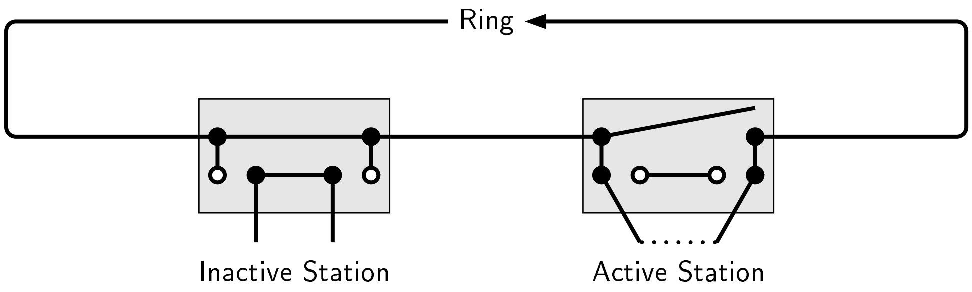


Figure 6.11: The bypass relay

6.5 Examples of Token Rings

The previous section's description of the token ring can be applied to most existing token rings. However, a number of enhancements and extensions found in several existing ring implementations are listed below.

6.5.1 The IBM Token Ring.

The IBM Token Ring functions, for the most part, as the generic token ring described above; however, there are several interesting and notable extensions.

The Busy-Token Frame

The busy-token frame format is illustrated in Figure 6.12 (all sizes are in bytes).

DEL (1)	CTRL(2)	DST (6)	SRC (6)	Data	FCS (4)	DEL (1)	FS (1)

Figure 6.12: The busy frame token format for the IBM Token Ring

The frame delimiters DEL, shown in Table 6.2, are bytes containing invalid Manchester bit encodings and different bit values to distinguish between the starting and ending delimiters (X denotes an invalid bit pattern).

Table 6.2: Encoding of DEL for the IBM Token Ring Frame

Starting delimiter:	0	0	0	X	X	0	X	X
Ending delimiter:	1	1	1	X	X	1	X	X

The control field, CTRL, consists of two bytes. The first byte is common to both the free and busy token frames and consists of the fields shown in Figure 6.13 (the Priority fields are discussed below; Monitor Count is simply the Monitor Passed bit).

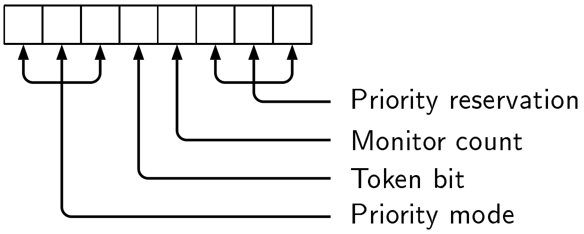


Figure 6.13: The format of the first CTRL byte

The second byte of the control field is transmitted only with busy token frames. The Frame format field indicates whether the data field contains ring signalling information or data link information. If the Frame format field indicates ring signalling information, *all* stations are expected to read the Control field (shown in Figure 6.14).

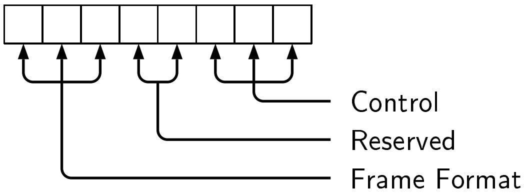


Figure 6.14: The format of the second CTRL byte

The destination and source addresses are stored in the DST and SRC fields respectively. The Data field is variable length, consisting of zero or more bytes. The FCS is a 4-byte CRC and covers the entire frame between starting and ending delimiters.

The free token frame consists of the two delimiters and the first byte of the CTRL field.

Priority Frames

In the generic token ring description, all stations have equal opportunity to transmit any frame to any destination. This can mean that priority frames (such as voice or video) are competing with data frames that could potentially be delayed.

To overcome this limitation, busy-token frames are divided into two modes: those that can be transmitted **asynchronously** (i.e., periodically) and those that need to be transmitted **synchronously** (i.e., on a regular basis). To handle synchronous communications, each busy frame can be associated with one of eight levels of priority, as indicated by the priority bits in the CTRL field (zero is the lowest priority, indicating asynchronous mode).

Normally, the ring functions in asynchronous mode, with all stations having access to the ring. Periodically, when a high priority station requires access to the network, the requested priority can be written into the Priority Reservation bits of a passing busy token. The station freeing the token is expected to copy the Priority Reservation bits into the Priority Mode bits of the new free token *and* maintain a copy of the current priority.

Stations with messages to be transmitted at the level of priority indicated in the Priority Mode bits may now do so, thereby ensuring that higher priority information can be sent.

The priority is returned to its previous value by the station that originally raised the priority. This station is responsible for monitoring the ring, when a free token passes by with the higher priority, the station must change the priority to the original priority.

In the worst case, stations with asynchronous data to be sent can wait forever if all traffic is devoted to synchronous communications.

The Frame Status Byte

The Frame Status byte, FS, follows the final delimiter and is used by the destination station to convey status information back to the transmitting station through two status indication bits A (acknowledgement) and C (frame-copied) (R denotes reserved). Table 6.3 shows the format of the Frame Status byte.

Table 6.3: Format of the Frame Status byte

R	R	C	A	R	R	C	A

The interpretation of the A and C bits are given in Table 6.4.

Table 6.4: Interpretation of the Frame Status bits

A	C	Meaning
0	0	Destination not responding.
1	0	Destination present but frame not copied.
1	1	Destination present and frame copied.

The token ring operates at 4 Mbps.

6.5.2 IEEE 802.5 Token Ring

The IEEE 802 Committee has a token ring network standard: IEEE 802.5, which is compatible with the IBM Token Ring design described above. The addressing scheme follows that of the IEEE 802.3, discussed in the previous Chapter.

6.5.3 FDDI (Fiber Distributed Data Interface)

FDDI is a 100 megabit per second (Mbps) fibre optic token ring standard developed by the American National Standard Institute (ANSI) committee X3T9.5, based upon the IEEE 802.5 token ring standard. An FDDI ring falls into the category of **Metropolitan Area Network** (or **MAN**) since it can span distances of up to 200 kilometres. Due to its high speed, FDDI can also be used as a **backbone** network, interconnecting smaller, low speed, local area networks (such as Ethernets or 802.5 token rings).

Physically, the FDDI network consists of two rings, one transmitting clockwise, the other counter-clockwise. Transmissions can occur on both rings (although the standard recommends that one ring remain in reserve) which gives an FDDI network an effective rate of 200 Mbps. Two classes of stations exist: an A type which connects to both rings, and a B type, which connects to a single ring. Up to 1000 stations can be joined to an FDDI network with a maximum distance of 2 kilometres between class A stations and 500 metres between class B stations. The relationship between class A and class B stations is shown in Figure 6.15.

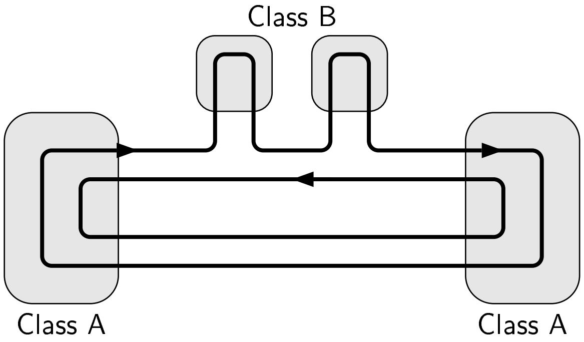


Figure 6.15: The relationship between FDDI Class A and B stations

As in 802.5, stations wait for the free token before transmitting. When a station is in possession of the token, it has a finite amount of time in which to transmit one or more packets. FDDI defines two types of packets: synchronous, or those which a station is guaranteed to transmit; and asynchronous, or those which a station can transmit if there is time available (there are eight levels of priority within asynchronous). Once a station has sent its last packet, it reissues the free token (FDDI operates as a multiple-token ring). The station is responsible for removing its packets from the ring even though it no longer has the token.

One of the arguments for a bidirectional ring is reliability; should both rings be cut for some reason, communications can continue by looping one ring back onto the other in the class A stations nearest the break, as shown in Figure 6.16.

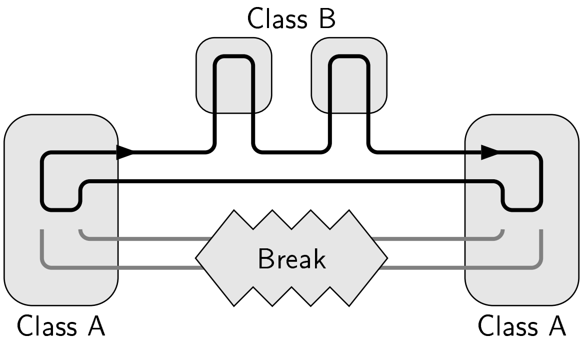


Figure 6.16: The bidirectional ring permits additional reliability

There are presently two FDDI standards: FDDI-1, intended for data traffic; and a new standard, FDDI-2, intended to offer better support for both data and voice.

6.6 Other Ring Networks

In addition to the token ring, there are several other ring structures that have been developed.

6.6.1 The Slotted Ring

A slotted ring is similar to the token ring, with the exception that there is a single circulating frame structure that contains space for data (anywhere from 2 to 32 bytes, depending upon the implementation). The frame has two states: free (indicating that a station can send data in this frame) or busy (meaning that this frame has data and should be read). As with the token ring, when the transmitting station receives its busy token frame, it frees the frame to allow another station access to the network.

Slotted rings typically have several circulating frames to help improve throughput. The number of frames on the ring is controlled by the monitor and must be known by all stations to ensure that a transmitting station frees the correct frame.

Example

Probably the best known slotted ring to achieve any degree of commercial success was the Cambridge Ring, developed in the late 1970s at the University of Cambridge. The format of a Cambridge Ring frame (more commonly known as a **minipacket**) is shown in Figure 6.17 (note, all sizes are in *bits*).

The Start bit, with a value of 1, precedes all other bits in the minipacket and is used by the ring hardware as a synchronization bit to signal the start of the minipacket. The status of the packet then follows and indicates whether the packet is in use (Full, value 1) or available for use (Empty,

Start	Full/Empty	Monitor	DST	SRC	User Data	Type	Response	Parity
(1)	(1)	(1)	(8)	(8)	(16)	(2)	(2)	(1)

Figure 6.17: Cambridge Ring Minipacket Structure

value 0). The Monitor bit is set by the monitor to handle the circulating busy situation; the bit is set by the monitor when a Full token passes the monitor and is cleared by the transmitting station.

The destination, DST, and source, SRC addresses are both 8-bits long. At most 254 stations are allowed on a single ring, addresses `0x00` and `0xFF` are reserved. Two bytes of data are sent in the User Data field, while the Type field indicates the type of data.

The Response bits are sent by the transmitting station with the value `11`; they are to be changed by the destination station and are interpreted as follows:

`11`. The destination has not changed the response bits, implying that the destination does not exist.

`01`. The destination has accepted the minipacket.

`10`. The destination has accepted minipackets from another source.

`00`. The destination is busy and cannot accept the minipacket at this moment.

The monitor station is a separate device; if the monitor fails, the entire ring will shutdown.

6.6.2 Register Insertion Ring

The register insertion ring doesn't really correspond to any of the ring algorithms that have been discussed so far; instead, it functions more as a ring of store-and-forward stations.

Each station in a register insertion ring has two registers, each used to hold a variable length frame: one from the upstream station, and the other from the local station. Frames arrive at a local station as a stream of bits which are copied into the upstream register. When the destination address bits have arrived, the local station compares the address in the register with its own address. If the frame is addressed to the local station, the remainder of the frame is copied to the station, otherwise the frame is forwarded to the downstream station, one bit at a time (the upstream register acts as a queue: with the oldest bits being sent first and the most recent arrivals being stored at the end of the queue).

Before the local station can transmit, the frame must be stored in the station's register. When the upstream register is empty, the station's ring hardware checks the station's register; if there is a frame to be transmitted, the bits in the frame are shifted onto the ring to the downstream station. Should bits arrive from the upstream station, the local station copies the incoming bits into the upstream register while completing the transmission from the station's register.

6.7 Summary

Local area networks allow device sharing amongst all stations on the network. Networks such as the CSMA/CD bus can potentially restrict access to the network (for example, because of exceptionally heavy network traffic) since access is non-deterministic. By changing the topology (either logically or physically) to a ring, it is possible to develop a local area network that permits deterministic access. In other words, there is an upper boundary on the length of time a station must wait before accessing the network.

The solution adopted by both the token bus and the token ring is to permit network access only when the station is in possession of the 'free' token. When a station receives the free token (from its predecessor, either logically or physically upstream from the station), and the station has a message to be sent, a transmission can take place. Once the station has finished transmitting, a new free token is placed onto the network, thereby permitting network access to the downstream stations.

The token bus differs from the token ring in a number of areas; the most obvious being the topology: one is a bus, the other is a ring. Another difference involves network error recovery methods: in the token bus, recovery from network errors is distributed amongst all machines; while in the token ring, the monitor station maintains control over the ring.

The token ring is by no means the only physical ring structure possible, a variation on the token ring is the slotted ring, in which the network supports a single frame structure consisting of a control field with space for the token and space for the bytes in the message.

6.8 Exercises

1. Explain why a token-bus should not be implemented as a singly-linked list.

2. Describe, with the assistance of illustrations, the logical structure of a token bus.

3. When the transmitting station removes the packet from the ring, no CRC calculations are performed. Would CRC calculations be useful when the data is removed from the ring? What benefits, if any, would this give to the transmitting station?

4. Show, by means of an example, how a synchronous communication takes place on the IBM token ring. Show the steps involved in raising the priority and then lowering it.

5. Implement a the token-bus atop an Ethernet.

Chapter 7

Testing Communication Systems

7.1 Introduction

One of the, if not the, biggest headache in developing communication software is determining where a fault is occurring; that is:

- was the transmitting DTE following the protocol and transmitting?
- was the receiving DTE following the protocol and receiving?
- was the channel operating correctly?

Any of these problems can result in untold hours of wasted time, trying to determine 'who-is-doing-what'. Not surprisingly, people implementing or maintaining communication systems in the 'real' world also run into these problems, especially when interconnecting DTEs from different manufacturers, both of whom claim to have implemented the same protocol.

Tools are needed that allow the person attempting to solve the problem to monitor the activities on the communication channel, thereby determining exactly which end is transmitting or where the protocol is not being observed. Fortunately, a number of tools are available that satisfy the above requirements.

7.2 Built-in Tools

Many devices (UARTs, USARTs, NICs, etc) support 'built-in' hardware facilities that permit the programmer to perform numerous software tests. Broadly speaking, these tests fall into two categories: **loopback testing** and **interrupt generation**.

7.2.1 Loopback Testing

A loopback test causes all data normally supplied to the device for transmission to be returned to the processor as if the data has just been received. This permits the local software to be tested without errors being introduced by the remote DTE or the communications channel since the device is separated from the communication channel. For example, a loopback test makes it possible to test both transmission and reception software locally.

Figure 7.1 demonstrates that when a loopback test is being performed. In the case of a UART, the serial transmission hardware continues to transmit marks, and the serial receive hardware is disconnected. Ethernet NICs that support loopback are typically disconnected from the network and data moves between the NIC's input and output queues.

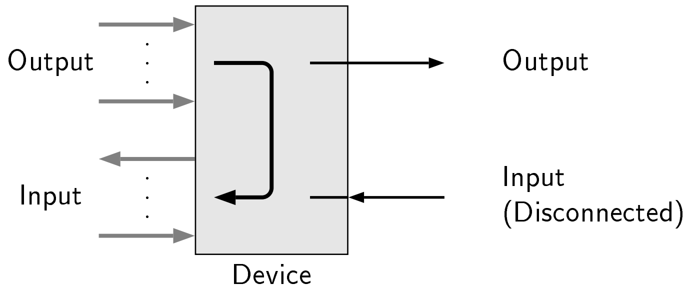


Figure 7.1: A generic loopback test

8250 UART

In 8250 UARTs that support loopback, loopback is set by writing `0x10` to the Modem Control Register; see Figure 7.2.

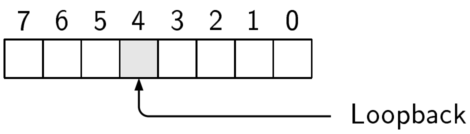


Figure 7.2: Modem Control Register: Loopback bit

3COM 3C509

In the 3COM 3C509, loopback can be programmed through the 16-bit network diagnostic port (window 4, port 6). The 3C509's loopback is considerably more advanced than the 8250 UART, as it supports four different types of loopback (note that the tests are listed in terms of complexity; moving 'deeper' into the NIC):

1. *FIFO loopback.* Data written to the output queue is copied to the input queue.

2. *Controller loopback.* Data written to the NIC is loopbacked at the output of the controller.

3. *ENDEC (Encoder/decoder) loopback.* Data passes through the FIFO and the controller, passing into the encoder/decoder, before being loopbacked.

4. *External loopback.* A test of the entire NIC, allowing data to pass through the entire NIC before being returned to the processor.

7.2.2 Generating Interrupts

All software, whether it's a simple first-year assignment or a commercial data base package must be tested before being released. The same holds true for data communications software. However, testing communications software has the added problem that certain errors are caused by random or spurious events that are hard to duplicate. For example, events such as overrun errors and framing errors are unpredictable, and may take many hundreds of hours of testing for the event to occur (and when the event does occur, it is difficult to duplicate).

8250 UART

The 8250 UART has been designed so that hardware interrupts (such as overrun and framing errors) can be generated through software; meaning that error handling software can be thoroughly tested before being released.

There are six hardware interrupts that can be software generated by the 8250 from the Line Status Register (Data Available, Overrun Error, Framing Error, Parity Error, Break Interrupt, and Transmit Holding Register Empty), as well as four from the Modem Status Register. These interrupts are caused by writing to any of the corresponding status bits in the Line Status Register and having the UART in loopback mode; Figure 7.3 shows the specific bits in the Line Status Register.

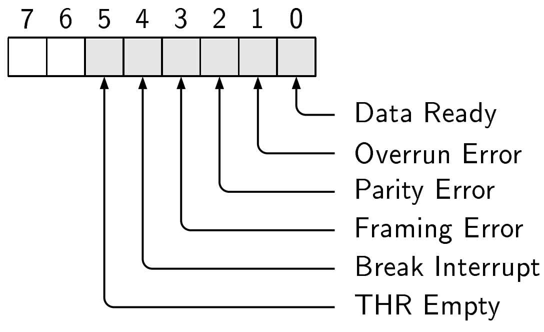


Figure 7.3: The Line Status Register bits affected in loopback mode

For example, the steps required to test the overrun error handling software on the second serial port (`SP2`, base address `0x2F8`), are as follows:

1. Turn loopback on (port `0x2FC`).

2. Write `0x02` to the Line Status Register (port `0x2FD`).

3. A line status interrupt is then generated by the UART, causing control to be passed to the interrupt handler responsible for serial port 2 (`sp2_ih()`).

4. The Interrupt Identification Register will have a value of `0x06` (Receiver Line Status Change) and the Line Status Register will have a value of `0x02`.

5. The interrupt is serviced and cleared as if it were any other interrupt.

Software generated interrupts can continue as long as the UART remains in loopback mode.

3COM 3C509

The 3C509 does not support the equivalent of a independent interrupt generation.

7.3 Channel Monitoring Tools

Built-in testing can force the hardware to emulate certain conditions for software verification; however, these tests still offer no suggestion as to the state of the channel itself. For example, if data transmitted from one DTE fails to arrive at the other, but both DTEs are 'working perfectly', then something is happening to prevent the data from arriving at the receiving DTE. There are three possibilities:

- the data *isn't* being sent by the transmitting DTE.
- the data is being sent, but the communication channel is failing to pass the data to the receiving DTE.
- the data is being sent from the transmitting DTE across the channel correctly, but isn't being *received* by the receiving DTE.

The first two of these possibilities (and perhaps the third) can be overcome by having the capability of observing the activities on the communication channel.

Broadly speaking, there are two types of tools available that permit the observation of the activity between two DTEs on a communication channel: **break-out boxes** and **line analysers**.

7.3.1 The Break-out Box

In the Chapter 2, it was demonstrated that a single communication channel interconnecting two DTEs consists of a number of different connections, each responsible for a specific signal. Since these signals are typically electrical impulses, the signals can be detected and displayed to the person monitoring the channel.

The break-out box is a tool designed to indicate which signals on each of the different connections between the two DTEs are active. In a break-out box, the state of a signal (i.e., active or inactive) is indicated by a light-emitting diode associated with the connection in question. A break-out box monitoring the signals on an RS-232 communication channel would typically signal the conditions on any or all 25 pins. For example, as data is transmitted between the two devices, the light emitting diodes displaying pins 2 and 3 would become active and inactive, depending upon the value of the transmitted byte (e.g., marks turn the LED on, while spaces turn it off).

In addition to monitoring the various signals on the channel, break-out boxes permit testing the channel to activate a non-existing signal. For example, if pin 2 (transmit) is idling (with marks), the breakout box allows the person testing the channel to force the pin into the space state. Similarly, a signal can be deactivated by breaking the connection. Break-out boxes are normally associated with UARTs as opposed to NICs.

Break-out boxes generally cost between \$50.00 and \$200.00.

7.3.2 Line Analysers

Although break-out boxes are useful in determining the state of various signals between the DTEs, they give no indication as to the data that is actually being sent. For example, when a byte is transmitted, the break-out box's transmission LED simply flashes on and off.

Line analysers (or **data analysers**), like break-out boxes, monitor the various signals that occur between the two DTEs. However, line analysers are designed to *show* the data that is being transmitted on the communication channel rather than only indicating that a signal is present. For example, if the byte 'A' is transmitted between a pair of DTEs, a break-out box simply flashes on-and-off; whereas, the line analyser displays the byte 'A'.

At a minimum, a line analyser permits the user to monitor the data on a communication channel by displaying the transmissions from both DTEs, with the data from each DTE appearing on alternate lines of the display. To allow as much flexibility as possible, line analysers often have keyboards that permit the user to specify the configuration of the channel (including line speed, parity, and word size), while more sophisticated and expensive line analysers will do this automatically. In addition, some analysers can be programmed to recognize specific protocols and will search for control sequences associated with the protocol before displaying the captured data.

Line analysers are considerably more expensive than break-out boxes, usually costing anywhere from \$3,000 to over \$20,000.

7.4 Network Analysers

A network analyser is another channel monitoring tool; however, unlike the line analyser, it monitors a network such as an Ethernet or Token Ring.

A simple network analyser can be built from a PC and a NIC. By placing the NIC in promiscuous mode, *all* traffic on the network will be received at the PC. Since the packet structure is known, it is a simple matter to reformat the packet for display on the PC. Advanced network analysers maintain known packet structures for different protocols and can display the packet's fields, based upon the type of the protocol.

The term 'sniffer' is often applied to Ethernet network analysers.

7.5 Summary

This Chapter has examined some of the tools that are available to assist in the testing and development of communications software. Some rudimentary tools are available within the UART itself, allowing conditions such as overrun and the break indication to be tested directly by the processor. For as useful as these tools are, they do not offer the user any indication as to what is happening on the communication channel itself.

Two different types of tool are available for examining the traffic on a channel: the breakout box and the line analyser. Breakout boxes permit the monitoring of various signals on the channel, such as whether a byte has been sent or received; however, no indication is given as to the value of the data. Line analysers on the other hand allow the user to actually see what is happening on the channel, thereby aiding in the development and implementation of communications software.

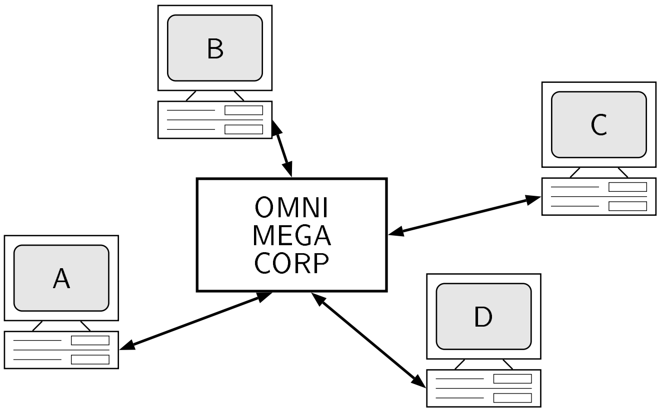
Finally, it is important to remember that there are three possible places to consider when testing communication software: the transmitter, the channel, and the receiver. Tools such as the line analyser allow a software developer to monitor the traffic on a channel and help in tracking down software errors.

7.6 Exercises

1. Write loopback test software for the 3C509 Ethernet. The software should exercise all four levels of Ethernet loopback.
2. Design and implement an Ethernet 'sniffer' to monitor traffic on a local area network. The sniffer should display the source address, the destination address, the type field and the first 30 bytes of the packet.
3. Upgrade the sniffer so that it can target a specific host address (as either the source or the destination).

Part 4

Computer communications within a single organization



Until now, all network communications have taken place within a single network. However, there are times when it is necessary for an application on a computer on one network to communicate with a computer on a separate network; consider the following examples:

- a company consists of a number of separate departments, each with its own local area network. If electronic information, such as reports or personnel data, is to be exchanged between the various departments, a mechanism is needed to transfer the information between the different networks.
- a company has a single network across which all its departments communicate. Several security breeches have been detected, prompting the need to split the network into several smaller networks, while still allowing traffic between departments.

The above are two examples of applications that are often referred to as **enterprise networks** or simply **intranetworks**. An enterprise network is one that 'spans' an organization or enterprise, while intranetwork is the term used to describe communications that take place within one or more connected networks within an organization.

Intranetworks can be designed in a number of different ways, the simplest involves connecting one network to another through a device known as a **repeater**. More sophisticated intranetworks allow the interconnection of networks using a **bridge**. Regardless of the type of intranetwork, the physical properties of the underlying local area network cannot be violated.

More recently, intranetworks have been designed using the Internet protocol; these intranetworks will be discussed in subsequent Parts of the text.

Chapter 8

Interconnecting Networks

8.1 Introduction

One of the biggest criticisms of the explosive spread and popularity of the PC is that the growth and choice of equipment is very often uncontrolled. For example, upper levels of management within a company may decide that all its employees will use specific machines, operating systems, and software packages. However, as employees learn more about the equipment, they may opt for different operating systems, software packages, or may even change machines entirely.

Although independent thought is admirable, it can lead to problems. For example, what happens if the payroll office manager unilaterally decides to purchase an 802.3 bus network to interconnect all of the stations in the payroll office; while at the same time, the personnel office manager purchases an 802.5 ring network? Sooner or later, someone will want to copy information from a station in the payroll office to a station in the personnel office; for example, from station N to station D in Figure 8.1.

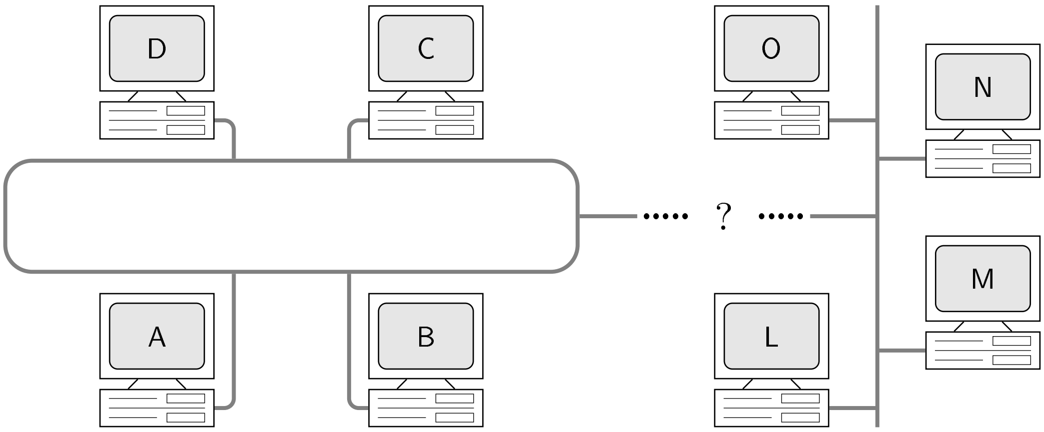


Figure 8.1: Interconnecting a ring to a bus

Since the stations are not on the same type of network, direct interconnection of the networks and subsequent information transfer is not possible because of electrical, packet structure, address, and other protocol differences. Solutions such as physically transferring information on diskettes may be satisfactory on occasion, but if large volumes of information are to be transferred on a regular basis, some other procedure is required.

Similar problems occur should two offices in different cities, each running their own local area networks, need to communicate. If the two cities are interconnected by a wide area network, the ideal solution would be to transfer information between the two networks across the intermediate wide area network. However, since the wide area network probably does not support the protocol used by the local area networks nor does the wide area network achieve the speeds of the local area networks, direct connection of these networks is not possible.

Clearly, many of these problems can disappear if the networks are identical, since the packet structures and protocols are the same: simply interconnect the two networks (i.e., make a larger bus or ring). However, there are instances when it may not be physically possible or even advisable to allow all of the traffic on one network to propagate onto another. Consider the following situations:

- two lightly loaded 802.3 networks are to be interconnected.

  Combining the two lightly loaded networks into a single, larger network may result in a heavily loaded network. The resulting network may offer less than satisfactory performance because of the additional network traffic.

- two local area networks that are more than one kilometre apart.

  The physical interconnection of the two networks may be impossible because of the maximum physical separation of stations allowed by the network.

- a number of local area networks, each supporting several hundred stations.

  The interconnection of the networks may be physically possible, but the resulting number of stations may exceed the number allowed on a single network.

- two local area networks are used within an organization, one with sensitive product information, the other for use by the office staff.

  Combining the two networks and allowing the sensitive information onto the general network may breach company security.

In the above examples, the need for network interconnection has been stated, while at the same time, some of the problems associated with network interconnection have been outlined. If networks are to be interconnected, it is necessary to develop facilities that can convert between the different packet structures, addressing schemes, and other network idiosyncracies while at the same time obeying the protocols associated with the individual networks.

8.2 Background

Connecting a network to one or more other networks is possible as long as the device(s) supporting the interconnection can handle the various differences between the networks. Although there are many different types and makes of device to support network interconnection, manufacturers and users alike often employ the generic term **gateway** when referring to such devices. In Figure 8.2, a gateway connects a ring network to a bus network.

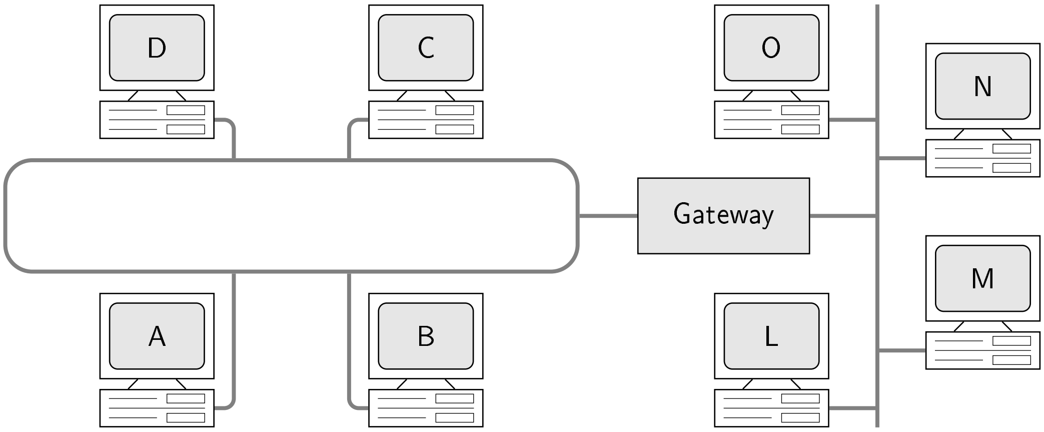


Figure 8.2: A gateway connecting a ring network to a bus network

A gateway connects two or more networks, while obeying the protocols associated with each individual network. For example, the gateway shown in Figure 8.2 must handle both the bus network protocol and the ring network protocol. If the gateway is responsible for changing the packet structure on one network to conform to the rules associated with the other network, it is often referred to as a **protocol converter**.

In some situations it may not be possible to have a single gateway (for example, if the networks are separated beyond the physical limits allowed by the networks or if security reasons dictate which stations are allowed on a network). As shown in Figure 8.3, it may be necessary to have a pair of **half-gateway**s, each attached to a network and the other half-gateway.

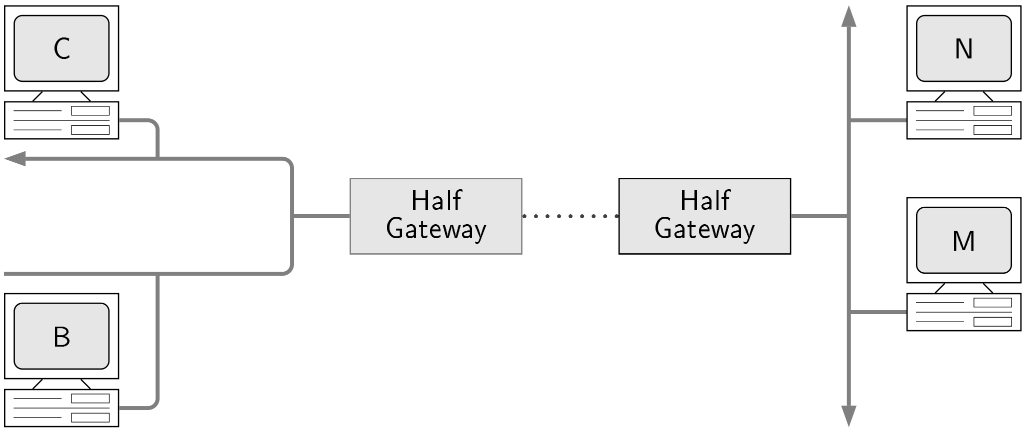


Figure 8.3: Two half-gateways interconnecting two networks

Each half-gateway is connected to a single network and conforms to the standards associated with that network. To send packets between the networks, it is necessary for the half-gateways to support a third, common protocol through which they communicate. Since administration of each half-gateway is left to the network to which it is connected, control over the packets that may be sent between the networks can be better maintained by the different organizations that run each network.

Other issues that are associated with network interconnection include:

Fragmentation and Reassembly. Maximum and minimum packet sizes can differ from network to network, potentially requiring the **fragmentation** or the division of the packet into smaller packets. The small packets are put back together into the larger packet using a process known as **reassembly**. When packets are fragmented, a mechanism such as sequence numbering is required to allow the receiving entity to reassemble the packet in its correct order.

The fragmentation can take place between the source and destination stations, using, for example, the smallest packet size of any of the intervening networks. When taking this approach, gateways are not required to perform any fragmentation or reassembly.

Alternatively, the gateways themselves can perform the fragmentation and reassembly. The destination station may still be expected to perform reassembly if the packet is too large for its network.

Connection-oriented vs. Connectionless Services. Gateways can potentially interconnect networks supporting connection-oriented (i.e., virtual circuit) services, typically wide area networks, with networks supporting connectionless (i.e., datagram) networks. In these situations, it is necessary to decide upon the level and quality of service required by the applications.

If the communication is to be treated as a virtual circuit, the gateway may be required to support a reliable protocol atop the datagram service. However, an application using datagrams may find many of its packets being sent reliably across a virtual circuit; potentially adding to the cost of the overall communication.

Routing. Gateways can be required to perform routing if there are multiple paths through the network between stations. Routing algorithms such as those described for wide area networks in Chapter 13, are often used by gateways.

Other Network Differences. In addition to the above, the source network may support features that are not available on the destination network. A good example of this is synchronous mode on ring networks (see Chapter 6): a station transmitting a synchronous packet expects a reply from the destination within a specific period. If the destination station resides on a separate network, the delays associated with the intermediate gateway(s) forwarding the packet may make a reply within the allotted time period impossible to achieve.

If a gateway acknowledges the packet, this should not be interpreted by the source station as an indication that the destination station has received the packet. In other words, higher level protocols supporting end-to-end acknowledgements may be required to ensure that packets reach their intended destination.

As already mentioned, the term gateway is a generic term applied to almost all network interconnection devices. However, by examining these devices on a layer-by-layer basis, one finds that there are three types of interconnection device: **repeaters** (physical layer), **bridges** (data link layer), and **gateways** (network layer). Protocol conversion, another issue in internetwork communication, has been discussed in previous Chapters.

8.3 Repeaters

A repeater is a device that interconnects two **homogeneous** (i.e., identical) networks, making a single, larger network. Repeaters are intended to take the traffic from one part of the network and transmit it directly onto the other part without making any changes to the frame structure or the physical layer (i.e., OSI layer 1) protocol. A repeater, its relationship with two networks, and its associated layer are shown in Figure 8.4.

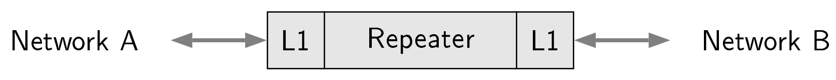


Figure 8.4: A repeater connects two networks at the physical layer

A repeater operates at the physical layer, amplifying or regenerating signals. Repeaters can help overcome segment length limitations and can be used to interconnect networks using different physical media (such as coaxial cable to fiber optic). A repeater connecting two bus segments passes the collisions between the two segments. A ring repeater extends the size of the ring; the number of repeaters necessary in a ring equals the number of rings being connected. To avoid the situation where two or more stations receive the same packet, all stations must be assigned unique addresses.

Finally, a repeater is not allowed to violate the physical limitations placed upon the network. For example, two maximum length Ethernets (i.e., 2.5 kilometres) cannot be joined by a repeater to form a single 5 kilometre Ethernet, since this violates the Ethernet's cable length restrictions.

8.4 Bridges

A bridge is a device or layer of software that allows the interconnection of two or more local area networks at the media-access (MAC) sublayer of the data link layer (OSI layer 2). The bridge structure is shown in Figure 8.5.

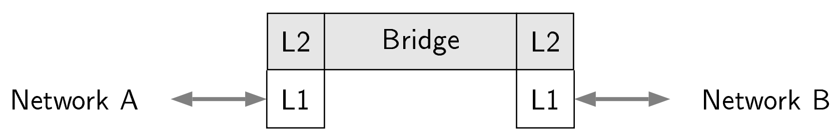


Figure 8.5: A bridge functions at the Data Link Layer

The resulting network is referred to as a individual LAN known as a **segment**. Each segment attaches to the bridge through a **port**; a bridge can connect to many segments. Bridges act as store-and-forward devices, taking packets from one segment and forwarding them onto others, following the rules associated with each segment.

Bridges offer a number of advantages over repeaters:

- different types of local area networks can be interconnected. For example, a bridge could connect an 802.3 bus to an 802.5 ring. Packets destined for the ring must wait until the bridge is in possession of the token before transmission, while packets intended for the bus are transmitted when the bus is idle.
- distance limitations can be overcome. Two or more 802.3 bus networks could be connected by a bridge; packets could be sent from one segment to the other through the bridge. The bridge could give give the illusion of a single, large network. But since the bridge operates at the data link layer, the physical layer requirements (such as distance limitations and electrical requirements) are not violated.
- a single, large local area network can be divided into a number of smaller segments, all interconnected through one or more bridges. The bridge can help overcome the limitations associated with operating a single network that is approaching its physical limitations (i.e., number of stations or maximum length).

The IEEE 802.1 Standard defines two types of bridge: those that are transparent to the stations involved in the communication (**transparent** bridges), and those that require the stations involved in the communication to be aware of the existence of the bridges (**source routing** bridges). Both types are now considered.

8.4.1 Transparent Bridges

To operate transparently, a bridge must function in promiscuous mode, receiving and inspecting all packets sent from each segment to which the bridge is connected. Packets are addressed to the destination station, rather than to a bridge. All stations must be assigned unique addresses.

In it simplest form, a transparent bridge forwards all packets received from one segment to all those to which it is connected. This approach has a number of limitations, the most serious being flooding: all packets are sent to all segments, regardless of the destination. By allowing all traffic to propagate between the interconnected segments through the bridge, a series of lightly loaded segments can turn into a single, heavily loaded bridged network.

This suggests that there may be a benefit in having the bridge perform some type of **packet filtering**. Ideally, the filtering should allow only packets destined to a remote segment to pass through the bridge, thereby ensuring that the only traffic on a segment is either packets for stations on that segment or packets enroute to a remote station being sent via a bridge.

Packet filtering is achieved by having the bridge maintain a list of addresses for the stations associated with each segment to which it is connected; the bridge can then selectively forward packets between segments by examining the *destination* address of each packet. If the packet's destination address is for a station on a segment other than the one from which the packet has been received, the packet can be forwarded to the destination's segment. To allow the station to make these routing decisions, a **forwarding database** is maintained, consisting of each known destination address and the segment from which the packet was received.

Using the example in Figure 8.6, if station S on segment 3 sends a packet to station E on segment 2, the bridge will forward the message onto segment 2 because station E does not exist on segment 3 but it does exist on segment 2 (this is determined by the bridge examining the forwarding database).

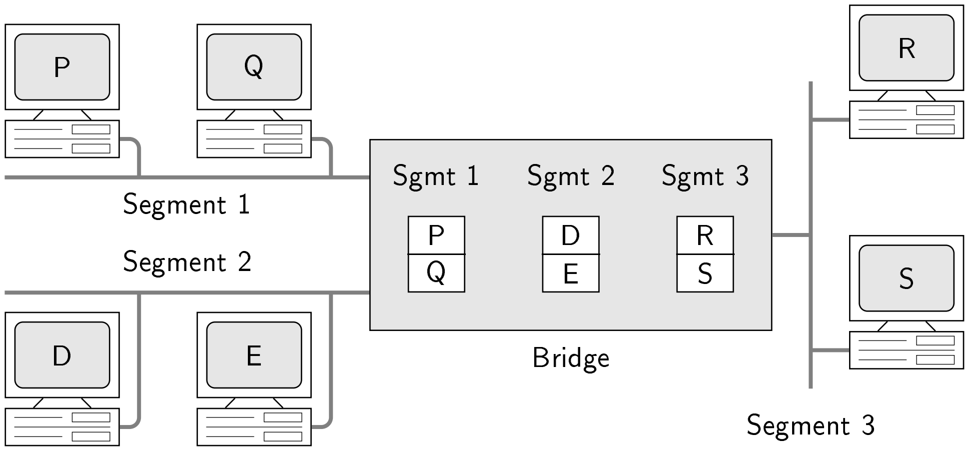


Figure 8.6: A transparent bridge

Packets sent to stations on the *same* segment are not forwarded even though they are received by the bridge. For example, a packet from station D to station E would not be forwarded by the bridge because station E is on segment 2 (once again, the bridge determines this from the forwarding database).

If the bridge receives a packet with an unknown destination address, the packet is forwarded on all segments except the one on which it was received. For example, if station P sends a packet addressed to station X, the packet will be forwarded on segments 2 and 3, since address X does not exist in the forwarding database. If other bridges are attached to any of these segments, they will continue to forward the packet until it reaches station X. Having the bridge selectively forward packets ensures that the only packets on a segment (other than those enroute to another bridge) are those intended for stations on that segment.

The forwarding database can be initialized in one of a number of ways:

- the station addresses can be stored in a file that is read by the bridge when it is brought on-line. The problem with this approach is that the contents of the file are fixed, meaning that changing a station (i.e., removing it or having it change segments) requires that the file be explicitly altered, typically by the network manager.

- individual stations can explicitly inform the bridge of their existence. When a station joins or leaves a segment, the bridge can be informed, causing it to update its forwarding database. Although this does eliminate the need for explicit database alteration by the network administrator, special protocols are required to allow the stations to inform the bridge of their status.

- the bridge can determine the locations of the various stations on the segment by examining the *source* address of each packet. That is, whenever the bridge receives a packet from a segment, it immediately knows that there is a station on the segment with the address found in the packet. For example, in the configuration shown in Figure 8.7, packets sent on segment 1 will have source addresses D or E, while packets sent on segment 2 will have source addresses R or S.

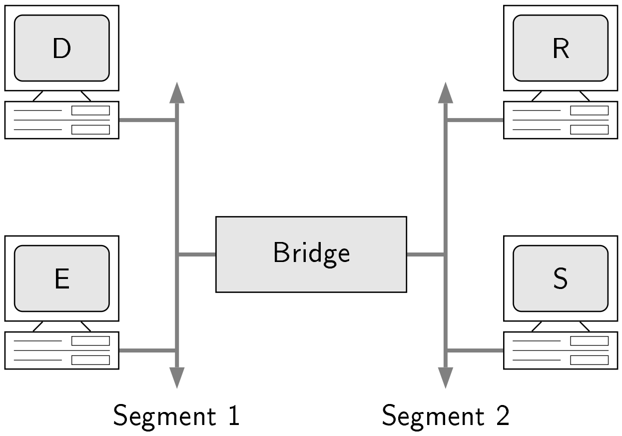


Figure 8.7: By examining the source of a packet, the bridge knows a station's segment

By examining the source addresses associated with each packet, the bridge can construct the forwarding database associated with each segment. This type of bridge is known as an **intelligent bridge**.

An intelligent bridge initially forwards *all* of the packets that it receives, 'learning' the location of each station from the *source* address while building the forwarding database. Eventually, the bridge 'knows' the location of each station on a segment and can filter the packets by examining the *destination* address.

Should a station change segments or be removed from its segment, the bridge can also gain this information from the traffic. A station changing segments would have its address appear on both segments; the bridge could then update its forwarding database on the basis of this information. In addition, the bridge determines when a station becomes inactive by periodically purging its forwarding database of stations that have not sent a packet within a fixed period of time (typically every few minutes).

IEEE 802 Transparent Bridge

The IEEE 802 transparent bridge is an intelligent bridge that performs packet filtering. Each 802 bridge maintains a forwarding database that consists of entries in the form: *port*, the segment from which the packet was received, and *address*, the address of the station.

All packets that are received by the bridge are subject to two separate operations:

Bridge Forwarding. When a packet is received, the bridge examines the forwarding database, comparing the list of station addresses to the destination address in the packet:

- if the address does not exist in the database, the packet is forwarded on all segments to which the bridge is attached, except the one on which the packet was received.
- if the address does exist in the database, the source port is then considered. If the port from which the packet is received is the same as the port in the database, the packet is discarded; otherwise the packet is forwarded to the segment specified.

Bridge Learning. Once bridge forwarding is completed, the packet's address is compared to the list of addresses in the forwarding database. If the address does not exist in the forwarding database, both the source port and the packet's address are recorded in the database.

Inactive stations have their addresses removed from the forwarding database by the bridge. The amount of time an address is left in the database is set by the network administrator.

If reliability is an issue, some network administrators may institute a policy of having two or more bridges connecting a pair of segments. By having two paths through the network, an interesting problem arises: packets may be forwarded indefinitely. Consider the situation shown in Figure 8.8, in which segments A and B are connected by two bridges and station S on segment B attempts to send a message to station L:

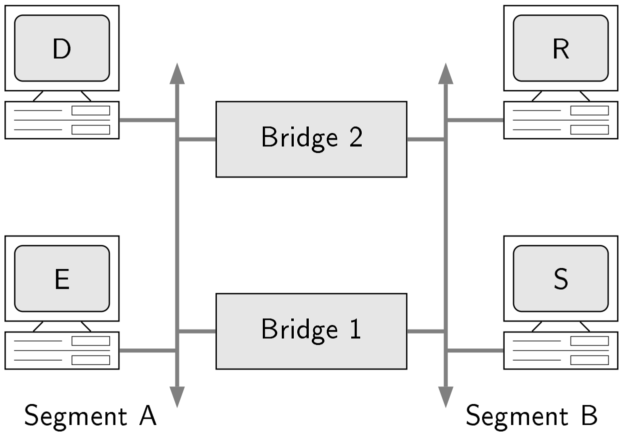


Figure 8.8: Interconnecting two networks with two bridges

When the packet addressed to station L is received by the bridges, both bridges attempt to forward the packet since (it is assumed) an entry for station L does not exist in either forwarding database (station L does not exist on either segment). Each bridge will then receive the packet forwarded on segment A from the other bridge; and will forward the packet addressed to station L onto segment B. This cycle will continue indefinitely.

If the bridge learning and forwarding algorithms are to succeed, there can be only a single path between any of the segments making up the bridged network, regardless of the physical topology. Fortunately, it is possible to impose a **spanning tree** onto a bridged network to ensure that there are no loops in the network. A spanning tree is a structure with one bridge as the root of the tree, and other bridges restricting the flow of packets so that no loops exist within the overall network. The 802 transparent bridge uses a **spanning tree algorithm** to generate spanning trees.

The spanning tree algorithm operates as follows. One bridge is chosen as the root of the spanning tree, typically the station with the lowest physical address. Then, all bridges determine the shortest path from the segments to which they are connected to the root bridge (i.e., the number of hops required by a packet to reach root bridge). If two (or more) bridges connected to the same segment require the same number of hops, the bridge with the lower physical address is chosen, the other bridge is *blocked*. (Blocking is a logical operation, the bridge still receives packets, it simply discards those that are not to be forwarded.) The spanning tree algorithm functions dynamically, with each bridge periodically informing all other bridges of its location, neighbouring bridges, and status. Should the status of a bridge or segment change (i.e., fail or become active), the spanning tree can be reconfigured.

For example, consider the LANs interconnected by bridges in Figure 8.9; since there are multiple paths through the network, packets can potentially remain cycling 'forever'.

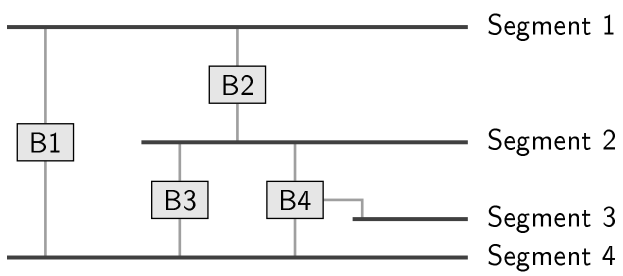


Figure 8.9: With multiple paths, a packet can circulate 'forever'

If bridge B1 is taken as the root of the spanning tree, then the other bridges can determine the shortest path to the root bridge. The shortest path from segment 2 to the root bridge can be by bridges B2, B3, or B4; assuming that bridge B2 has the lowest physical address of the three bridges, bridges B3 and B4 block (i.e., ignore) all packets sent to and from segment 2. The shortest (and only) path from segment 3 is via bridge B4, therefore bridge B4 will carry all packets destined for and received from segment 3. Bridge B3 blocks *all* packets and bridge B4 blocks packets sent across segment 2. The resulting spanning tree is shown in Figure 8.10 (the dotted lines denote physical links that are currently blocked).

The bridged network is now free of all loops. Since the bridges are always in communication, a change in topology (for example, a bridge failure) will result in a new spanning tree.

8.4.2 Source Routing Bridges

Careful observation of the spanning tree in the previous section shows that the resulting bridged network, although free of possible loops, is not necessarily optimal. For example, a transmission from segment 3 to segment 2 results in the packet being sent via bridges B4, B1, and B2; even though the two segments are physically connected by bridge B4.

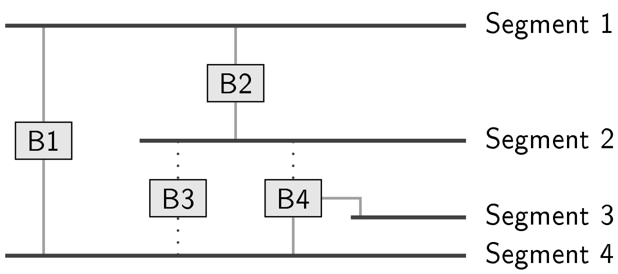


Figure 8.10: A spanning tree (from Figure 8.9)

The designers of the IBM Token Ring recognized this limitation and developed an alternative to the spanning tree for bridged token rings known as **source routing**. Source routing operates as the name implies, the source station specifies the route of the packet rather than letting the bridges determine the route. Not surprisingly, the route the logical link control packet will take is stored within the frame by the source station; see Figure 8.11.

DST Addr	SRC Addr	Routing-Info	DSAP	SSAP	Control	Data	FCS

Figure 8.11: The logical link control packet for source routing

Routing-Info is a variable-length field up to 18 bytes in length. The first two bytes are the routing control field and contain the following information:

- whether the packet is a broadcast packet, intended for all rings;
- the length of the Routing Info field;
- the direction of the packet, either from or to the source station;
- an indication as to the largest frame supported by the bridge for broadcast packets.

The remaining 16 bytes contain up to eight route-designators. The 16-bit route-designator field consists of a unique 12-bit LAN number and a 4-bit bridge number, identifying a specific bridge on a LAN.

Before a packet can be sent, the source must determine the route to be taken by the packet. This is achieved by the source sending a **discovery** packet with the destination station's address to *all* token rings making up the network. As each discovery packet makes its way through the network, the route of the packet is recorded in the discovery packet by each bridge. Those packets which reach the intended destination are returned by the destination to the source back along the recorded route. The source can then choose which route to use for its communication from the discovery packets returned.

Once the route has been chosen, the source station sends packets with the high-order bit of the address set. This signals the bridge that the packet is intended for a remote token ring rather than the local token ring. The bridge inspects the Routing Info field to determine the route.

Source routing has been adopted by the IEEE 802 Committee for routing in interconnected 802.5 token rings. However, the source routing algorithm is not restricted to ring local area networks, it can be applied to other local area networks as well as wide area networks.

8.5 Gateways

In many internetworking situations, it is not possible to use a repeater or a bridge because the networks to be connected support different physical properties, addressing schemes, and possibly even owners. A gateway (also known as a **router**) is a device, operating at the network layer (i.e., OSI layer 3), that is intended to overcome these limitations; the relationship between the gateway and the OSI layers is presented in Figure 8.12.

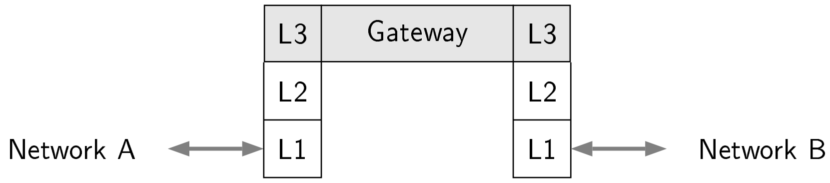


Figure 8.12: The relationship between a Gateway and OSI layers 1 through 3

Since gateways are associated with wide area networks, there are, broadly speaking, two types of gateways: those for connection-oriented services and those for connectionless services.

8.5.1 X.75

The same arguments against the direct connection of dissimilar local area networks can also be applied to the direct connection of wide area networks, even if they share a common protocol such as X.25:

- the internal protocols used by the network may differ. Connecting a node on one network to a node on another may not be possible because of physical connection (layer 1) differences or internodal protocol (layer 2) differences.
- since most public wide area networks are run by different organizations, each party is usually strongly reluctant to permit the 'other' organization to connect one of 'their' nodes to 'our' network.

Instead, most wide area networks that allow connection to other wide area networks communicate through a pair of half-gateways using a common gateway protocol. In the case of X.25, the gateway protocol is ITU X.75.

X.75 is a connection-oriented protocol and is used when an application on an X.25 host (or PAD) specifies an international call in the call request packet. (The international call is indicated by an X.121 address indicating a network other than the current network.) The node to which the X.25

host is connected then creates a virtual circuit to the half-gateway node. The half-gateway creates another virtual circuit to the half-gateway on the remote wide area network. The remote half-gateway places a virtual circuit to the destination X.25 host (or another half-gateway; repeating the cycle). Once the virtual circuits are established across the various networks, communications may proceed.

The layer 3 protocols between a pair of X.25 hosts (denoted H) between nodes (denoted N) and a pair of half gateways (denoted G) are illustrated in Figure 8.13.

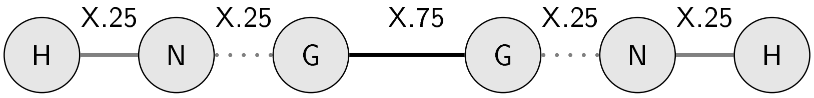


Figure 8.13: The relationship between X.25 and X.75

The X.75 protocol is similar to the X.25 protocol. Additional features in X.75 are needed for routing and accounting.

8.6 Summary

Ideally, all the services required by an application should reside on the application's network. However, due to equipment costs, physical distances, and organizational differences, the service required by the application may reside on a different network. In order for the application to access the service on the remote network, facilities must be in place to allow the interconnection of the different networks. Broadly speaking, there are three such approaches, each associated with an OSI layer:

Repeaters. A repeater is a device that allows the connection of homogeneous networks at the physical layer. Repeaters are not responsible for routing or fragmentation, nor are they affected by the type of service offered (i.e., connection-oriented or connectionless).

Bridges. Bridges operate at the data link layer and permit the interconnection of heterogenous networks. A local area network made from a number of local area networks (known as segments) is called a bridged local area network. Two types of bridges have been considered in this Chapter: transparent bridges and source routing bridges. Transparent bridges require less processing on the part of the stations, but often result in the use of a non-optimal spanning tree to ensure that packets do not circulate forever; whereas source routing bridges require the active involvement of the source station in the establishment of a potentially optimal route through the network.

Gateways. Gateways, like bridges, also permit the interconnection of heterogeneous networks, but operate at the network layer. Taking this approach, a network operated by different organizations supporting different protocols and addressing schemes can be interconnected.

8.7 Exercises

1. Explain both how and why a repeater connecting two bus networks permits collisions to be forwarded.

2. Show, by means of examples, the difficulties in making synchronous mode work when connecting a pair of rings by a bridge.

3. The Chapter described one possible type of transparent bridge in which all packets are forwarded, resulting in flooding and, potentially, packets cycling forever. If a hop-count is included in each packet, would this help? How would this compare with the spanning tree and source routing algorithms? Remember, in the spanning tree algorithm the bridges are in constant communication with one another.

4. An alternative method of routing is known as **reverse path forwarding**, in which packets are forwarded only if they are received on a link with the shortest path back to the source of the packet. Packets are forwarded on all links except the one on which it was received.

   Consider how reverse path forwarding could be implemented. How would each gateway 'know' whether one link was a shorter path back to the source than another? Suggest a structure for the routing table. Could the reverse path ever be changed – how?

5. Suggest methods whereby multicast could be supported on a bridged local area network. The solution(s) should avoid simply broadcasting each multicast message on all networks, ideally each multicast packet should only be transmitted on those networks were a member of the multicast set exists.

   Can multicast be supported by the spanning tree algorithm? The source routing algorithm?

6. Three approaches to initializing an intelligent bridge were suggested in this Chapter. Describe the methods needed to update the station address tables if:

   (a) a station was removed entirely from the network.

   (b) a station was removed from one network and placed on the other (keeping its physical address).

7. Show the resulting spanning tree if bridge B1 in Figure 8.14 were to fail.

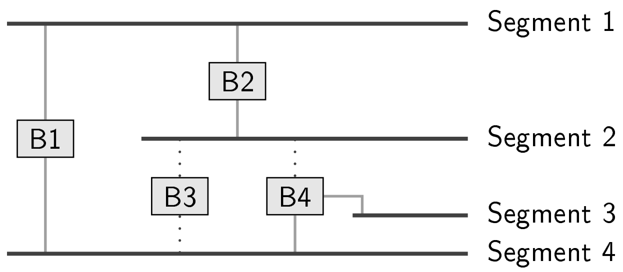


Figure 8.14: Diagram for Exercise 7

8. Design and implement an intelligent bridge using the 3C509. Note that the PC will require two network adaptors and it will be necessary to distinguish between the two of them.

   If a second hub isn't available, connect either card to a single PC that will generate fictitious network traffic (using a variety of source addresses). A special twisted cable will be required for this connection.

   Once implemented, attempt any or all of the remaining questions.

9. To prove that the bridge actually works, transmit packets from a station on one network to a station on the other. Now, consider the following questions:

   - How long does it take for the bridge to 'learn' about all the stations on the different networks?
   - What happens to packets that are addressed to stations that do not exist on either network?
   - What happens if two stations on different networks have the same address?

10. The bridge algorithm works fine for a bridge connecting two networks; however, it is not general purpose and could not be used to support more than two networks. Redesign the routing algorithm so that it is general purpose and can function with anywhere from 2 to 16 different links.

    Implement the algorithm. Devise a test plan to exercise the software.

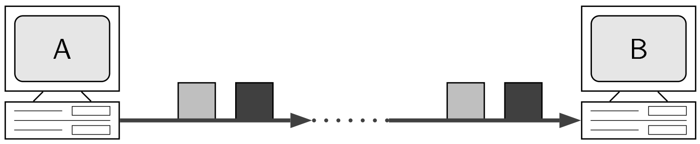
11. What changes would be necessary to the bridge software if the two interconnected local area networks support different packet sizes?

Part 5

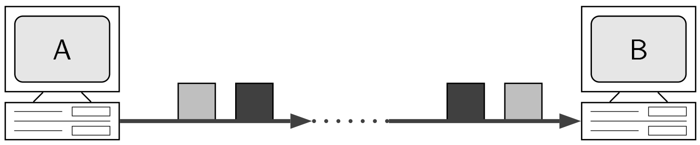
Reliable Computer Communications

When a packet or message is transmitted, the transmitting DTE has no indication as to the success of the transmission. Similarly, the receiving DTE only 'knows' that a transmission has occurred when it receives a packet or message. There are four possible scenarios:

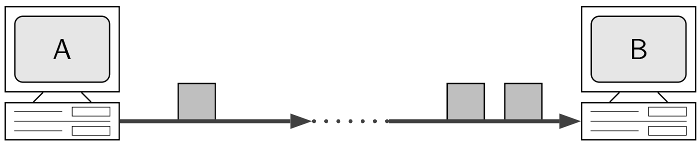
1. Packets that are transmitted are received in the order sent and no errors are detected:



2. In a network with multiple paths, one or more packets that are transmitted are received out-of-order:



3. In a network with multiple paths, a transmitted packet is duplicated and received two (or more) times by the receiving DTE:



4. A transmitted packet is lost, either a network fault (noise or someone tripping over a cable) or some other fault such as network congestion (too many messages on the network) or an internal software failure (they do happen):

A communication is either reliable or it is not. A reliable one attempts to handle all of the conditions shown above, an unreliable one does not. Given the importance of computer communications to today's society, it should not be surprising that a great deal of time and effort has gone into the development of reliable **protocols**.

Part 5 examines three different types of reliable protocol; each of the protocols are discussed in terms of the characteristics introduced in Chapter 1, notably:

- Coding of Information
- Control
- Error checking and recovery
- Channel utilization
- Synchronization and Timing, and
- Transparency

Although all of these characteristics must be considered when designing a protocol, the success of a protocol is often determined in how it handles error checking and recovery. There are a number of techniques that allow a receiving DTE to determine whether a frame has been received in error. The question is: once the receiving DTE has detected the error, what steps should be taken to correct it?

Since there is only one transmitter and one receiver, the possible places where the error can be corrected are somewhat limited:

- the receiving DTE can request the transmitting DTE to retransmit the frame in error: this is known as **backward error recovery**.

  Because of the distances and speeds involved, most terrestrial communications do not expect the receiving DTE to correct errors detected within the frame. Instead, most protocols are written so that the receiving DTE *discards* the frames in error and requests a retransmission of the frame using a technique known as **automatic repeat request** (or **ARQ**). There are a number of different ARQ algorithms used in backward error recovery, Chapters 9, 10, and 11 examine the three most widely used.

- the receiving DTE can attempt to correct the error itself using what is known as **forward error recovery**; forward error recovery techniques are discussed in Chapter 12.

Chapter 9

The Stop-and-Wait Protocol

9.1 Introduction

The simplest ARQ protocol is the stop-and-wait protocol, it requires the transmitting DTE to send a frame and then wait for a response. All communications cease until the destination DTE responds.

9.2 The Stop-and-Wait Protocol

A **stop-and-wait** protocol requires the transmitting entity to send one frame at a time and await a reply from the receiving entity. The reply is an **acknowledgement**, indicating whether the frame has been received correctly.

If the frame is correctly received, the receiving entity responds with a **positive acknowledgement** (or *ACK*). Figure 9.1 is a **time sequence diagram** that illustrates the basic cycle of a stop-and-wait protocol (unless otherwise indicated, it is assumed that the frame the DTE is about to transmit contains a message).

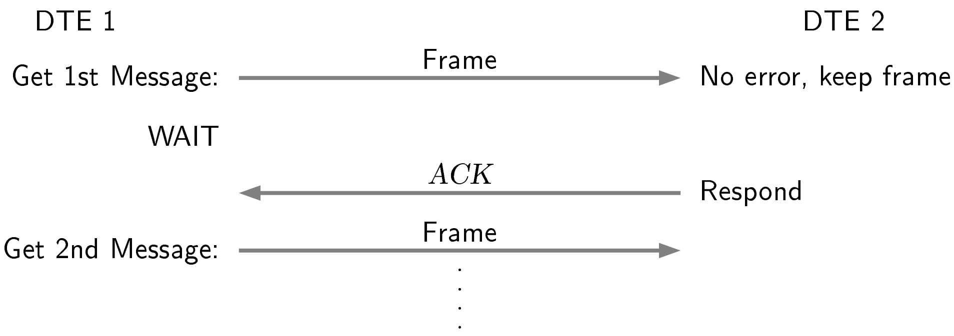


Figure 9.1: A time-sequence diagram for the stop-and-wait protocol

In Figure 9.1, DTE 1 gets a message, transmits the message in a frame, and then waits for a reply from DTE 2. When DTE 2 receives the error-free message, an acknowledgement is returned. Upon receipt of the *ACK*, the transmitting entity (DTE 1) can discard the message just sent and get the next message for transmission.

However, if a frame is received incorrectly, the receiving entity (DTE 2) responds with a **negative acknowledgement** (or *NAK*), requiring the transmitting entity to **retransmit** the *original* frame. As shown in Figure 9.2, once the frame is received correctly, transmission can continue with the subsequent frames.

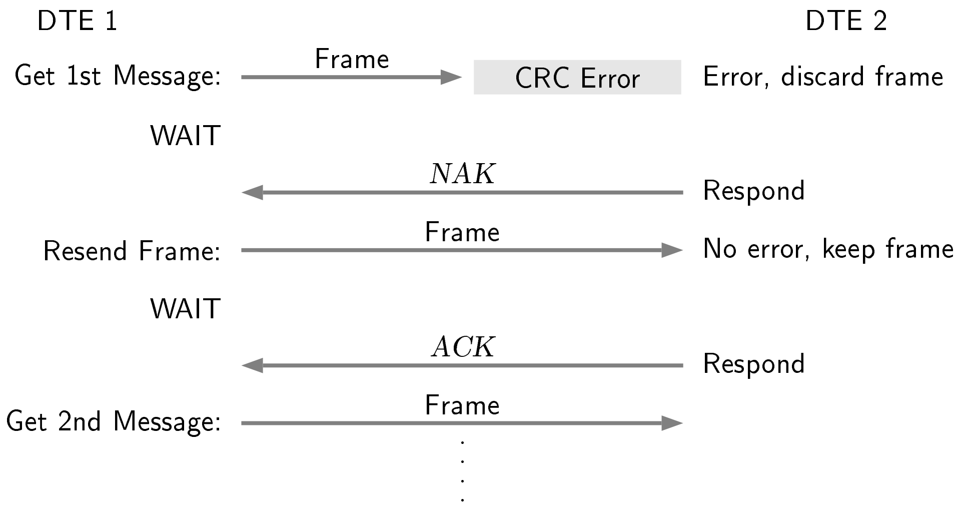


Figure 9.2: Error detection and correction in the stop-and-wait protocol

The transmit-frame and receive-acknowledgement (positive or negative) cycle continues until all messages have been sent (in frames) and acknowledged.

The algorithms for the stop-and-wait protocol can be written as follows:

- Transmitting Data:
  1. Send a frame (containing a message) to the remote entity.
  2. Wait for an acknowledgement.
  3. If the acknowledgement is positive, get the next message. Otherwise, keep the same message for retransmission.
  4. Repeat steps 1, 2, and 3 while there are more messages to send.

- Receiving Data:
  1. Wait for a frame.
  2. If no errors are detected, keep the frame (extract the message) and respond with a positive acknowledgement.
  3. If errors are detected, discard the frame and respond with a negative acknowledgement.

4. Repeat steps 1, 2, and 3 while there is more to be received.

The stop-and-wait protocol as described assumes that *all* frames and acknowledgements reach their intended destination. This could be an incorrect assumption since the flow of data may cease if a frame or an acknowledgement is lost (due to noise or a momentary loss of connection), leaving the transmitting entity waiting *indefinitely* for an acknowledgement. For example, if the acknowledgement (positive or negative) is lost, the transmitter of the original frame might wait forever, stopping the flow of information. An example of the loss of a positive acknowledgement is shown in Figure 9.3.

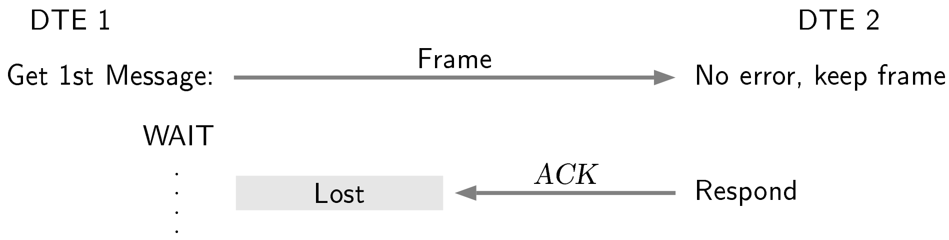


Figure 9.3: Information flow ceases if the ACK is lost

There are a number of ways in which the absence of an acknowledgement can be overcome to ensure the continuing flow of frames. The most common means is to limit the amount of *time* the transmitting entity can wait for a response. That is, if a response isn't received within a specified time period, the transmitting entity simply retransmits the frame; as shown in Figure 9.4.

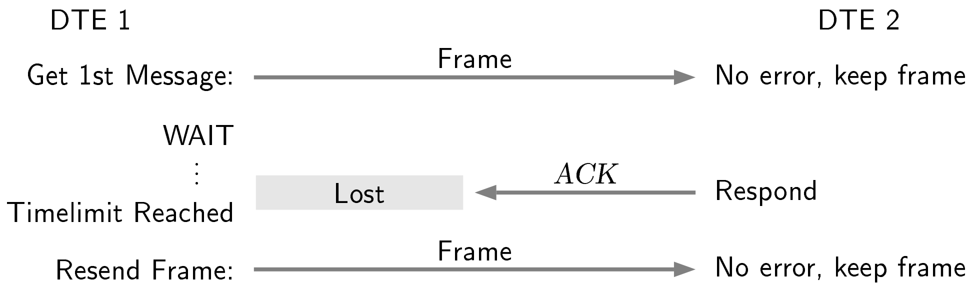


Figure 9.4: Using a time limit to detect a missing message

This solution, however, can lead to another problem: DTE 2 has now received a *second* copy of DTE 1's original frame (meaning that the same message has been received twice). The question arises: how can DTE 2 distinguish between frames that are retransmissions of previous ones and frames that are entirely new?

One possible solution is to have DTE 2 compare the contents of the previous frame (i.e., the message) with the one just received: if they are the same, the one just received can be discarded. This is not necessarily a good idea, since it ignores the possibility that two or more different messages could have the same value: such as a pair of blank lines in a file. An alternative is to have each frame associated with an identifier that allows the receiving DTE to distinguish between incoming frames. For example, the identifier could be an integer that is sent with the message as part of the frame and is unique to each frame; see Figure 9.5. The identifier is more commonly referred to as a **sequence number**.

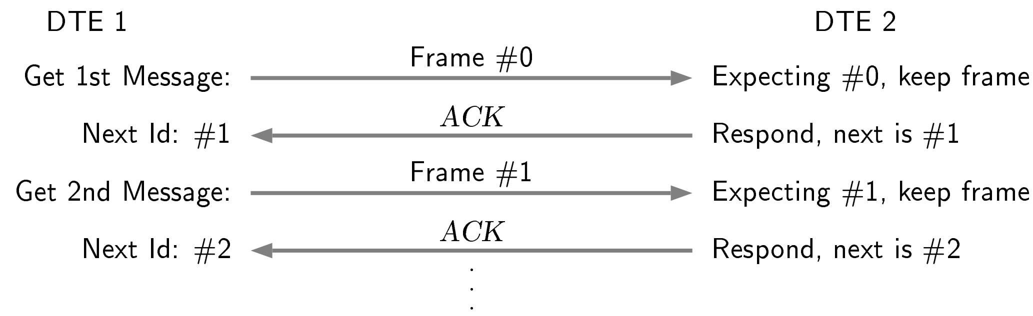


Figure 9.5: Associating each frame with an identifier

Duplicate frames can be identified and rejected by having the receiving DTE examine the sequence number associated with each frame, for example, as in Figure 9.6.

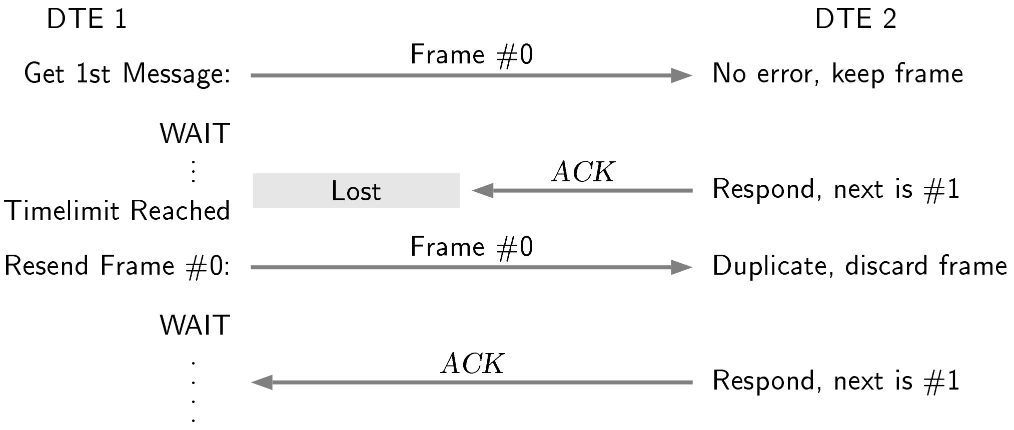


Figure 9.6: Recognizing a duplicate frame by its sequence number

In Figure 9.6, DTE 1 sends frame #0 and waits for a reply. Since no reply is received before the timelimit is reached, frame #0 is retransmitted. DTE 2 initially receives frame #0 and responds with an *ACK* since 0 is the expected sequence number; the expected sequence number is then increased to 1. When the next frame arrives with a sequence number less than the expected sequence number, DTE 2 correctly assumes that the frame is a duplicate and discards it. The frame is a duplicate because the previous *ACK* was lost, meaning that DTE 2 must retransmit the acknowledgement.

Now, when a duplicate frame is received, the receiving DTE need only inspect the sequence number to determine whether the frame is new or a retransmission of the previous one. As before, when the transmitting DTE finally receives a positive acknowledgement, the next frame can be transmitted.

It may appear that each frame sent must have its own unique sequence number (for example, starting at 0 and proceeding to infinity). However, careful examination of the transmission-acknowledgement cycle reveals that at any particular moment, only a single frame requires an acknowledgement. This means that the stop-and-wait protocol needs at most *two* distinct sequence numbers: one for the frame about to be sent, and the other for the previous frame. By alternating between two sequence numbers (say, 0 and 1), the two DTEs can remain in-step and determine which frames are lost or are received in error. Figure 9.7 is an example of the stop-and-wait protocol transmitting five messages using two sequence numbers.

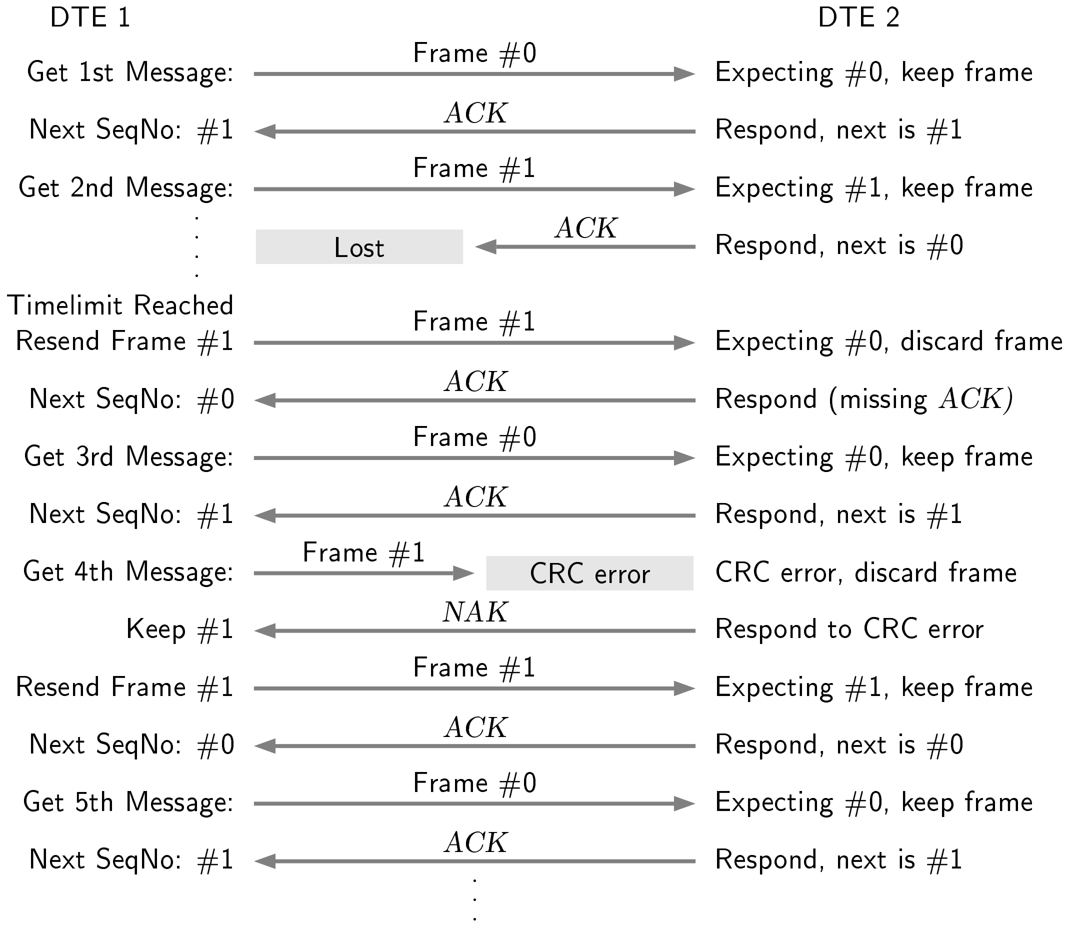


Figure 9.7: An example of the stop-and-wait protocol

The receiving DTE must maintain state information regarding the frame it is *about* to receive (that is, the expected sequence number). Furthermore, both DTEs must agree to a common starting sequence number (typically 0). A complete stop-and-wait transmission state machine is shown in Figure 9.8 (the first frame is sent with sequence number 0 and enters state T2).

In the diagram shown in Figure 9.8, state T2 is entered after frame #0 has been sent (either initially or after receiving an *ACK* in state T1). The machine stays in state T2 if a Timeout occurs or a *NAK* is received; either of which events cause frame #0 to be retransmitted. Upon receipt of an *ACK* in state T2, frame #1 is sent and control passes to state T1. A Timeout or a *NAK* causes

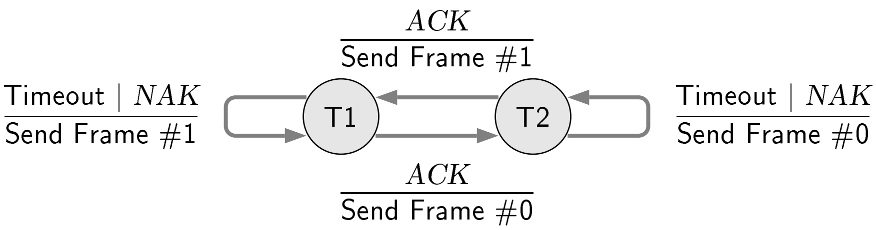


Figure 9.8: Stop-and-wait transmission state diagram

frame 1 to be retransmitted from state T1, while an *ACK* results in a state change (to T2) and the transmission of frame #0. Although not shown, the machine should terminate upon receipt of an acknowledgement of the last frame.

A stop-and-wait receive state machine for the stop-and-wait protocol is shown in Figure 9.9. The receive state machine consists of two states: R1 (to be entered initially while waiting for frame #0, or after receiving frame #1); and R2 (entered after receiving frame #0). In both states, the reception of a frame with a bad CRC causes the transmission of a negative acknowledgement (*NAK*), while the receipt of a valid frame causes a positive acknowledgement (*ACK*) to be transmitted. Receiving a valid frame after it has already been acknowledged (for example, receiving frame #1 in state R1) indicates that the transmitter did not receive the previous acknowledgement, requiring the transmission of an *ACK*. The reception of a bad CRC in either state could be a retransmission of a previous frame (if the last *ACK* was not received) or it could be the transmission of a new frame. For example, in state R2, if frame #0 is received with an invalid CRC, the last *ACK* was not received and the retransmission was in error. If frame #1 is received with a bad CRC in state R2 it means that the last frame was correctly acknowledged, but frame #1 was damaged during its transmission. In either event, the response is a *NAK*.

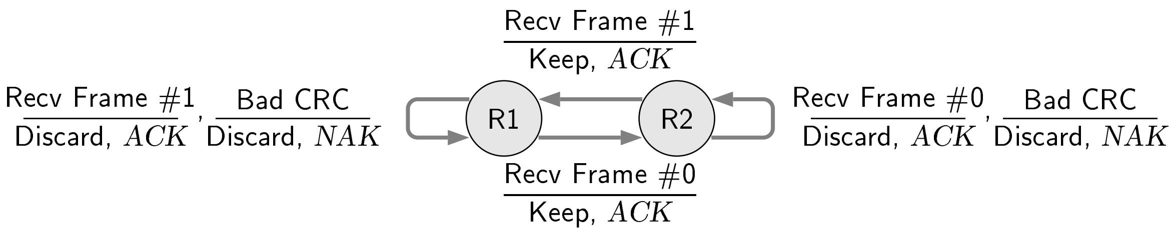


Figure 9.9: Stop-and-wait receive state diagram

Interestingly enough, the stop-and-wait protocol does not require the negative acknowledgement: if the DTE receives a frame with a CRC error, there is no need to respond since the transmitter will eventually timeout and retransmit the same frame again. The advantage of using the negative acknowledgement is that throughput can be increased because a frame with a bad CRC is retransmitted as soon as the transmitting DTE receives the *NAK*.

9.3 Byte-Oriented Synchronous Protocols

Byte-oriented synchronous protocols are those that use the control characters found in character codes (such as ASCII or EBCDIC) to delimit frames and support other aspects of the protocol. Although there are numerous byte-oriented protocols in existence, the following is a brief introduction to the 'grandparent' of them all: IBM's early (and highly successful) Binary Synchronous Communications Protocol (also known as Bisync or BSC; so named because it allows the synchronous transmission of binary data). Bisync was designed around the EBCDIC character set although ASCII implementations do exist.

The Bisync protocol is intended for batch processing in which users submit their 'jobs' to a **remote job entry** (or RJE) station, typically an input device and a line printer. The job, consisting of a file of instructions is read and transmitted to the central host for processing. Once the processing has finished, the output is **spooled** to disk and when the opportunity arises, sent to the user's RJE for printing.

Bisync is a stop-and-wait protocol that allows data transfer between pairs of **stations**. Bisync has a window size of one (i.e., there can be at most one pending data frame) and two sequence numbers (`0` and `1`). Each data frame is associated with a sequence number: the frames transmitted first, third, fifth, and so on, with sequence number `1`; and frames transmitted second, fourth, sixth, and so on, with sequence number `0`. Data frames do not carry the sequence numbers; instead, sequence numbers are returned with acknowledgement frames. Acknowledgement frames are transmitted as two characters, as shown in Figure 9.1 (note that the sequences are different for ASCII and EBCDIC).

Table 9.1: Bisync Acknowledgement Sequences

Acknowledgement	EBCDIC	ASCII
ACK `0`	*DLE* `0x70`	*DLE* `0`
ACK `1`	*DLE* `/`	*DLE* `1`

Whenever a data frame is received correctly, the receiving station changes the acknowledgement (from `0` to `1` or vice versa) and responds with the acknowledgement. Once a data frame has been properly acknowledged, the transmitting station changes the expected acknowledgement (from `0` to `1` or vice versa). A typical transmission is shown in Figure 9.10.

Data frames received in error are negatively acknowledged (using a *NAK*). The transmitting station is expected to retransmit the frame, while the receiving station is to reply with the correct acknowledgement. An example of frame retransmission is given in Figure 9.11 (in this example, assume that the acknowledgement for frame N is *ACK 1*).

If a frame is not acknowledged (i.e., the receiving station never receives the frame or the acknowledgement is lost) within 2 seconds, the transmitting station must poll the receiving station to determine the value of the last acknowledgement sent. Polling takes place using the *ENQ* (enquire) control character. There are three possible outcomes of the poll:

The correct acknowledgement is received. The frame was received correctly by the receiving station, but the acknowledgement was lost. As shown in Figure 9.12, the next frame can be

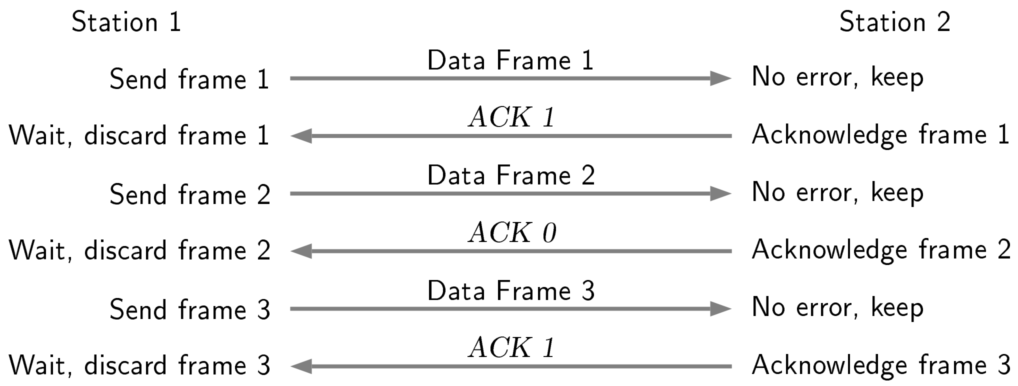


Figure 9.10: A typical, error-free Bisync sequence

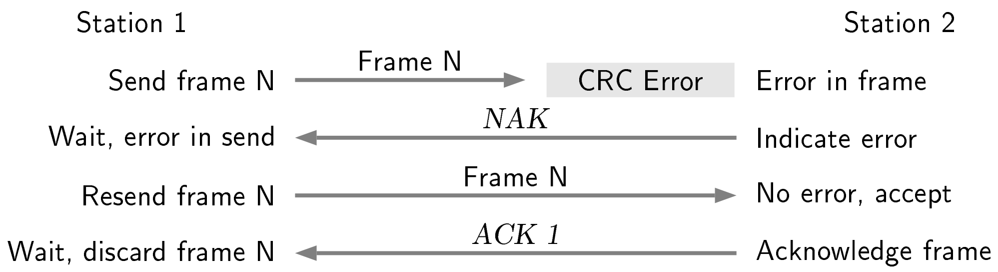


Figure 9.11: Frame transmission due to CRC error detection

transmitted at this point (assume that frame N is to be acknowledged with *ACK 0*).

The wrong acknowledgement is received. The receiving station never received the data frame and is still responding with the acknowledgement of the *last* correctly received frame. The transmitting station must retransmit the last data frame, as shown in Figure 9.13.

The receiving station never responds. It is possible that the receiving station could be momentarily off-line or simply too busy to respond to the poll. To accommodate situations such as these, the transmitting station will wait for 3 seconds and poll again. If, after three such cycles, no response is obtained, the transmitting station aborts the transmission.

There are a number of variations and exceptions to the time and the number of repeats the transmitting station performs. For example, in some installations the number of polls can be greater to accommodate transmission delays; or the time between polls can be increased.

Other responses. There are other responses allowed by Bisync. For example, should the receiving station be unable to accept more data frames because of such problems as no paper in the line printer, the communication channel can be kept active by the receiving station responding with *WACK*s (wait acknowledgement). The transmitting station continues polling (with *ENQ*s) until an acknowledgement is received.

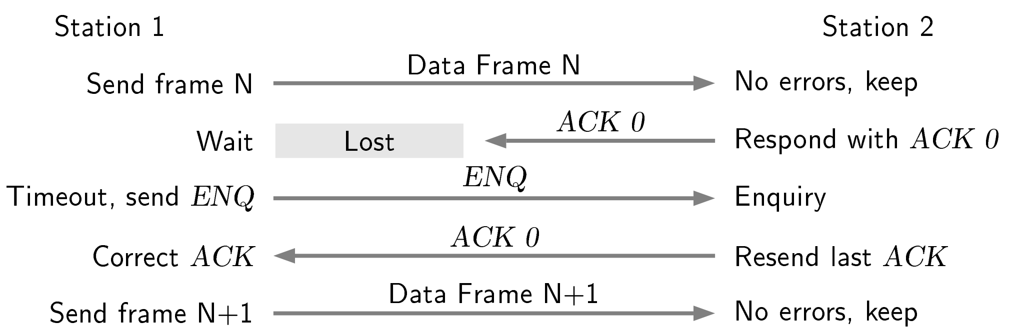


Figure 9.12: The correct acknowledgement signals transmission of the next frame

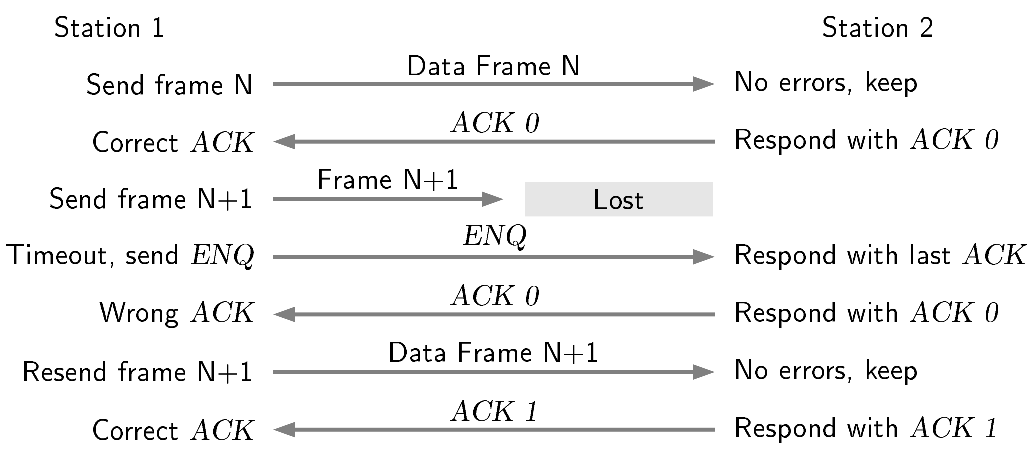


Figure 9.13: The previous acknowledgement signals a retransmission

Although there are exceptions (for example, a station with only a line printer, or a station with only a card reader), most Bisync implementations allow either station to transmit and receive information. Bisync requires the station about to transmit to send an *ENQ* character, the potential receiving station is to respond with an *ACK 0*, after which the data transfer may proceed. In Figure 9.14, station 1 is bidding for the line.

One problem to consider with Bisync implementations is: what happens if both stations attempt to transmit at the same time? From the algorithm described above, both will send *ENQ*s and wait for the response. Since only one station can transmit at any time (remember, Bisync is half-duplex), one of the two stations must 'win' the bid. The solution adopted by Bisync is to designate one station the **master** station and the other the **slave**. A slave station must defer transmission to the master station whenever a conflict occurs. When both stations bid for the channel (with *ENQ*s), the master station is 'guaranteed' to win the bid because it has a shorter polling wait period: one second for the master as opposed to three seconds for the slave.

Once a station has finished transmitting, the receiving station is signaled with an end-of-transmission character or *EOT*. At this point, both stations return to the state where either can

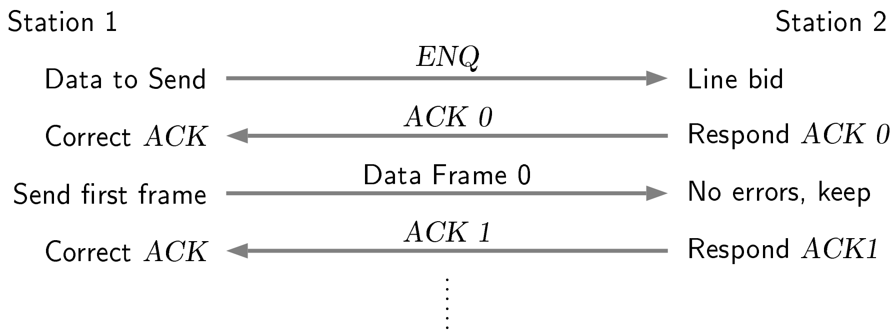


Figure 9.14: The line-bid sequence

bid for the channel. *EOT* can also be used to abort a transmission. If the transmitting station sends an *EOT*, the receiving station the returns to the line-bidding state.

Although Bisync is half-duplex, there is a provision for forcing the transmitting station to give up the line, thereby allowing the receiving station to send a message. Either station, when receiving, can respond with an *RVI* or reverse interrupt, which is equivalent to an acknowledgement. Upon receipt of the *RVI*, the transmitting station sends one more block, waits for the acknowledgement and then sends an *EOT*. The receiving station can then bid for the line.

9.3.1 The Bisync Frame

Bisync, like any synchronous protocol, requires the use of framing information to allow a receiving DTE to distinguish between data and the idle channel. Bisync uses special control characters.

Start-of-frame is indicated by either of the following control characters:

SOH Start Of Header, or

STX Start of TeXt.

End-of-frame is also a control character, represented by one of:

ITB Intermediate Transmission Block,

ETB End of Transmission Block, or

ETX End of TeXt.

ETB and *ETX* signal the receiving station that the line is to be turned-around for an acknowledgement. *ITB* on the other hand can be used to denote the end of a record; the *ITB* is followed by another record (typically, although not necessarily, starting with an *STX*). The number of *ITB* blocks allowed is implementation dependent; however, after a maximum of about four unacknowledged records, the fifth must be terminated by an *ETB* or an *ETX*. The *ETX* is a special case of

ETB, indicating that no further frames will arrive (*ETX* is often used to signal that the end-of-file has been reached).

Frames consisting of **printable** bytes only (i.e., alphanumerics, carriage return, line feed, tab, and form feed) are known as **non-transparent** frames; Figure 9.15 shows the non-transparent frame format (note, unless otherwise indicated, all frames are sent from left-to-right).

$$\begin{bmatrix} SOH \\ STX \end{bmatrix} \cdots\cdots\cdots \quad \mathsf{Data} \quad \cdots\cdots\cdots \begin{bmatrix} ITB \\ ETB \\ ETX \end{bmatrix}$$

Figure 9.15: The non-transparent frame format

A 16-bit cyclic redundancy check (CRC-16) is calculated on all the data bytes and is transmitted after the end-of-frame character. The CRC covers only the data from the start-of-frame character (*STX* or *SOH*) to the end-of-frame character (*ITB*, *ETB*, or *ETX*). A separate CRC is calculated for each frame, including *ITB* frames. The CRC associated with an *ETB* or *ETX* following a series of *ITB* blocks is generated only on the last block.

The Bisync protocol allows a limited number of control characters to be sent in non-transparent frames. This leads to another problem: how can messages, such as executable images or binary files, consisting of non-printable control characters, be transmitted? For example, in Figure 9.16 part of the data stream consists of the control character *ETB*, the receiver cannot distinguish between the *ETB* indicating the end-of-message *ETB* and the data *ETB*; meaning that the receiver will stop receiving after the first *ETB* (the *data ETB*!).

$$\begin{bmatrix} SOH \\ STX \end{bmatrix} \cdots\cdots\cdots \quad ETB \quad \cdots\cdots\cdots \begin{bmatrix} ITB \\ ETB \\ ETX \end{bmatrix}$$

Figure 9.16: Non-transparent frames cannot include certain control characters

The solution to this problem is to have the transmitting DTE inform the receiving DTE that non-printable characters are included in the transmission. A communication which permits the inclusion of control characters as data within the frame is known as a **transparent transmission**. A transparent transmission allows *any* byte to be transmitted as data. As shown in Figure 9.17, control characters in a transparent frame are prefixed with the *DLE* character.

$$DLE\begin{bmatrix} SOH \\ STX \end{bmatrix} \cdots\cdots\cdots \quad \mathsf{Data} \quad \cdots\cdots\cdots \quad DLE \begin{bmatrix} ITB \\ ETB \\ ETX \end{bmatrix}$$

Figure 9.17: The transparent frame format

Should an *ETB* or virtually any other control character be sent as data, it will accepted as data because it is *not* prefixed with a *DLE*.

Although prefixing framing control characters with *DLE* is intended to ensure that *any* character can be sent as data, problems arise should a message contain a *DLE* followed by an end-of-frame character (i.e., one of *ITB*, *ETB*, or *ETX*). As shown in Figure 9.18, the receiving DTE cannot distinguish between *DLE* followed by *ETX* sent as data or sent as an end-of-frame sequence.

$$\mathit{DLE}\begin{bmatrix}\mathit{SOH}\\ \mathit{STX}\end{bmatrix}\cdots\cdots\ \mathit{DLE\ ETX}\cdots\cdots\quad \mathit{DLE}\begin{bmatrix}\mathit{ITB}\\ \mathit{ETB}\\ \mathit{ETX}\end{bmatrix}$$

Figure 9.18: The *DLE-ETX* byte sequence cannot be sent in a frame

Since the *DLE* and the character that follows it are both data, neither can be left out of the transmission. The solution, shown in Figure 9.19 is to *prefix* any *DLE* found in a message by another *DLE*, then to transmit both *DLE*s.

$$\mathit{DLE}\begin{bmatrix}\mathit{SOH}\\ \mathit{STX}\end{bmatrix}\cdots\cdots\mathit{DLE\ DLE\ ETX}\cdots\cdots\quad \mathit{DLE}\begin{bmatrix}\mathit{ITB}\\ \mathit{ETB}\\ \mathit{ETX}\end{bmatrix}$$

Figure 9.19: All data *DLE*s must be prefixed by *DLE*

When the receiving DTE receives a *DLE* in the data stream, the DTE enters a special state and waits for one of a *DLE* (taken as data) or a control character (typically signalling the end-of-frame).

9.3.2 Synchronization

Synchronization of Bisync DTEs is achieved through the use of a series of synchronization characters (*SYN*), prefixed to the start of every frame. The *SYN* characters synchronize the transmitting and receiving clocks, thereby allowing the transmission of 'pure' data. *SYN* characters can be placed in a frame to ensure that the receiver's clock maintains synchronization (there is yet another exception: if the frame is a transparent frame, the *SYN* character must be prefixed by a *DLE*).

9.3.3 Byte-Oriented Hardware

Many manufacturers of byte-oriented synchronous hardware make the hardware support asynchronous communications as well. Hardware that supports both synchronous and asynchronous communications is known as a **USART** or **Universal Synchronous/Asynchronous Receiver/Transmitter**. The structure of the USART, shown in Figure 9.20, is similar to that of the UART.

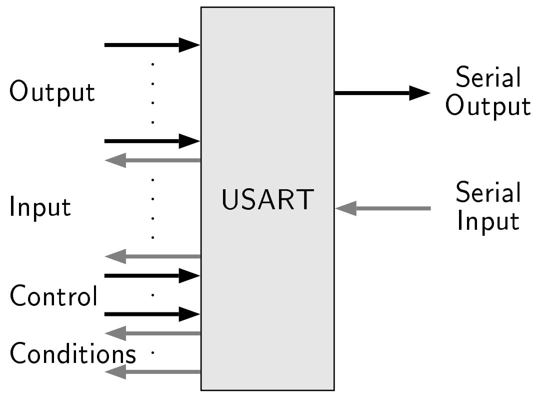


Figure 9.20: The USART

The Intel 8251A is used on the IBM PC to support the Bisync protocol. Functionally, the 8251A USART is very similar to the 8250 UART in that the baud rate, character length, stop bits, and parity can be selected under program control. The program can also choose asynchronous or synchronous operation. (The 8251A supplied with the PC operates in synchronous mode only.)

When transmitting a Bisync frame, the software is required to format the entire frame (including the *SYN* characters and the CRC). The frame is transmitted one byte at-a-time. The software is expected to supply data to the 8251A at such a rate that the transmit buffer is never empty. Should the transmit buffer become empty, the 8251A starts to idle with *SYN* characters. This can clearly lead to problems in transparent mode, since *SYN* characters sent as timing characters are to be prefixed with *DLE*.

If a Bisync station is to receive a frame, the 8251A must be put into *hunt* mode, scanning the channel for one or more *SYN* characters. The software can also specify whether the search is for one or two *SYN* characters. The bytes are returned to the processor as they are received.

9.4 Summary

This Chapter has examined the stop-and-wait protocol, the simplest of the ARQ protocols. Stop-and-wait, as the name suggests, requires the transmitting DTE to send a message and then wait for a response. The most basic implementation of the stop-and-wait protocol expects the receiving DTE to respond after it receives a message with an acknowledgement. However, since the original message or the acknowledgement can be lost, it is necessary to introduce a new element into the design of the protocol, notably *time*.

If the waiting period is timed (i.e., there is an upper limit on the time the transmitter will wait before retransmitting the message), there is no possibility of the transmitter *hanging* while waiting for a response from the destination DTE.

By incorporating time into the protocol, a new problem has been introduce, that is, the destination DTE could receive the same message more than once. To overcome this problem, it is necessary to incorporate **sequence numbers** into the message stream. Each message that is sent is uniquely identified, meaning that the destination DTE can distinguish between retransmissions and new messages.

The Chapter also introduced a byte-oriented protocol, Bisync, as an example of a commercial version of the stop-and-wait protocol. Bisync differs from 'traditional' view of the stop-and-wait protocol in that the acknowledgements, rather than the messages, contain the sequence number.

9.5 Exercises

1. Show, by means of illustrations or examples, why throughput can be improved in the stop-and-wait protocol by using a *NAK* rather than a time out.

2. In Bisync, why does a timing *SYN* require a *DLE* prefix when transmitting transparently? What happens if the *DLE* is omitted?

3. Using the state machine descriptions for the stop-and-wait protocol, show how the protocol recovers from the error shown in Figure 9.21.

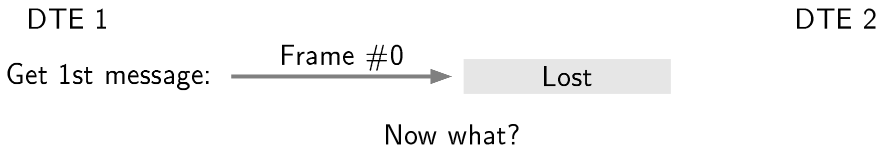


Figure 9.21: How does the stop-and-wait protocol recover?

4. Why is (or isn't) it necessary to use prefix bytes when transmitting messages on the Ethernet?

5. Explain why sequence numbers are a form of synchronization.

6. Implement a version of the stop-and-wait protocol to run atop an Ethernet.

Chapter 10

The Go-Back-N Protocol

10.1 Introduction

A closer examination of the stop-and-wait protocol reveals that it is essentially a half-duplex protocol. For example, after the transmitting entity sends a frame, the channel is turned-around to allow the receiving entity the opportunity to respond with an acknowledgement. If the connection between the two entities is full-duplex, then using the channel in a half-duplex fashion is both a waste of time and bandwidth. The question now is – how can a full-duplex communication channel be used to increase throughput?

10.2 The Go-Back-N Protocol

One potential way of increasing throughput is to allow the transmitting DTE to send a number of frames while simultaneously receiving responses. Figure 10.1 is an example of such a scenario (the channel is assumed to be full-duplex since frames and acknowledgements are transmitted simultaneously; the cross-over of a frame and an acknowledgement indicates simultaneous transmission and reception).

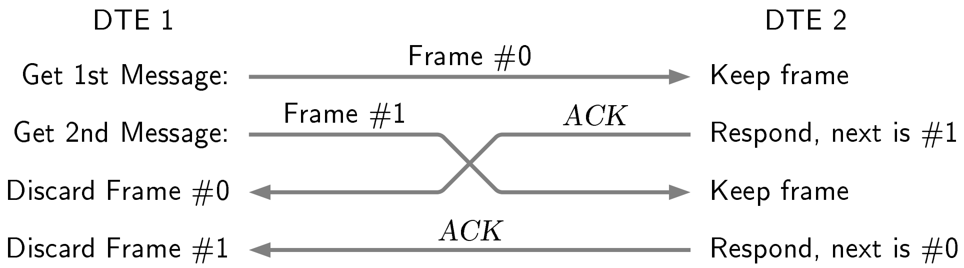


Figure 10.1: The simultaneous transmission of frames and acknowledgements

In Figure 10.1, DTE 1 discards frame #0 upon receipt of the first acknowledgement and frame #1 upon receipt of the second acknowledgement. This approach will work correctly as long as no frames are lost, as Figure 10.2 illustrates.

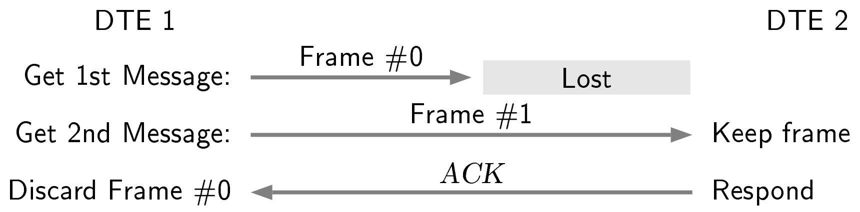


Figure 10.2: The transmitter discarding the wrong frame

In Figure 10.2, DTE 1 receives an acknowledgement and proceeds to discard frame #0: something that shouldn't happen because the first frame has been lost, never reaching DTE 2; the acknowledgement that DTE 1 receives is for frame #1. What is needed is a mechanism whereby the acknowledgement can indicate the specific frame being acknowledged, rather than a single acknowledgement being used to acknowledge all frames.

A commonly adopted solution is to append an identifier to the acknowledgement (in much the same way as the identifier is appended to the frame). The identifier indicates the sequence number of the *next* frame that the receiving DTE expects to receive. This is analogous to sending letters to someone, then telephoning and asking which letters they have received: the person might reply that they had received the first and second letters but are still awaiting the third; or if no letters had been received, the response might be that they were still waiting for the first letter. In the case of the transmitting DTEs, a typical transmission scenario could be as shown in Figure 10.3 (note that the acknowledgements indicate the sequence number of the *next* expected frame).

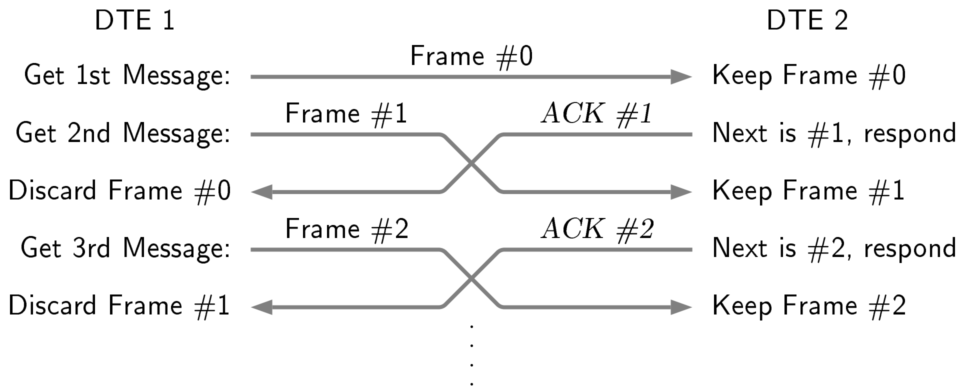


Figure 10.3: Adding a sequence number to the acknowledgement

Upon receipt of an acknowledgement, the transmitting DTE can discard those frames with sequence numbers up to (but not including) the sequence number sent with the acknowledgement, since the acknowledgement is indicating the *next* frame expected by the receiver. For example, if an acknowledgement with sequence number 3 is received, frames #0, #1, and #2 can be discarded (if this hasn't already been done).

Both the transmitting and receiving DTEs must maintain sufficient information to determine the current state of the communication (i.e., the number of the next frame to transmit, the number of the next expected frame, and so on). This information is represented as a pair of **state variables**:

Vs: The **send state variable** (sometimes written as **V(S)**) is the sequence number of the next frame to be transmitted; the initial value of Vs is typically 0. Vs is incremented as each new frame is transmitted.

Vr: The **receive state variable** (also written as **V(R)**) is the sequence number that is next expected by the receiver; Vr has an initial value identical to that of Vs (i.e., 0). Receipt of a frame with a frame number equal to Vr causes Vr to be incremented.

The sequence number associated with each frame is referred to as the **send sequence number** and is written as **Ns** (or **N(S)**). The transmitter assigns the value of Vs to Ns; once assigned, the value of Vs is incremented.

The sequence number appended to the acknowledgement, the **receive sequence number**, **Nr** (or **N(R)**), is the value of Vr after the frame has been accepted by the receiver. To the receiving DTE, Nr is the value of the next expected sequence number; while to the transmitting DTE, all frames with sequence numbers less than Nr have been accepted by the receiving DTE.

As an example, Figure 10.4 shows the same sequence of events that took place in Figure 10.3, this time using the state variables Vs and Vr, and the sequence numbers Ns and Nr. Prior to the start of a communication, both the transmitting and receiving DTEs must ensure that their values of Vs and Vr are identical (in this and subsequent examples, both are assigned the value 0). DTE 1 sends the first frame which contains message 1 and Ns, which has been assigned the current value of Vs, 0 (the assignment written as Ns←Vs). Vs is incremented by 1 (this is denoted by Vs++); DTE 1 keeps the frame until its reception is acknowledged. Upon receipt of the first frame, DTE 2 compares the value of Vr with the value of Ns taken from the frame; since both are equal (indicated by Ns=Vr), the value of Vr is incremented (from 0 to 1) and the contents of the frame are kept.

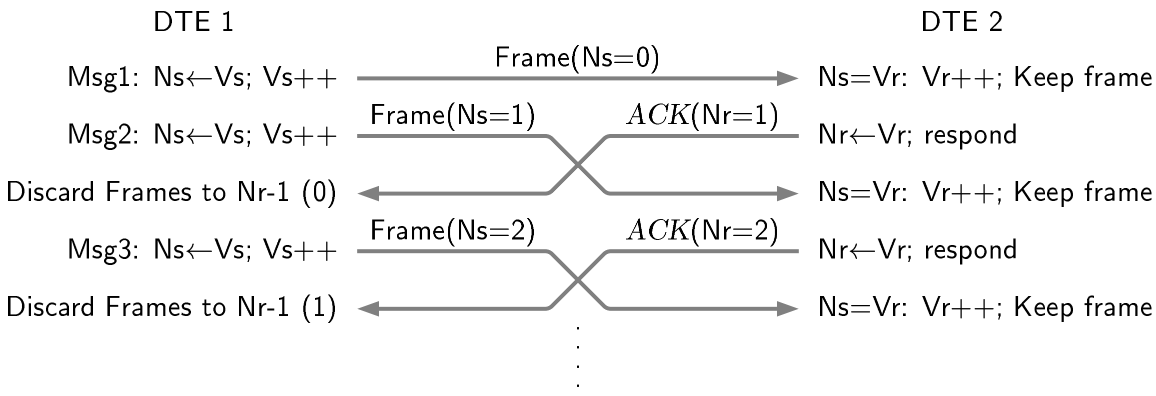


Figure 10.4: Using the state variables (Vs and Vr) and sequence numbers (Ns and Nr)

DTE 2 responds with an acknowledgement; the value of Nr is obtained from the current value of Vr (in this case, 1). When DTE 1 receives the acknowledgement, all frames with sequence numbers less than Nr are discarded (i.e, frame Ns=0). This cycle continues until DTE 1 has no more messages to transmit.

Including sequence numbers with acknowledgements ensures that the transmitting DTE 'knows' which pending frames to discard. However, there are three other possible error conditions that must also be considered: the loss of an acknowledgement, the loss of a frame, and the lack of a response from the receiving DTE.

If an acknowledgement is lost, the transmitting DTE continues to transmit (since there is no indication that the acknowledgement was lost) and when the next acknowledgement is received, the transmitting DTE can determine which frames are to be discarded. This problem is illustrated in Figure 10.5 (both Vs and Vr are assigned 0 initially).

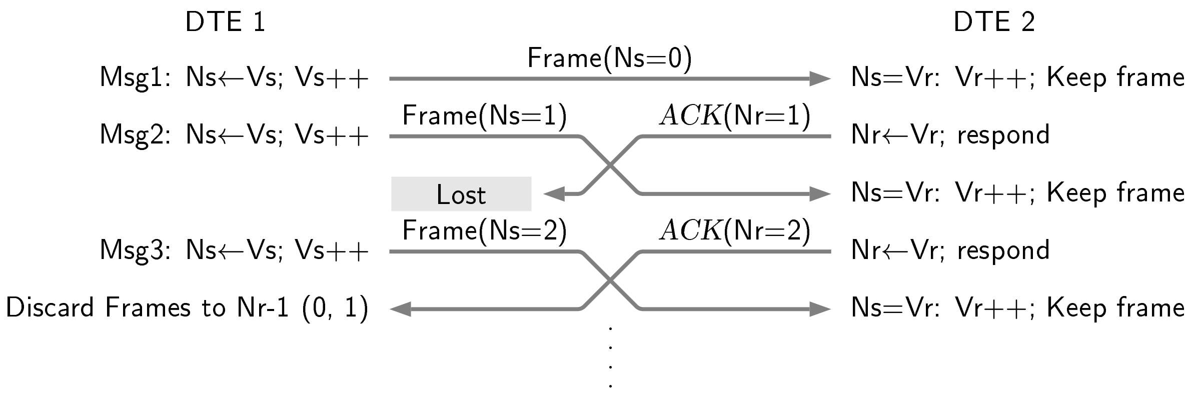


Figure 10.5: The receive sequence number (Ns) includes all previous sequence numbers

In Figure 10.5, the acknowledgement for frame (Nr=0) (i.e., *ACK*(Nr=1)) is lost but DTE 1 continues to transmit frames. Upon receiving *ACK*(Ns=2), DTE 1 can dispose of both frame Ns=0 and Ns=1, since the acknowledgement indicates that *all* frames with sequence numbers less than Nr=2 have been accepted.

The second error condition that must be considered is how to handle the loss of a frame. If a DTE receives a frame with an invalid or out-of-sequence value of Ns (identifiable because Ns does not equal Vr), the receiving DTE should respond with an indication that the frame's sequence number is invalid, thereby permitting the transmitting DTE to restart its transmission with the correct frame. The transmitting DTE is to resume with the frame that the receiving DTE next expects. By responding with a negative acknowledgement (and the sequence number of the next expected frame, that is, Vr), the transmitting DTE can distinguish between a normal acknowledgement and an error condition. In Figure 10.6, frame (Nr=0) is lost, causing DTE 2 to respond with a request for frame (Nr=0).

The negative acknowledgement received by DTE 1 indicates that DTE 2 is still expecting frame (Ns=0), even though frames Ns=0, Ns=1, and Ns=2 have been sent. Since DTE 1 has no indication as to which frames have been received correctly and which are in error, the worst is assumed and transmission resumes from the specified frame (in this case, frame (Ns=0)). Note that during the

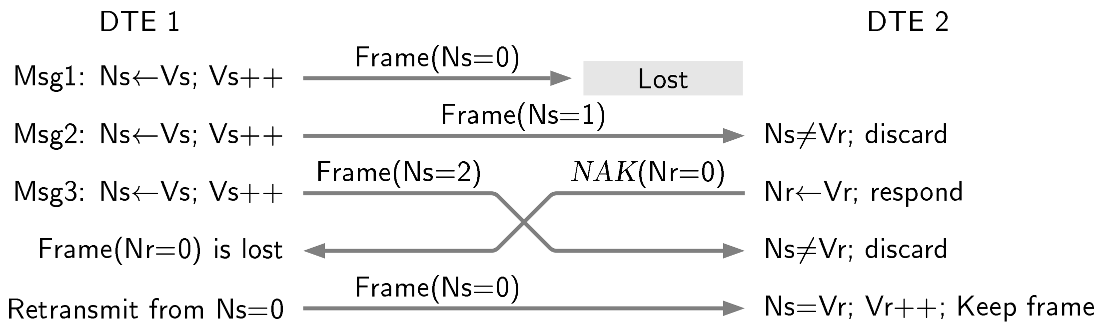


Figure 10.6: The retransmission of a lost frame

retransmission, the value of Ns in each packet remains unchanged and Vs is not incremented.

Protocols that expect the transmitting DTE to retransmit all frames from the sequence number indicated by the receiving DTE are known as **Go-Back-N** protocols. In a Go-Back-N protocol, all frames that are received with an invalid or unexpected value of Ns are discarded.

The third error condition occurs when the transmitting DTE sends a frame and does not receive a response. The solution adopted in the stop-and-wait protocol can also be applied to the Go-Back-N protocol: each frame that is sent must be acknowledged within a certain time period, otherwise the frame is again sent. Since there may be several frames awaiting acknowledgement, each frame is associated with a unique timer; as acknowledgements are received, the timers are turned off and the frames discarded.

There are two different scenarios where frame loss may occur. In the first, the frame does not arrive at the receiving DTE; this is illustrated in Figure 10.7. When the timer associated with frame (Ns=0) expires, DTE 1 retransmits the frame, including its original value of Ns; Vs remains unchanged since this is a retransmission.

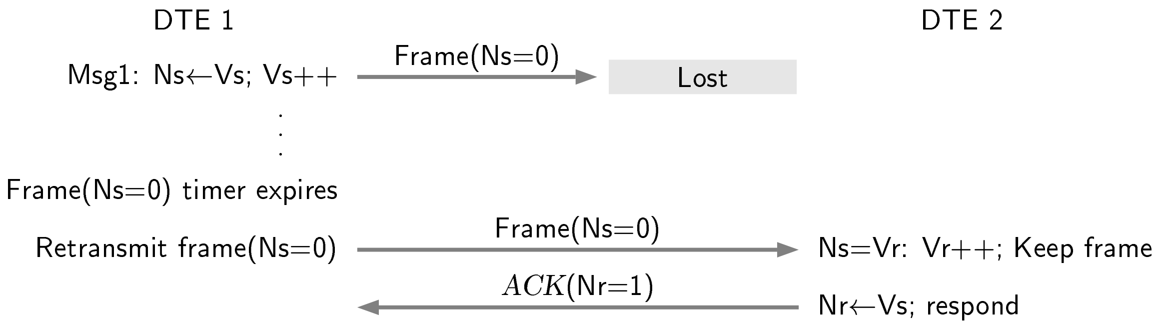


Figure 10.7: Using a timeout to retransmit a lost frame

The second scenario occurs when the frame has arrived and is acknowledged by the receiving DTE, but the acknowledgement is not received by the transmitter. In this situation, when the timelimit expires, the frame is retransmitted but discarded by the receiving DTE. As shown in Figure 10.8, DTE 2 responds with an acknowledgement indicating the next expected sequence number. Note that the transmitting DTE (i.e., DTE 1) cannot distinguish between the two scenarios.

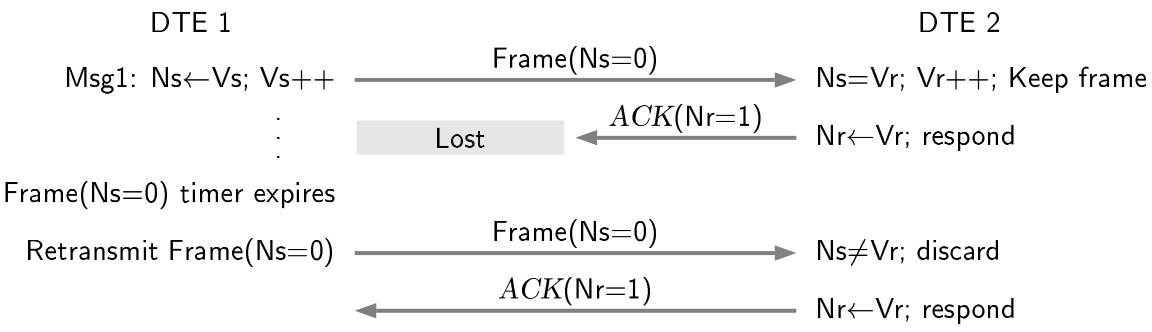


Figure 10.8: Acknowledging (and discarding) a duplicate frame

To avoid unnecessary retransmissions, the timer associated with the acknowledged frame is turned off as soon as the correct acknowledgement is received. Should several frames be waiting for acknowledgements and the incoming acknowledgement indicates that all of the frames have been accepted, each timer is turned off and the associated frame discarded.

One of the arguments for the Go-Back-N protocol is that it can be implemented on a full-duplex channel, thereby increasing channel throughput. However, there are two competing objectives which must be considered when increasing channel throughput:

- first, the transmitting DTE should be operated in such a way to keep the channel 'full', and
- second, acknowledgements must be returned at regular intervals in order to minimize the number of retransmissions required should one of the previous frames be received in error.

In the stop-and-wait protocol, two sequence numbers are needed (one for the previous frame and one for the current frame). The number of sequence numbers required in a Go-Back-N protocol is not so readily apparent. If only two sequence numbers are used (i.e., 0 and 1, as in stop-and-wait), then the number of frames that the transmitting DTE can send without receiving an acknowledgement is one, as Figure 10.9 illustrates (Note that the increment operator, ++, is assumed to operate modulo-2, meaning that the possible values for Vs and Vr in this example are 0 and 1).

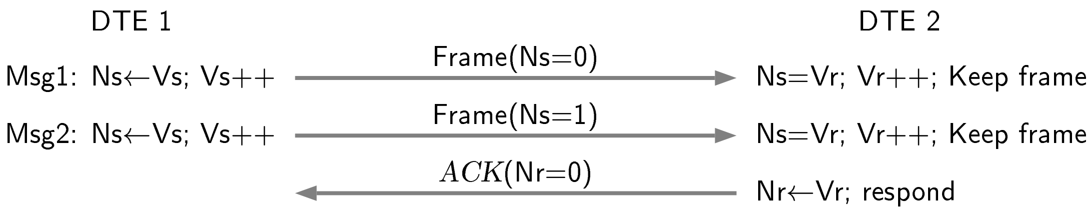


Figure 10.9: A problem with sequence numbers

In Figure 10.9, the receipt of *ACK*(Nr=0) (by DTE 1) can mean *either*:

1. two frames were received (Ns=0 and Ns=1), and DTE 2 is expecting a third frame (to be sent with sequence number Ns=0), *or*
2. DTE 2 is still expecting the first frame (that is, frame Ns=0 was lost), meaning that both frames must be retransmitted.

Since it is necessary for protocols to be well defined and unambiguous, then the number of unacknowledged frames must be less than the total number of possible sequence numbers (otherwise problems such as those described above can occur). In short, if there are N sequence numbers available, then at most $N-1$ frames can be unacknowledged.

Consider, for example, a protocol that supports three sequence numbers (i.e., 0, 1, and 2) and that the transmitting DTE must wait for an acknowledgement after sending two frames (each with a unique sequence number). Upon receipt of the acknowledgement, further frames may be transmitted (or retransmitted, should a negative acknowledgement be received). Figure 10.10 illustrates a typical frame/acknowledgement cycle ('Next' denotes the next pair of send sequence numbers that DTE 2 is expecting).

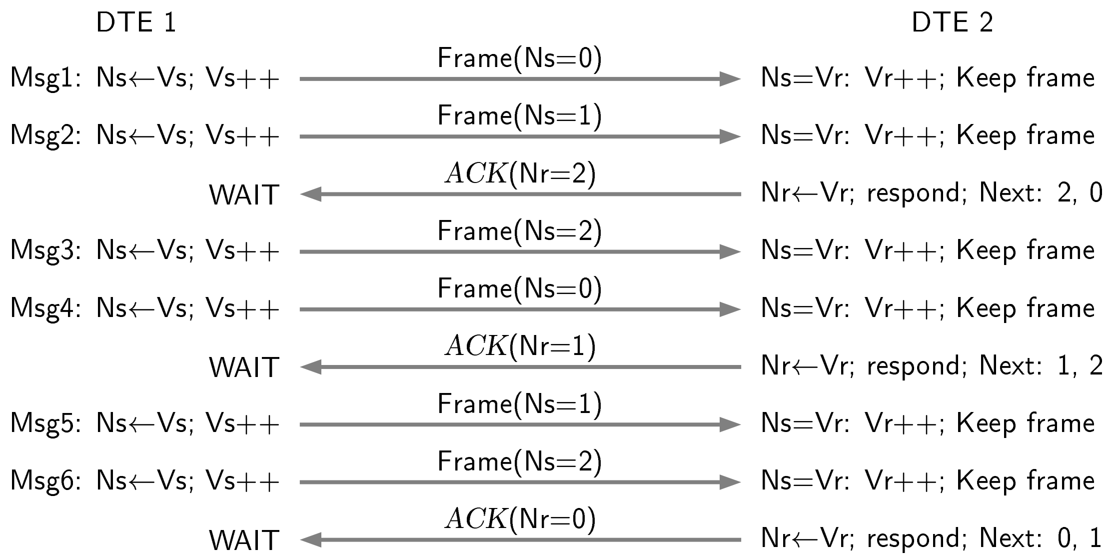


Figure 10.10: Go-Back-N using three sequence numbers

Protocols that allow a transmitting DTE to send a number of frames before the receiving DTE responds are referred to as **sliding window protocols**. The term 'sliding window' can have a number of connotations:

- as each message is sent, the **transmission window** (i.e., the number of frames that can be sent before an acknowledgement is required) is decreased in size by one; transmissions continue until the window has a size of zero, at which point transmissions cease. Whenever an acknowledgement is received, the transmission window is opened (the exact amount depends upon the value of Nr returned by the receiver); transmissions can resume, decreasing the window size.

In the above example, DTE 1 can send two frames before waiting for a reply, it therefore has a transmission window of size 2. Each acknowledgement received indicates that the two messages sent were received correctly, therefore fully opening the transmission window.

- the window 'slides' forward, indicating the sequence numbers to be used for the next set of message(s) to be sent. In the above example, three sequence numbers exist (0, 1, and 2); since they are always being re-used, the list of available sequence numbers can be considered as a long repeating sequence of the same three values:

  0 1 2 0 1 2 0 1 2 ...

  At any moment, two of the sequence numbers can be in use; as acknowledgements are received, the window slides over the next set of possible sequence numbers. For example, after the initial transmission, the window sits over 0 and 1:

  $\boxed{0\ 1}$ 2 0 1 2 0 1 2 ...

  After the first acknowledgment, the window slides over the next two sequence numbers:

  0 1 $\boxed{2\ 0}$ 1 2 0 1 2 ...

The number of frames that can be sent without being acknowledged is known as the **window size**. Initially, the transmission window is set to the value of the window size. As each frame is sent, the transmission window is decreased until it reaches zero (i.e., the window is closed), at which point no further frames can be sent. In the example described above, the window size is two.

To maximize throughput, the transmission window should never be allowed to close, which means that the receiving DTE must respond with sufficient acknowledgements to ensure an uninterrupted flow of frames. This can be achieved in a number of ways: every frame can be acknowledged, or the window size can be made large enough to require fewer acknowledgements. In Figure 10.11, there are five possible sequence numbers (giving a maximum transmission window size of four). By responding after every third frame, the receiving DTE can keep the window open (if it waited until the fourth frame, the transmission window would have closed).

As with so many things in data communications, there are tradeoffs that must be considered when specifying a window size. For example, the larger the window, the more pending-frame space required (since a frame cannot be discarded until it has been acknowledged); furthermore, if a frame is *NAK*ed, all subsequent pending frames must be retransmitted. With these things in mind, many systems use a window size of two, which is similar to the double buffering used in the UART: one frame is being transmitted while the other is being prepared.

The number of frames sent by the transmitting DTE can be controlled by the receiving DTE imposing **flow control**. Flow control restricts the rate at which frames are acknowledged (in the most extreme case, if the receiving DTE does not respond at all, the transmitting DTE is forced to retransmit all pending frames, as their timers expire). As shown in Figure 10.12, given a window size of two, the receiving DTE could turn the Go-Back-N protocol into a stop-and-wait protocol (assume three sequence numbers).

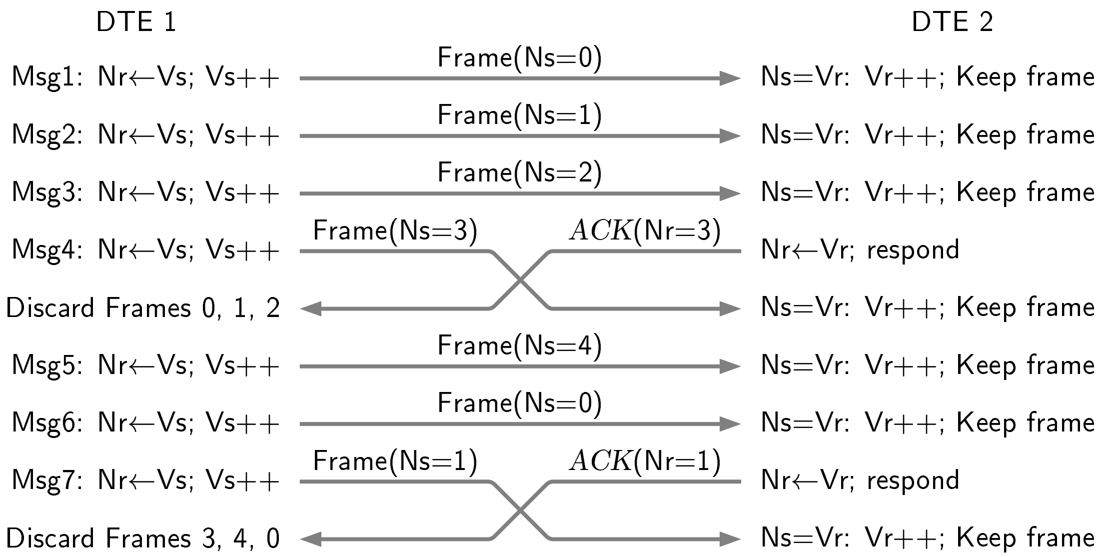


Figure 10.11: With a window size of four, DTE 2 can respond every three frames

In Figure 10.12, the DTE 2 never allows DTE 1 to send more than one frame before the transmission window closes. This effect is achieved by having DTE 2 responding with an acknowledgement Nr value equal to the value of Ns taken from the most recently received frame (as opposed to the value of Vr, the expected sequence number of the next frame). For example, after receiving frames Ns=0 and Ns=1, instead of responding with Nr=2 (an indication that frames Ns=0 and Ns=1 were received correctly), the receiving DTE responds with Nr=1, thereby allowing the window to move forward one sequence number only, as shown in Figure 10.13. To avoid DTE 1 retransmitting any pending frames, DTE 2's acknowledgements must arrive before each pending frame's timer expires.

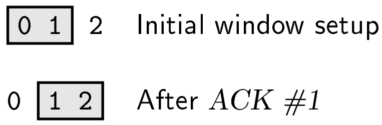


Figure 10.13: The transmission window for Figure 10.12

The number of sequence numbers available in any protocol is usually a power of two (if the sequence numbers are sent as a binary number) and depends upon the protocol. Most protocols vary the number of possible sequence numbers from 2 (a one bit sequence number, 2^1) to 8 (a three bit sequence number, 2^3). In situations where there may be long transmission delays, up to 128 sequence numbers can be specified (a seven bit sequence number, 2^7).

Until now, data transfer has been half-duplex; that is, one DTE has been transmitting frames and the other has been receiving them. Should both DTEs have information to exchange, the data transfer methods examined thus far force one station to wait until the other has completed its transfer. Since the communication channel is full-duplex, it would appear that at the channel level, at least, there is no reason why information cannot be sent in both directions simultaneously using

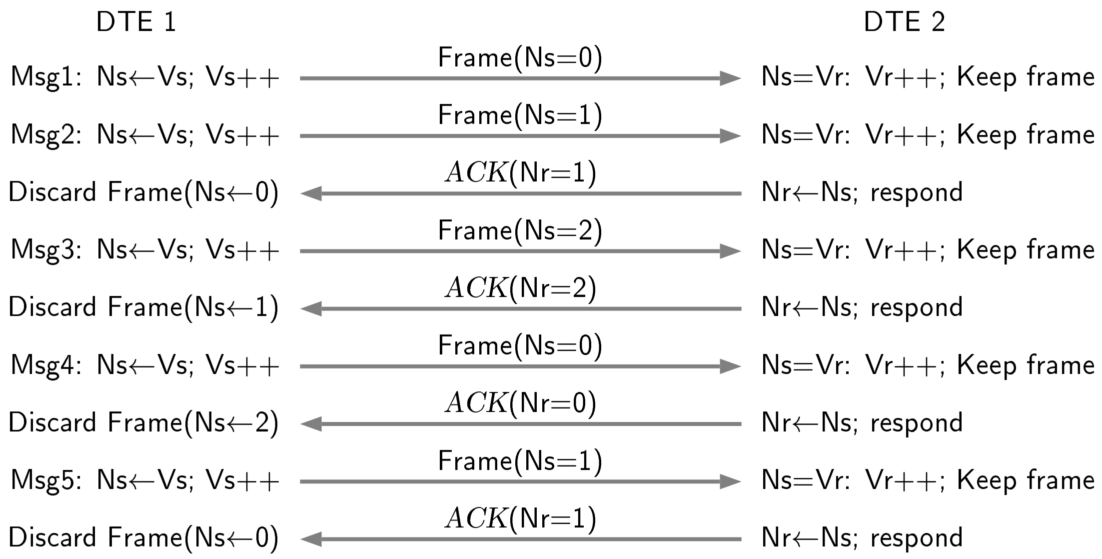


Figure 10.12: A Go-Back-N implementation of stop-and-wait

a **bidirectional data transfer**.

To accomplish this transfer, the change required to the Go-Back-N protocol is actually quite simple: acknowledgements must be sent with the frames using a technique known as **piggybacking**. When piggybacking acknowledgements, each frame is sent with its own send sequence number (i.e., Ns) *and* the sequence number of the frame next expected to be sent by the destination DTE (i.e., Nr, obtained from the DTE's receive state variable, Vr). The frame still carries a message.

When a DTE receives a frame, it must check the values of both Ns and Nr:

Ns. As before, the value of Ns is compared to the DTE's receive state variable Vr. If the two are equal, the frame is accepted and Vr is incremented. However, if the values are not equal, the frame is discarded.

Nr. The value of Nr is an acknowledgement, informing the local DTE of the frames that have been accepted by the remote DTE. A value of N acknowledges all frames up to frame $N-1$; the acknowledged frames can be discarded by the local DTE.

A possible exchange between two DTEs performing a bidirectional transfer is presented in Figure 10.14. In this example, DTE 2, rather than responding with an acknowledgement, responds with a frame containing the value of Nr=1, indicating that all frames with sequence numbers less than 1 have been acknowledged. Since there is only one frame format, the original frame sent from DTE 1 has a value of Nr=0.

If a station does not have information to send but has a frame to acknowledge, it responds with an acknowledgement rather than a frame. The acknowledgement contains the value of the station's next expected sequence number (i.e., Nr). For example, in Figure 10.15 DTE 2 has a single frame to transmit; this frame can be used to carry the value of Nr, all subsequent responses must be

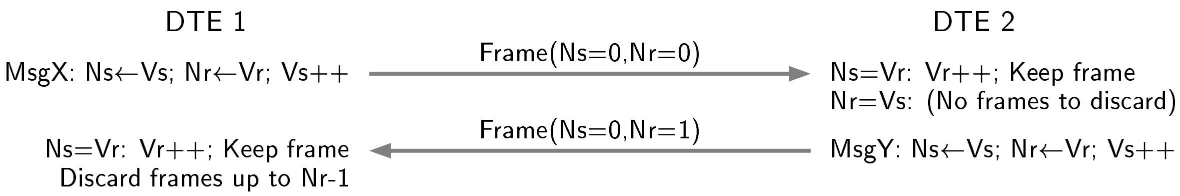


Figure 10.14: Bidirectional transfer with piggy-backed acknowledgements

sent as acknowledgements. Whenever a frame or an acknowledgement is received, the value of Nr is examined: all pending frames with Ns values less than that of Nr are discarded. Furthermore, the value of Nr sent in the frame remains constant until the value of Vr changes. Note that in the Go-Back-N protocol, only frames are acknowledged, acknowledgements are not.

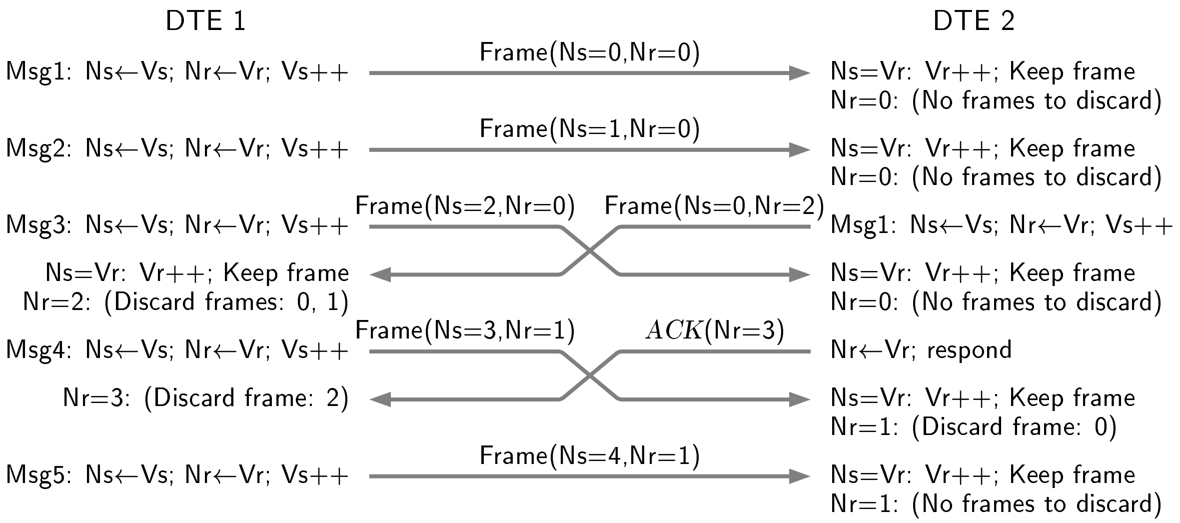


Figure 10.15: Mixing frames and acknowledgements

If a frame is lost or received in error, the same rules apply in a bidirectional transfer as in a unidirectional half-duplex: the receiving DTE must inform the transmitting DTE of the error with a *NAK*; or if frames have not been received, the transmitting DTE should timeout and retransmit the frames not yet acknowledged. For example, a frame received out of sequence (due to the loss of a previous one), should result in a negative acknowledgement, as illustrated in Figure 10.16. Note that when the frame is being retransmitted, the value of Ns cannot change (since this is the frame's send sequence number); however, the value of Nr can change if any frames are received and must be acknowledged.

Similarly, unacknowledged frames should also be retransmitted once the timer associated with each frame has expired; in Figure 10.17 two frames are lost but are retransmitted once the timer has expired.

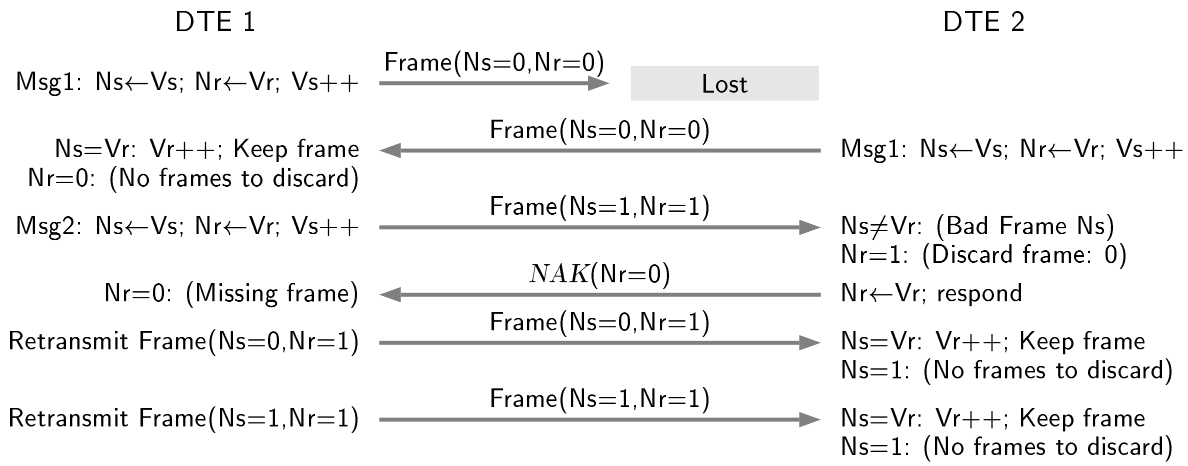


Figure 10.16: Error handling in a bidirectional transfer

10.3 Bit-Oriented Synchronous Communications

Byte-oriented synchronous communications, such as those described above for Bisync, although still in widespread use, suffer from a number of drawbacks, including:

- two frame structures are required: one for transparent data and the other for non-transparent data. In fact, there are numerous exceptions when dealing with the two different frame structures, suggesting that Bisync is actually two protocols rather than one.
- using a *DLE* to signal that the next byte should be treated as data is an additional overhead both in terms of processing (adding and removing the *DLE*) and throughput (the transmission of 16-bits of information to represent 8-bits).

Both of these problems can be overcome if *bit* rather than byte-oriented communications are used. A good example of a bit-oriented synchronous protocol is IBM's Synchronous Data Link Control or SDLC.

In SDLC, all frames (containing information or acknowledgements) have the same structure, shown in Figure 10.18.

Flag	Address	Control	Information	FCS	Flag

Figure 10.18: The fields of an SDLC frame

The fields in the SDLC frame are defined as follows:

Flag. The Flag byte delimits the frame and is used to synchronize the two stations. Both the leading and trailing Flag bytes have the same pattern: `01111110`. SDLC allows multiple frames to be sent with a single Flag byte separating each frame.

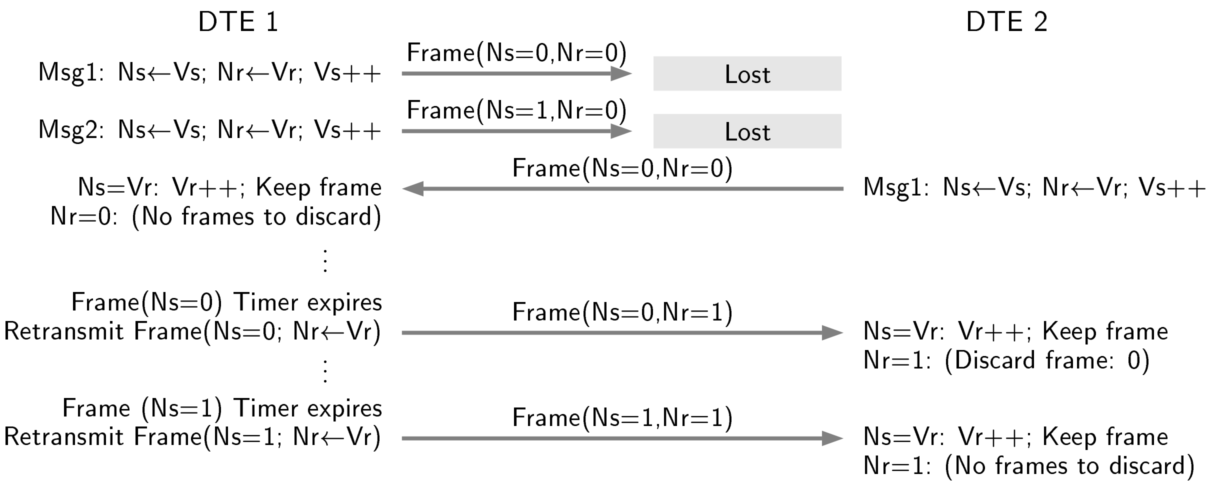


Figure 10.17: Frame retransmission after timeout

Address. The Address field denotes the intended destination of the message (if the frame is sent by the **Primary** station) or the address of the station sending the frame (if the frame is sent by the **Secondary** station). The Address field is intended to allow several Secondary stations to share a single communication channel, controlled by a single Primary station through the use of polling.

Control. The Control field is one byte long and defines the function of the frame. A frame supports one of three formats (defining the frame's function):

Information Transfer Format. The information transfer format Control field signals the receiving station that the frame is an **Information Frame**, and contains one or more bytes of data. The format of the information transfer Control field is as follows:

0	Ns (3 bits)	P/F bit	Nr (3 bits)

Ns is the send count, which is the sequence number of this (information) frame. Ns is applicable only to information frames.

Nr is the receive count, indicating the sequence number of the next expected information frame from the station to which this frame is destined.

Supervisory Format. The Supervisory frame is intended to assist in the orderly flow of Information frames between stations. The Supervisory frame Control byte has the following format:

1	0	Code (2 bits)	P/F bit	Nr (3 bits)

Code is one of the following:

Ready-to-Receive (00). Ready-to-Receive or RR is an acknowledgement; it signals that the next expected frame is Nr.

Receive-Not-Ready (01). Receive-Not-Ready or RNR is both an acknowledgement (the next expected frame is Nr) and an indication to the transmitting station that there is a temporary problem at the receiving station.

Reject (10). Reject or REJ is a negative acknowledgement, requesting the transmitter to start transmission from frame Nr.

Nr is the receive count, indicating the sequence number of the next expected information frame.

Unnumbered Format. Unnumbered frames are used to convey information specific to the function of the communication channel connecting the various stations. For example, the Primary station can initialize Secondary stations through the use of Unnumbered frames and stations can report their status in an Unnumbered frame. The format of an Unnumbered Control frame field is as follows:

1	1	Code (2 bits)	P/F bit	Code (3 bits)

Code values include:

UI (Unnumbered Information). The frame contains data in the Information field.

DISC (Disconnect). The Primary station is disconnecting the Secondary station specified in the Address field.

UA (Unnumbered acknowledgement). An affirmative response to a command such as DISC.

The P/F (Poll/Final) bit is common to all Control field formats, its interpretation is given in Table 10.1.

Table 10.1: The interpretation of the P/F bit

P/F bit	Transmitter	Interpretation
Set	Primary (to Secondary)	Secondary is to start transmission.
Set	Secondary (to Primary)	Secondary has finished transmission.
Clear	Either	A transmission is in progress.

Information. The Information field contains data being sent from one station to another. If the frame is an unnumbered frame, the Information field may be present, carrying information specific to the SDLC protocol such as a test pattern or station identification data. Information frames carrying data for applications on the destination DTE use the Information field; there is no specified length of the Information field, but it must be a multiple of 8 bits. Supervisory frames do not have an Information field.

FCS. The FCS is the Frame Check Sequence (another term for CRC). The FCS is calculated using CRC-CCITT and covers the entire frame, excluding the Flag bytes. The FCS is generated by the transmitting station and placed after the Control or Information field (the Information field is optional in some frames); the receiving station calculates its own version of the FCS until the trailing Flag pattern is detected. The result of the receiving station's calculation should be `0xF0B8`.

SDLC uses a sliding window protocol with a total of eight possible sequence numbers; the window size is application dependent, with the maximum size being seven. The section on the Go-Back-N protocol describes the basic functions of SDLC.

10.3.1 Transparency

To be truly useful, a protocol must be able to support transparent data, thereby allowing any bit pattern to be transmitted. In byte-oriented protocols such as Bisync, a special 'escape' byte (*DLE*) allows the receiving DTE to distinguish between control bytes sent as data and control bytes used for control sequences. As well, the 'escape' byte requires a set of special rules to handle its own inclusion.

In bit-oriented protocols such as SDLC, the problem of transparency still remains, since the sequence `01111110` delimits the end-of-frame, meaning that a byte sent with the value `01111110` will cause the receiving DTE to stop receiving. The solution is to use a technique known as **bit stuffing** whereby the transmitting hardware inserts a `0` bit after any sequence of five `1`s (except when the end-of-frame sequence is sent).

The receiving hardware removes the extra bit using the following algorithm:

1. If a sequence of five `1`s has been received (i.e., `11111`), then wait for the next bit.
2. If the bit is a `0`, discard it and continue assembling the stream of bits into bytes.
3. If the bit is a `1`, keep it – this should be the end-of-frame sequence.
4. Reset the bit-stuffing sequence count and resume the count when the next `1` is received.

The following examples illustrate how bit stuffing works (`0` bits that are inserted are enclosed as $\boxed{0}$ and the information received has the inserted bit removed):

Data to send:	0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0
Insertion at:	
After insertion:	0 1 1 1 1 1 $\boxed{0}$ 1 1 1 1 1 $\boxed{0}$ 1 1 0 1 1 1 1 1 $\boxed{0}$ 1 0
Data received:	0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0

10.3.2 Bit-Oriented Hardware

Intel manufactures the Intel 8273, the PC's programmable SDLC communications adapter. The 8273 performs a number of tasks specific to SDLC, thereby reducing processor overheads. For example, if a frame is to be transmitted, the 8273 transmits the Flag byte followed by the Address and Control fields. If an Information field is to be transmitted, it is supplied to the 8273 using direct memory access (DMA), thereby reducing processor overheads. Lastly, the FCS and final Flag byte are sent by the 8273. Incoming frames are treated in a similar fashion, with the Address and Control fields being written to hardware registers and the Information field transferred to memory without processor intervention. The FCS is checked as well. The processor is interrupted upon reception of a frame.

10.4 Summary

The stop-and-wait protocol is subject to a number of limitations, the most serious is its throughput: each message that is sent causes the transmitter to wait until a response is received or a timeout occurs. The **go-back-N** protocol overcomes this limitation by allowing the transmitting DTE to send multiple messages before an acknowledgement is required.

The go-back-N protocol is a **sliding window** protocol. The transmitting DTE has a fixed number of messages that can be sent; as each message is sent, the number decreases (the window decreases in size). As acknowledgements are received, the number increases (the window increases in size). Ideally, the flow of acknowledgements will ensure that the window never closes, meaning that there is a continuous flow of data from the transmitting DTE to the destination DTE.

The go-back-N protocol overcomes another limitation associated with the stop-and-wait protocol, notably that it is a **full-duplex** protocol, allowing both DTEs involved in the communication to exchange messages simulataneously. This is done by **piggybacking** acknowledgements on message frames.

One implementation of the go-back-N protocol is IBM's SDLC (Synchronous Data Link Control). The SDLC implementation of the go-back-N protocol is **bit-oriented**, meaning that transparency is supported at the bit, rather than the byte level. Support for bit-oriented protocols uses a technique known as **bit-stuffing** to support transparency.

IBM's version of the go-back-N protocol has been widely adopted by many organizations for many applications. The ISO version of SDLC, known as HDLC (High-Level Data Link Control) is used as a link control protocol in many wide area networks (see Chapter 13). Another version, used for error detection and correction in high-speed modems is known as LAP-M (Link Access Protocol for Modems).

The principle limitation of the go-back-N protocol is that whenever the destination DTE fails to receive a message, the transmitter must retransmit all messages sent after the transmission of the lost message. This requirement is due to buffer limitations at the destination DTE.

10.5 Exercises

1. When a go-back-N packet is received, what is the relationship between Ns, Nr, Vs, and Vr?
2. Explain why the stop-and-wait protocol is a special case of the go-back-N protocol.
3. Why can't the window size be equal-to or greater-than the number of sequence numbers?
4. Explain how the go-back-N protocol can be used to support flow control.
5. Give an example of piggybacking.
6. How does SDLC distinguish between its three different frame formats?

7. What is the purpose of the poll/final bit?

8. What is transmitted by the hardware if the transmitting DTE transmits a message with Ns equal to 7, the poll/final bit set, and the value of Nr is 7?

Chapter 11

Selective Retransmission

11.1 Introduction

The Go-Back-N protocol offers considerably more flexibility and throughput than the stop-and-wait protocol. However, Go-Back-N has a problem in that when a frame is received out of sequence or in error, all frames subsequently received are discarded until the frame with the correct sequence number is retransmitted. Consider the situation shown in Figure 11.1 in which frames Ns=2 and Ns=3 are retransmitted, even though only frame Ns=1 was in error. Ideally, only those frames received in error should be retransmitted, thereby avoiding unnecessary retransmissions.

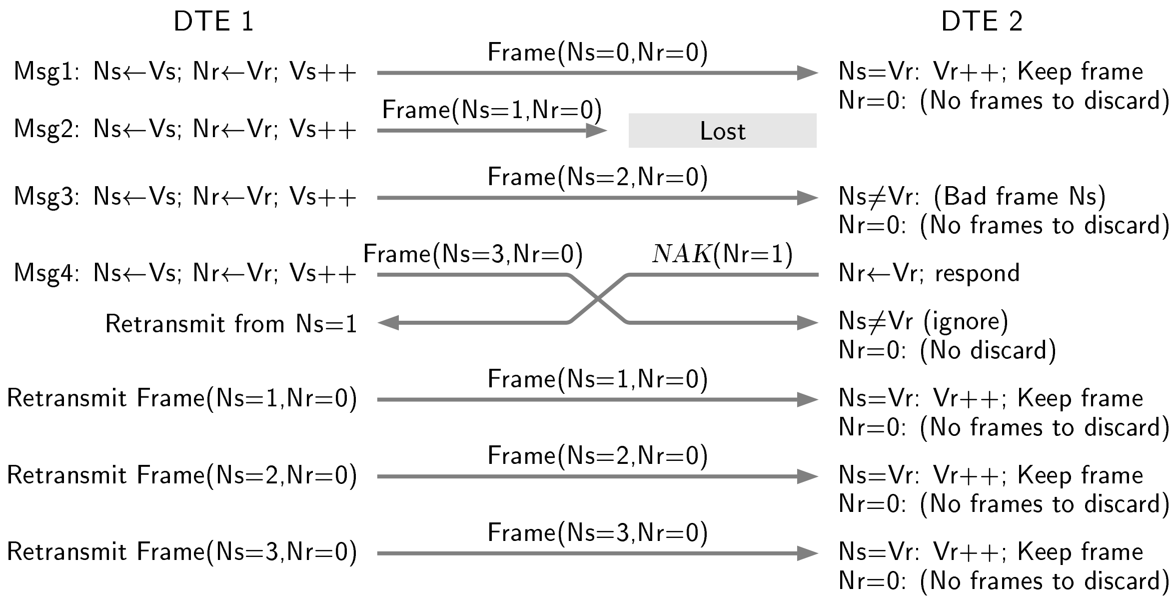


Figure 11.1: A Go-Back-N limitation: the retransmission of valid frames

The solution to this problem requires the receiving DTE to hold those frames that it has received correctly, and request the transmitting DTE to retransmit only those frames that are missing or received in error. This type of protocol is known as **selective retransmission**.

11.2 Selective Retransmission

Selective retransmission is a technique whereby only those frames that have not been received correctly (or have apparently not been received) are retransmitted. Selective retransmission is achieved through the receiving DTE explicitly acknowledging each frame as it is received. A missing frame will not be acknowledged, causing the transmitting DTE to eventually retransmit the unacknowledged frame (either by detecting the missing acknowledgement or by the time-limit associated with the frame expiring); for example, see Figure 11.2. Note that in this example, Nr refers to the frame number being acknowledged as opposed to the next expected frame.

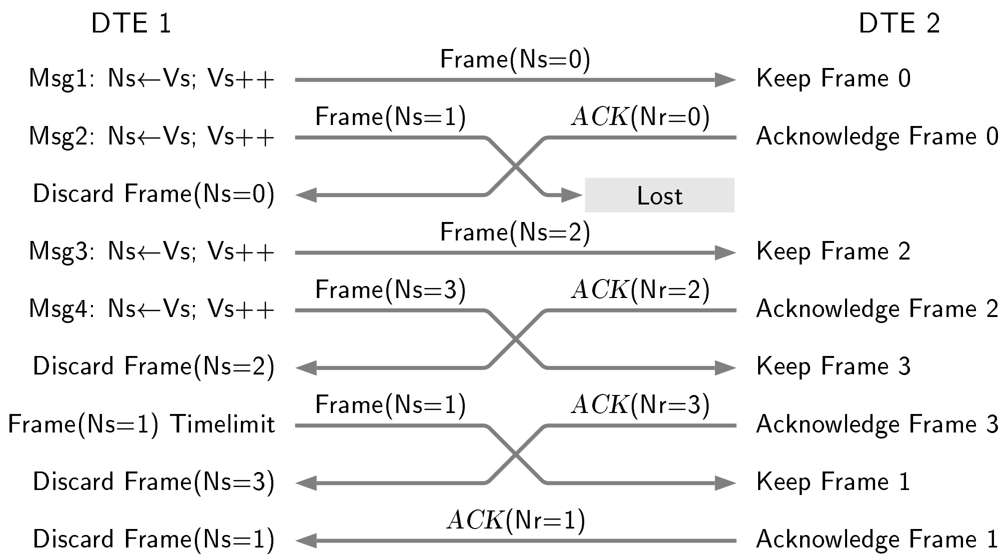


Figure 11.2: Selective retransmission: only one frame is retransmitted

The time it takes to recover from the loss of a frame depends upon the time limit associated with each frame. This delay can be reduced by having the receiving DTE transmit a *NAK* that explicitly identifies the missing frame. In Figure 11.2, DTE 2 can transmit a *NAK* 1 as soon as frame Ns=1 is determined to be missing (i.e., after frame Ns=2 is received). Upon receipt of the *NAK*, DTE 1 can retransmit frame Ns=1.

The loss of an acknowledgement can lead to some interesting problems if there are a limited number of sequence numbers. For example, consider the situation shown in Figure 11.3, in which two sequence numbers are used to represent the frames that are sent.

When DTE 2 receives the retransmitted frames Ns=0 and Ns=1, there is no indication that these are retransmissions, with the result that DTE 2 will keep *two* copies of frames Ns=0 and Ns=1. Using a transmission window which slides as each acknowledgement is received will not work for selective retransmission. For example, by using three sequence numbers (0, 1, and 2), and requiring

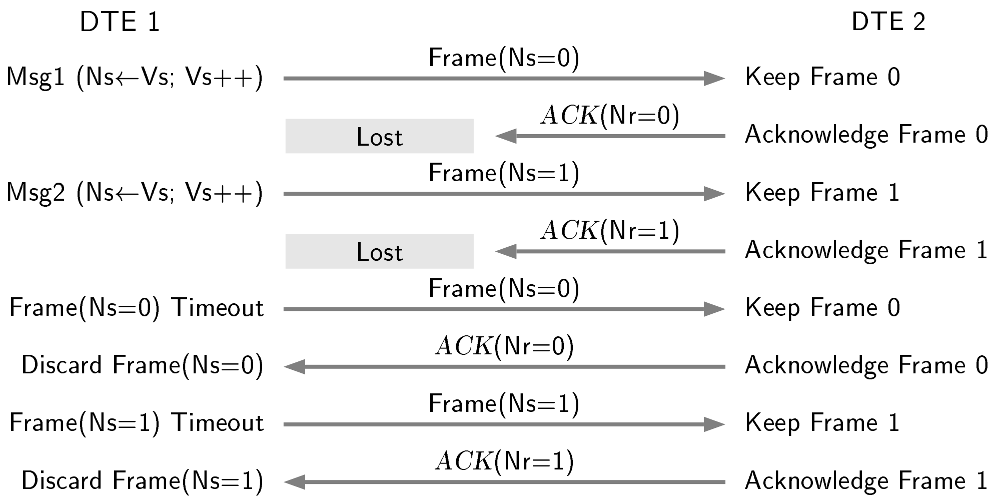


Figure 11.3: Sequence numbering problem with selective retransmission

the transmitting DTE to have at most two frames waiting for acknowledgement and sliding the window forward as acknowledgements are received, fails if frames Ns=0 and Ns=1 are transmitted and an acknowledgement is received for frame Ns=1 only. If the transmitting DTE sends frame Ns=2 and retransmits frame Ns=0, the receiving DTE once again has no indication that frame Ns=0 is a retransmission.

There are at least two solutions to this problem. One approach is to use a form of sliding window that moves the window forward only when the 'oldest' frame has been acknowledged.

Another, more common approach, is to require the transmitting DTE to inform the receiving DTE of the range of sequence numbers to be used. This allows the receiver to reserve space for the frames and 'know' which sequence numbers are expected. As an example, a transmitting DTE could inform the receiver that sequence numbers 0, 1, and 2 are to be used for the first set of frames being sent. When the transmitter has successfully received acknowledgements for these sequence numbers, it informs the receiver of the next range of sequence numbers to be used.

Although selective retransmission requires the destination DTE to maintain all correctly received frames (a potential storage problem), the fact that it reduces unnecessary retransmissions makes it particularly attractive in networks where bandwidth is an issue.

11.3 Summary

The major limitation of the go-back-N protocol is the requirement that the transmitting DTE retransmit not only the missing frame, but all those that were transmitted after it. In a low error environment, this may be acceptable, but in other cases, the retransmission of valid frames is a waste of bandwidth. This limitation can be overcome using selective retransmission: a protocol that requires only missing frames to be retransmitted.

Selective retransmission redefines acknowledgements; the acknowledgement refers to the frame number accepted as opposed to the next expected frame number. A missing frame can be detected by the transmitter with a timeout or by the receiver failing to receive a given frame number. Unlike go-back-N, selective retransmission requires the receiver to have sufficient buffer space to hold all frames; by making the receiver hold out-of-sequence frames, the bandwidth required for the transmission can be reduced.

IBM's SDLC, designed to support a go-back-N protocol, has an additional supervisory code, SREJ, or **selective reject**, which can be used for selective retransmission. The most widespread use of selective retransmission is the Internet protocol, TCP, discussed in Chapters 14 and 15.

11.4 Exercises

1. Show, by means of illustrations or examples, how *NAK*s can improve throughput for selective retransmission.
2. Show, by means of illustrations or examples, how a sliding window protocol can be used to support selective retransmission.
3. Under what conditions does selective retransmission degenerate into the go-back-N protocol?
4. Implement the selective retransmission protocol on the Ethernet.

Chapter 12

Forward Error Recovery

12.1 Introduction

In certain situations, it may not be possible or even desirable to retransmit a frame that has been received in error. For example, space probes with limited storage capacity that are hundreds of millions of miles from Earth cannot be expected to hold onto frames while waiting for acknowledgements. Solutions such as transmitting the same message twice not only take twice the bandwidth, but, as shown in section 3.5, do not offer the receiving DTE any indication as to which frame (if any) is correct.

12.2 Forward Error Recovery

In applications such as the one just described, it is necessary to supply the receiving DTE with the capability to both detect and, if possible, *correct* any errors found in the frame. This is known as **forward error recovery**, and requires the transmitting DTE to send sufficient additional information within the frame so as to allow the receiving DTE to correct the detected errors. The events that occur in forward error recovery are illustrated in Figure 12.1.

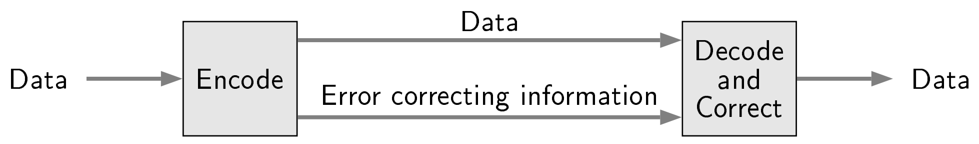


Figure 12.1: Forward error recovery

The **Hamming code** is one of several different **error-correcting codes** that use parity bits both to detect and correct errors. In the Hamming code, a frame consists of two types of bit: data bits (obtained from the data to be transmitted), and parity bits (generated from the data bits stored in the frame). The parity bits are stored in those frame-bit positions that are powers of 2; for example, 1, 2, 4, 8, and so on (the rightmost bit of a frame is considered to be bit 1); the data bits are stored in the remaining frame-bits.

The value of a parity bit is determined by its position in the frame; the parity bit stored in bit 2^i includes all frame-bit positions that have bit 2^i set. For example, the parity bit stored in frame-bit position 2^0 (i.e., frame-bit 1), is generated from frame-bits 3, 5, 7, 9, and so on (since these frame-bit positions have bit 1 set). Table 12.1 shows the association of frame-bits and parity bits in a 7-bit frame.

Table 12.1: The association of frame-bits and parity bits

Frame	Binary	Parity bits		
bit	value	4	2	1
1	001			X
2	010		X	
3	011		X	X
4	100	X		
5	101	X		X
6	110	X	X	
7	111	X	X	X

The 7-bit frame shown in Table 12.1 has three parity bits (stored in frame-bit positions 1, 2, and 4, since these are powers of 2), and allows four data bits (stored in the remaining frame-bit positions, i.e., 3, 5, 6, and 7). The Hamming code uses even-parity, meaning that the values of the parity bits can be obtained using exclusive-OR; Table 12.2 shows how the three parity bit values in Table 12.1 are generated (Px denotes the parity bit in frame-bit position x, while Dx denotes the value of the data bit in frame-bit position x).

Table 12.2: Parity bit values for Table 12.1

$$\begin{aligned} \mathrm{P1} &= \mathrm{D3} \oplus \mathrm{D5} \oplus \mathrm{D7} \\ \mathrm{P2} &= \mathrm{D3} \oplus \mathrm{D6} \oplus \mathrm{D7} \\ \mathrm{P4} &= \mathrm{D5} \oplus \mathrm{D6} \oplus \mathrm{D7} \end{aligned}$$

For example, to transmit the nibble `1101` using the above algorithm first requires the assignment of the data bits; in this case D3, D6, and D7 are set, while D5 is clear. The assignment of the bits in a 7-bit frame is shown in Figure 12.2.

D7	D6	D5	P4	D3	P2	P1
1	1	0	?	1	?	?

Figure 12.2: Frame including data and parity bits

To obtain the parity bits, it is necessary to generate each parity from the data bits in each frame-bit.

The parity bits are therefore assigned as follows:

$$\begin{aligned} \mathsf{P1} &= 1 \oplus 0 \oplus 1 \\ \mathsf{P2} &= 1 \oplus 1 \oplus 1 \\ \mathsf{P4} &= 0 \oplus 1 \oplus 1 \end{aligned}$$

(i.e., P1 is clear, P2 is set, and P4 is clear). The frame to be transmitted is assembled from the four data bits and the three parity bits, and is shown in Figure 12.3.

D7	D6	D5	P4	D3	P2	P1
1	1	0	0	1	1	0

Figure 12.3: Frame including data and parity bits

Once the data and parity bits have been assigned to the frame, it can be transmitted. Upon receipt of the frame, the parity and data bits can be extracted; however, the question remains: is the frame in error and, if so, how can the error be corrected?

In Chapter 2, error detection with parity bits was achieved by calculating the parity on the entire frame (including both the data bits and the parity bit). The same approach is taken with the Hamming code; however, in the case of the Hamming code, since there are several parity bits, each parity must be calculated. In the 7-bit frame example, there are three parity bits, which means that three parity checks are required, as shown in Table 12.3 (note, Ex denotes the status of the parity associated with frame-bit x; 0 means that no error was detected, while 1 signals an error).

Table 12.3: Hamming code parity checking

$$\begin{aligned} \mathsf{E1} &= \mathsf{P1} \oplus \mathsf{D3} \oplus \mathsf{D5} \oplus \mathsf{D7} \\ \mathsf{E2} &= \mathsf{P2} \oplus \mathsf{D3} \oplus \mathsf{D6} \oplus \mathsf{D7} \\ \mathsf{E4} &= \mathsf{P4} \oplus \mathsf{D5} \oplus \mathsf{D6} \oplus \mathsf{D7} \end{aligned}$$

In this example, the receipt of the frame `1100110` would result in:

$$\begin{aligned} \mathsf{E1} &= 0 \oplus 1 \oplus 0 \oplus 1 \\ \mathsf{E2} &= 1 \oplus 1 \oplus 1 \oplus 1 \\ \mathsf{E4} &= 0 \oplus 0 \oplus 1 \oplus 1 \end{aligned}$$

where E1, E2, and E4 are all clear, indicating that no error was detected, meaning that the nibble value is `1101`.

Should an error occur (remember, this code only detects a single-bit error), what happens? Assuming that the same frame was transmitted, but instead of receiving `1100110`, the frame `1110110` was received (i.e., bit D5 was inverted), the result of the error check will be:

$$\begin{aligned}\mathsf{E1} &= 0 \oplus 1 \oplus 1 \oplus 1 \\ \mathsf{E2} &= 1 \oplus 1 \oplus 1 \oplus 1 \\ \mathsf{E4} &= 0 \oplus 1 \oplus 1 \oplus 1\end{aligned}$$

In this situation, `E1` and `E4` are set, while `E2` is clear. Clearly, the error has been detected, but how can it be corrected?

To understand how error correction is achieved it is necessary to re-examine Figure 12.1. Each frame-bit in this Figure is associated with a unique encoding of parity bits; for example, frame-bit 6 is represented by parity bits P4 and P2 being set, while P1 is clear. Should any frame-bit be inverted in the transmission, the parity check at the receiver will be incorrect; furthermore, only those parity values associated with the inverted frame-bit will signal an error. For example, if frame-bit 6 was inverted, parity bits P4 and P2 would be set, while P1 would be clear; indicating that frame-bit 6 (from $4 \times 1 + 2 \times 1 + 0$) was in error and should be inverted.

Therefore, when the frame `1110110` was received, the result of the error check indicated E1 and E4 set and E2 clear. This means that $4 \times 1 + 1 \times 1$ or frame-bit 5 was inverted (i.e., D5 should have a value of 0 rather than 1). By inverting bit 5 in the frame (i.e., `1100110`), the corrected nibble can be extracted as `1101`.

The Hamming code examined in this section will detect any single-bit errors; including errors in the parity bits. Single and multiple-error correcting Hamming codes are often used when storing data in both internal and external storage. Other examples of error-correcting with the Hamming code are given as Exercises.

12.3 Summary

The ARQ protocols examined in the previous chapters require the receiver to indicate to the transmitter the state of the communication and whether a given frame should be retransmitted. On most terrestrial applications, this is acceptable, since the distances are limited and delays are on the order of seconds. Protocols which require the transmitter to recover from the frame error are support **backward error recovery**.

However, in some applications, especially those involving applications with limited storage capacities or long transmission delays, it may make more sense to have the receiver recover from the error, rather than waiting for a retransmission. Protocols that use the receiver to correct transmission errors support what is known as **forward error recovery**.

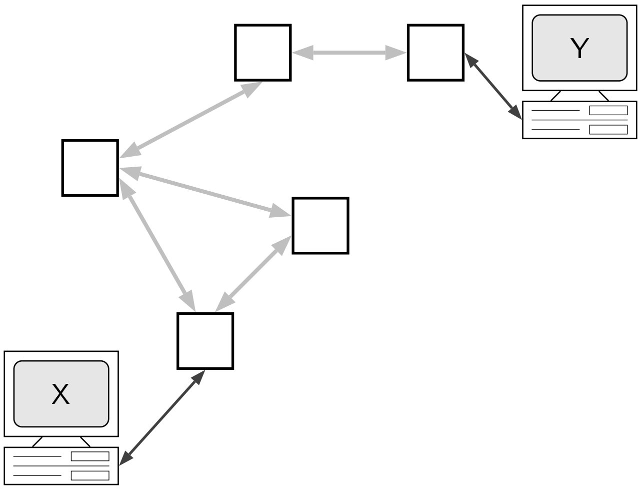
12.4 Exercises

1. Show the encoding of the following nibbles using the Hamming code: `0000`, `1001`, `0110`, and `1010`. Verify the results by decoding the encoded data.

2. Which of the following 7-bit Hamming frames are correct; correct those that are incorrect: `0000010`, `1111011`, `1111111`, and `1000111`.

Part 6

Computer communications across different networks



Until now, all network communications have been **intranetwork** communications; that is, the communications take place within a single organization, across one or more networks. However, there are times when it is necessary for an application on a computer on one network to communicate with a computer on a separate network; consider the following examples:

- a company may have offices scattered across the world, all well outside the physical limitations of a local area network. Options open to the company include using the telephone system to interconnect the offices or accessing a commercial network.
- a group of universities and colleges may all contribute to purchase a 'super-computer' which is sited on a local area network at one of the schools. Access to the 'super-computer' from research machines on local area networks at other institutions may require the interconnection of two or more of the various local area networks.

Until recently, the first of the above examples (an organization accessing a commercial network), would have been handled by a single **wide area network**. In a wide area network, users attach to a common, shared network, which is operated by a third party, typically a national telecommunications organization.

The second example (universities and colleges sharing network facilities) requires what is known as an **internetwork communication** or communications *between* distinct networks. Internetwork communications, unlike those using a wide area network, take place across a number of interconnected networks. The various networks that make up the internetwork are referred to as an **internet** or **catenet** (short for 'concatenated network').

Not surprisingly, an internet spanning a region or even a country exhibits many similarities to a wide area network. However, there are a number of interesting differences:

- First, a wide area network is physically a single network, whereas an internet is made up of many distinct networks (that may include wide area networks).
- Second, the wide area network supports a common set of data link protocols, shared by the nodes that make up the network; while in an internet, different networks will probably support different data link protocols.
- Third, a single addressing structure is used within a wide area network, ensuring that each node in the network is uniquely identified; while in an internet, each network supports its own addressing scheme, with no guarantee that the addresses are not duplicated on other networks.

Part 6 examines both wide area networks and internetworks, highlighting their similarities and differences. In Chapter 13, the internal structure of the most widely used wide area network protocol, X.25, is discussed. Chapter 14 looks at the problems associated with designing an internet and the solutions adopted for today's "Internet."[1]

[1]An *i*nternet is a collection of distinct networks that support shared communications; the *I*nternet is the name given to the world-wide network that is constantly in the news.

Chapter 13

Wide Area Networks

13.1 Introduction

The material covered thus far has illustrated techniques for minimizing transmission overheads, overcoming distance limitations, and maximizing channel utilization. However, even these techniques cannot resolve problems such as the following:

- A branch office uses a multiplexer and modem to connect to their head office located several hundred kilometres away. In order to allow 'instant access' to the head office, a telephone connection is maintained throughout the working day (including coffee breaks, lunch time, and meetings).

  Although the occasional file transfer and some busy periods during the day may, from time-to-time, keep the channel at near capacity, there is insufficient network demand to warrant the telephone charges paid by the company.

- A telephone company analyzes its network traffic and discovers that the average duration of a voice call is about 5 minutes, while data calls can last for hours. Meanwhile, the annual growth in data traffic exceeds that of voice by roughly a 10 to 1 margin.

These two examples highlight an interesting contradiction: although more data traffic may suggest additional revenue to the telephone company, it can also result in more customer dissatisfaction because of failed call attempts (by both data and voice customers) due to congestion in the telephone network caused by the data traffic.

There are a number of possible solutions to this problem, including:

- install more telephone circuits, thereby reducing the probability of failed calls, and
- offer inducements to existing and potential data customers to get them off the voice network.

The first solution, increasing the number of circuits, will alleviate the situation only for the short term, since data traffic continues to grow. This leaves the second solution, getting data customers off the voice network, as the only real alternative. The problem is: where do they go?

13.2 Alternatives to the Public Telephone Network

13.2.1 Private Lines

A common approach to getting data customers off the voice network is for the telephone company to lease them a **private line**, also referred to as a **leased** or **dedicated** line. The private line connects the data customer's DTE directly (via the telephone company's circuits) to the remote DTE. There are a number of advantages in having a private line, including minimal line noise, guaranteed access to the telephone network, and the 'freeing-up' of a voice circuit.

This solution has been taken even further: some larger companies rent entire private telephone networks from telephone companies or other organizations, thereby ensuring that all their calls (both data and voice) can be placed. It is becoming increasingly commonplace to find private companies (other than the telephone companies) with their own private telephone networks.

13.2.2 Public Data Networks

In some countries, **public data networks** (or **PDN**s) have been installed, and are designed specifically for the transmission of data rather than voice. Circuit switched data networks provide digital transmission facilities (i.e., the modulation and demodulation of signals is not required) for high-speed, high-volume communications. A typical digital data network may allow full-duplex communications for both asynchronous and synchronous DTEs at speeds of up to 56 kilobits-per-second. Many telephone companies support data networks designed specifically for digital (as opposed to voice) communications. For example, in the United States, AT&T offers **Data-Phone Digital Services** (DDS), while in Canada, Telecom Canada offers a service known as **Dataroute**.

Access to the voice network can be avoided by attaching DTEs directly to the digital network, using a DCE (or **data circuit-terminating equipment**) designed specifically for digital technology.

Accessing Public Data Networks

In Chapter 2, RS-232-C (V.24) and its proposed successors: RS-422-A, RS-423-A, and RS-449 were discussed. To overcome the well-known limitations of RS-232-C, the new standards increase the number of connectors from a single 25-pin 'D' connector to a pair of 'D' connectors for RS-449: a 37-pin primary channel connector and an optional 9-pin secondary channel connector. For all of the benefits associated with these new standards, the number of connector pins increases the cost. Ideally, what is needed is a mechanism that supports *fewer* connections but offers at least the same number of functions.

ITU-T Recommendations **X.20** (asynchronous) and **X.21** (synchronous) are examples of standards defining digital access to public data networks. In these standards, functions are coded as digital messages and passed between the DTE and DCE as streams of character strings using 15-pin connectors rather than devoting a single pin to a specific function. X.20 and X.21 provide services other than data transfer; for example, automatic calling features are also supported, as well as signals that allow the DTE to determine the state of a call. The X.21 circuits are shown in Figure 13.1.

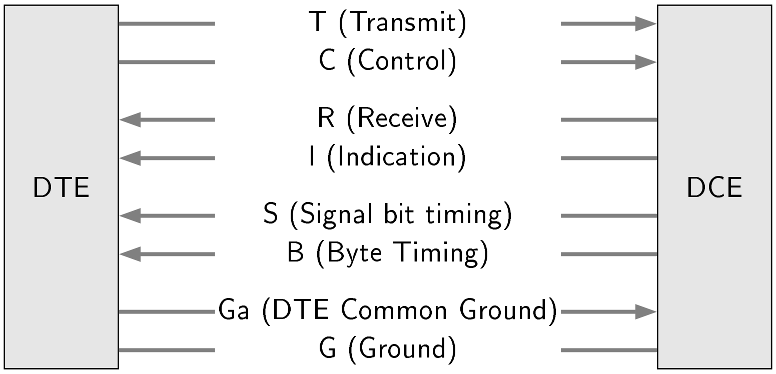


Figure 13.1: X.21 Circuit definitions

In X.21, both the DTE and DCE are defined as state machines, with the various states assigned to one of four phases: quiescent, call control, data transfer, and clearing (call) circuits. State transition is caused by changing the value of the signals on the T (Transmit), C (Control), R (Receive), and I (Indicate) circuits (T and C are associated with the DTE, while R and I are associated with the DCE).

For example, if both the DTE and DCE are in the quiescent phase (i.e., a call can be signalled by either the DTE, as an outgoing call, or the DCE, as an incoming call), the T and R circuits idle with a continuous stream of 1s while the C and I circuits are in an off state (signalled by a continuous stream of 1s). The DTE can signal a call request by changing the value of the T and C circuits to a stream of 0s and entering the on state (a stream of 0s), respectively. The DCE responds with either a stream of '+'s indicating that the DTE can proceed to place the call or a stream of *BEL* (ASCII 0x07) characters, signalling that the outgoing call has collided with an incoming call. Either response (i.e., the '+' or the *BEL*) is prefixed by a pair of *SYN* bytes and is sent on the R circuit (the I circuit remains in the off state).

In the case where the call can be placed, the DTE supplies the DCE with the number of the party called as a stream of ASCII (IA5) digits on the T circuit. The DTE can request specific facilities (such as making the call a 'collect call') as part of the data stream; these facilities are defined in ITU-T X.2. The final digit is followed by a continuous stream of 1s on the T circuit, after which the DTE waits for the DCE to signal the progress of the call. While the DCE is attempting to place the call, the R circuit continues to idle with a stream of '+' characters. The DCE can inform the DTE of the progress of the call (for example, Number Busy or Invalid Facility Request) by sending Call Progress Signals on the R circuit in place of the '+' characters. Call Progress Signals are defined in ITU-T X.96.

A DTE is informed of an incoming call by receiving a *BEL* on the R circuit (prefixed by a pair of *SYN* characters) while in the quiescent phase. The DTE accepts the call by changing the C circuit from a stream of 0s to a stream of 1s.

Data transfer can commence when the T and R circuits indicate a stream of 1s and the C and I circuits are in the on state (a continuous stream of 0s). X.21 supports full duplex communications; data is sent on the T circuit and received on the R circuit. If data is not available, the circuit (T or R) idles with 1s, while the C (or I) circuit remains in the on state.

Either DTE can clear the call by setting the T circuit to transmit a stream of 0s and returning the C circuit to the off state. The DCE connected to the DTE clearing the call responds by setting the I circuit to off. The remote DCE signals the call clearing request to the remote DTE by setting the R and I circuits to a stream of 0s and the off state, respectively.

Both balanced (X.27) and unbalanced (X.26) circuits are defined for both X.20 and X.21. X.27 is similar to the RS-422-A standard and is recommended for DTE-DCE line speeds greater than 9600 bps. For lower speeds, unbalanced circuits may be specified using X.26, which is similar to RS-423-A.

Two ITU-T standards, X.20*bis* and X.21*bis*, are available for DTEs that are configured to support either RS-232-C or V.24. X.21*bis* (and X.20*bis*) are designed to convert X.21 (and X.20) signals into an equivalent RS-232-C or V.24 value; thereby allowing a DTE with analog equipment access to a digital public data network.

13.2.3 Sharing Channels

The impact of data traffic on the voice network is clearly minimized through the use of private lines or public data networks. However, the problem of channel utilization still exists: there will be periods throughout the day for which the channel is being paid but when it is not in use.

Ideally, what is needed is a communication channel that is always available to the customer, but for which the customer is charged only when transmitting or receiving information. If this channel could be shared amongst a number of different customers, costs could be shared as well (based, perhaps, on the volume of data transmitted each month). For example, a number of different customers in city A could share a channel to city B. Multiplexing could be used to ensure that the information sent by a customer in city B arrives at the correct DTE in city A and vice versa. By using statistical multiplexing, the channel would be used only by those customers who actually required the channel at any particular moment, as depicted in Figure 13.2.

Figure 13.2: A shared channel between two cities, A and B

In Chapter 4, all communications were assumed to take place between one or more central hosts and a group of remote DTEs multiplexing the same channel owned by a single organization. However, in this situation, there can be a number of DTEs at either end of the multiplexed channel belonging to different customers, as depicted in Figure 13.3.

There are essentially two approaches that can be used to support the transfer of information between the DTEs:

- First, all DTEs can be statically paired using a predefined connection (for example, DTE L in city A is always connected to DTE S in city B). The static connection is simple to implement and requires very little processing on the part of the multiplexers; however, users are given little flexibility.

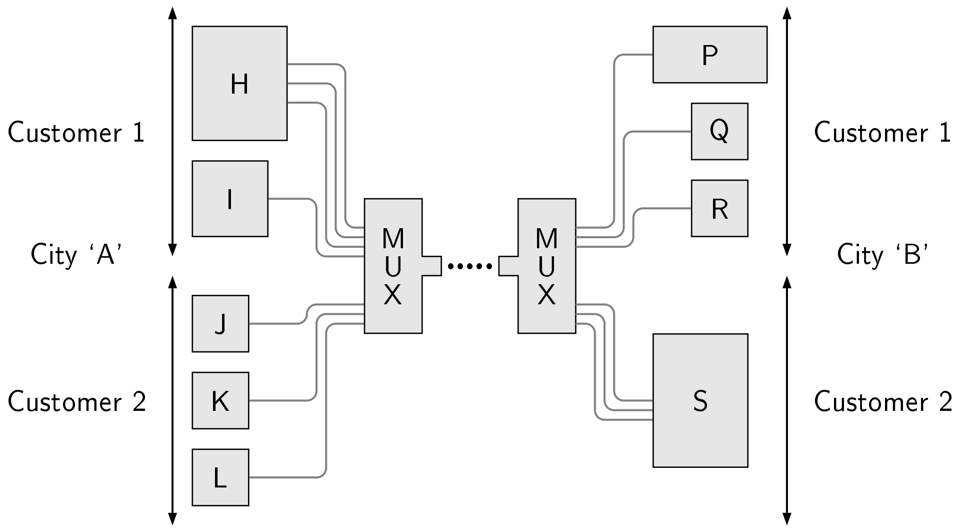


Figure 13.3: Sharing a MUX between different customers in different cities

- The second approach is to allow the various DTEs to dynamically select their intended destination (for example, the user of DTE I in city A may want to connect to any of DTEs P, Q, or R). The dynamic connection is more difficult to implement for a variety of reasons: mechanisms are needed to detect whether the remote DTE is available, as well as to determine if both DTEs are owned by the same customer. However, the advantage of this approach is the flexibility it offers the user.

It may turn out that organizations in a third city (city C) also need to communicate with facilities in city B: resulting in another multiplexed channel, this time between cities B and C; as shown in Figure 13.4.

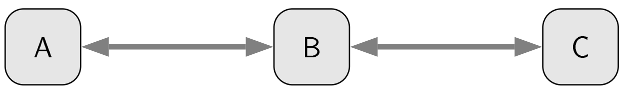


Figure 13.4: Two multiplexed channels connecting three cities

Now, what happens if a customer in city A wants to communicate with the group's office in city C? There may not be sufficient traffic to warrant putting another pair of multiplexers in cities A and C thereby connecting the two cities. On the other hand, the costs of communicating between cities A and C directly may be so prohibitive as to force the abandonment of the planned communication.

Of course, there *is* a link between cities A and C (albeit an indirect one) via the multiplexers in city B. To permit communication between DTEs in cities A and C requires the interconnection of the two multiplexers in city B. Once that connection is made, communications between city A and city C can pass through city B, giving the impression that cities A and C are directly connected, as shown in Figure 13.5.

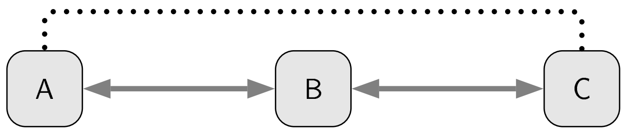


Figure 13.5: A logical connection between cities A and C

The interconnection between the two multiplexers in city B can be either:

Direct. The two multiplexers can be directly connected by having each multiplexer treat the other multiplexer as one of its DTEs, as depicted in Figure 13.6.

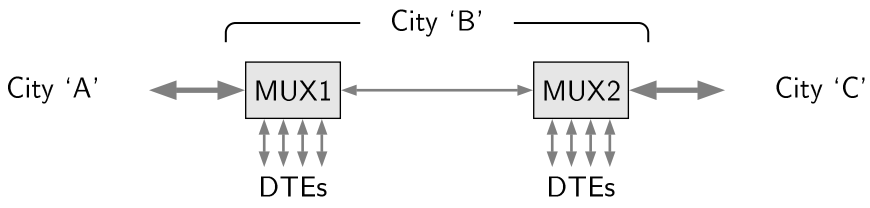


Figure 13.6: Connecting A and C by direct MUX interconnection

The functions performed by the multiplexers depend upon a number of issues. For example, if the connection(s) are dedicated to specific DTEs in cities A and C, the multiplexers simply pass the data through the related connection. The connection between the two multiplexers is configured to give the illusion to MUX1 that MUX2 is the DTE in city C; while to MUX2, MUX1 is the DTE in city A.

However, if the connections between the two multiplexers are accessible by any DTE, then additional information must be passed between the DTEs prior to beginning any information transfer. For example, if a DTE in city A is to send information to a DTE in city C, then the following steps will probably be required:

1. obtain a free channel connecting MUX1 to MUX2.
2. inform MUX2 of the intended destination DTE in city C.
3. allow the communication to proceed.

Indirect. The two MUXes can be indirectly connected by passing information between a DTE that is connected to both MUXes, as shown in Figure 13.7.

When data is sent to a DTE in a remote city (say, from city A to city C), it is first passed to a port on the DTE connected to both multiplexers. The DTE can then forward the data out the port associated with the destination DTE in city C. The shared DTE can be made to support both communication methods described for directly connected multiplexers.

This approach offers considerably more flexibility than directly connecting multiplexers since the tasks are *well defined*. That is, the multiplexers are responsible for handling multiplexed

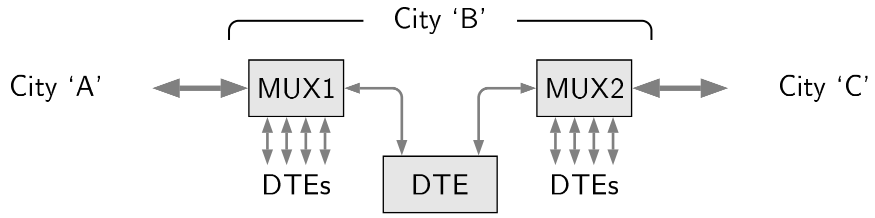


Figure 13.7: Indirect connection of MUXes by an intermediate DTE

data and forwarding the data to a specific port; while the shared DTE is responsible for establishing the connections and taking the data from an incoming port and supplying it to another, outgoing, port.

In addition to the above, a subtle change has occurred in the way the communication takes place. Until now, all communications have been point-to-point with no intermediate devices other than those directly involved in the communication. Now however, communications can take place between a *number* of intermediaries.

13.3 Wide Area Networks

The previous section proposed the designing of a system of interconnected multiplexers to support data traffic, rather than a combination of voice and data as in the existing telephone network. In addition, if statistical multiplexing is utilized, the bandwidth is employed only when necessary, potentially reducing the costs associated with using the telephone system.

In the mid-1960s, people working in the field of data communications recognized the potential of developing computer **networks** that would allow the transmission of data between geographically dispersed DTEs. Since these networks were envisaged to span whole continents (or even the entire world), they were called **wide area networks** or **WAN**s.

A wide area network is an interconnection of a number of multiplexing devices known as **nodes**. Each node in the network is directly connected to at least one other node through a communication channel known as a **link**. If a node is directly connected to N other nodes, it is associated with N links. Links are typically high speed communication channels such as optical fibre, coaxial cable, microwave, or even satellite.

Additionally, each node supports a number of DTEs. A DTE can be directly connected to a node, or the node may support 'dial-in' facilities whereby remote users can access the wide area network. Once connected to the network, a DTE potentially has access to any other DTE on the network. A hypothetical wide area network is shown in Figure 13.8 (each square is a node, while the lines are links).

The internal structure of a node closely resembles the multiplexer-DTE-multiplexer combination discussed in the previous section. A node performs a variety of functions, including:

- allowing DTEs to send and receive information on the network.

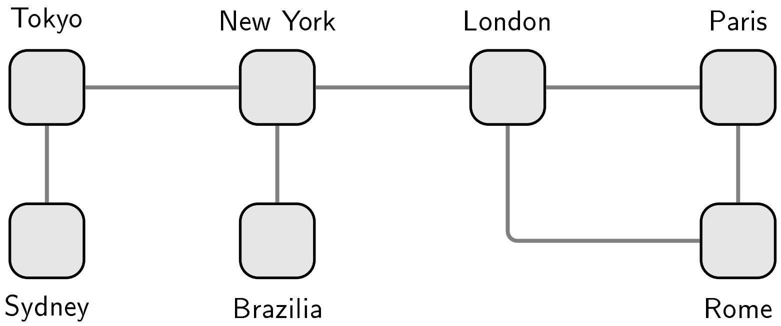


Figure 13.8: A hypothetical wide area network

- routing information to DTEs via other nodes. For example, in the above wide area network, a DTE in Sydney could send information to a DTE in Paris via the Tokyo, New York, London (and possibly Rome) nodes.

- multiplexing information on the links; thereby, sharing the links amongst a number of DTEs. For example, all the DTEs attached to the Sydney node would share the link to Tokyo in order to reach DTEs on nodes other than Sydney.

How the above functions are realized in a wide area network depends upon a number of issues, some of which are now considered.

13.3.1 Topologies

There is no prescribed 'shape' to a wide area network, since the geographic area covered may vary from country to country or continent to continent. However, if the geographic factors are ignored, there are a limited number of topologies:

Fully Connected. A fully connected network is one in which every node on the network connects to every other node. If the network consists of N nodes, each node will require $N - 1$ links. For example, the network in Figure 13.9 has six nodes; each node has five links.

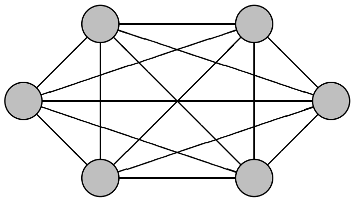


Figure 13.9: A fully connected network

Fully connected networks are usually very costly because of the number of links required. However, they are usually very fast (since there are no intermediate nodes to handle a message), and they offer a high degree of reliability (if a link fails, there should be an alternate path to the intended destination).

Star. A star topology is one in which all nodes connect to a single, central node (often referred to as the **hub**); all communications between nodes pass through the hub node. Figure 13.10 is an example of a start network.

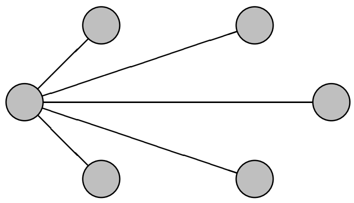


Figure 13.10: A star network

Although the star reduces the number of links to a minimum (i.e., N nodes require $N-1$ total links), the failure of the hub node will result in the complete cessation of all communications.

Mesh. Another topology, probably the most common of all wide area networks, is known as a mesh or partially connected network. In a mesh network, usually distance or known traffic volumes determine which of the nodes are connected; a possible mesh network is shown in Figure 13.11.

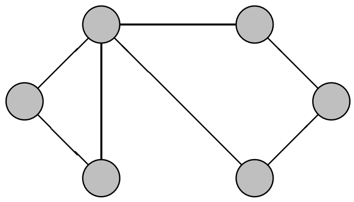


Figure 13.11: A mesh network

Mesh networks can also evolve as communication patterns change over time. For example, a city might not initially have a node, but if it subsequently gains political or economic stature, a node might be added. Similarly, the traffic volumes between two cities might be low, justifying the use of an intermediate node; however, over time the volumes might grow to the point where a separate link directly connecting the two cities may be needed.

Other topologies. There are a number of other network topologies possible. Two specialized topologies are the **ring** and **bus**; their topologies are shown in Figure 13.12.

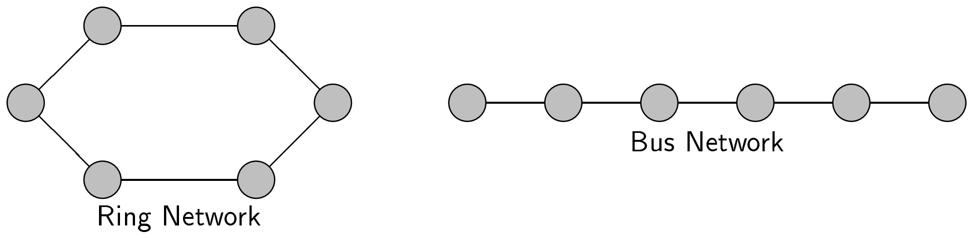


Figure 13.12: The ring and bus network topologies

13.3.2 Addressing DTEs

Sending information on a wide area network is not the same as sending information in a point-to-point communication. In a point-to-point communication there is only one possible destination (the DTE at the other end of the channel), whereas in a wide area network there can be any number of possible destinations. Therefore, at a minimum, whenever a DTE supplies information to a node for transmission, the node must know the intended destination.

The intended destination is simply another DTE attached to a node somewhere on the network. How much information does the node require in order to get the information to the destination? Consider how a letter addressed to someone in another country is processed by the post office. The one thing the post office doesn't have is a list of all the people in the world; however, it does have a list of countries and, possibly, regions within countries. The letter is first sent to the country in question; then to a region within the country; then to the city; and so on down to the person's house. In other words, there is a **hierarchy** used to route the letter.

The same approach can be applied to sending information in a wide area network: the information is not sent directly to a DTE or a port on a DTE. Instead, the information is sent to the node to which the destination DTE is attached. The node can then supply the information to the DTE or perhaps a port on the DTE (assuming the DTE supports several connections to the node).

All networks require some form of **address** to ensure that the information arrives at the intended destination. Broadly speaking, there are two types of addresses:

Hierarchial. A hierarchial address is a single address consisting of a series of fields, each of which is used to identify part of the destination. For example, a hierarchial address could consist of the node, the DTE, and the port on the DTE.

Global. A global address is a single address that uniquely identifies a single entity on the network (typically a DTE attached to a network node).

All DTEs attached to the network are associated with an address, potentially meaning that any pair of DTEs can communicate; assuming they agree to the same protocol.

In addition to the destination address, most, if not all, wide area networks require the address of the transmitting DTE (the **source address**), to also be sent. The source address can be used for a number of different purposes: the most obvious is that it allows the receiving entity a means whereby a reply or an acknowledgement can be returned.

13.3.3 Information Representation

To be effective and to attract as large a market as possible, wide area networks often support a variety of protocols, thereby allowing different DTEs access to the network. For example, the network should permit asynchronous DTEs to co-exist with synchronous DTEs. In addition, the wide area network should be transparent: DTEs accessing the network should not be required to change their protocol. For instance, if a telephone circuit between a pair of asynchronous DTEs is replaced by a wide area network, the asynchronous communications should continue as before.

These two seemingly contradictory objectives (co-existence of protocols and transparency) require the wide area network to support a universal data structure that is internal to the network and recognized by the nodes, regardless of the DTE information carried. The structure, commonly known as a **packet**, should be able to carry the information specific to the protocol as well as control information, specific to the wide area network. Typical control information can include the source and destination addresses as well as some form of error checking. A generalized packet structure is shown in Figure 13.13.

Packet Specific Information	DTE Specific Information	Packet Checksum

Figure 13.13: A generalized packet structure

13.3.4 Packet Routing

The various nodes that make up the network are responsible for the **routing** of packets through the network, from the source node to the destination node and eventually the destination DTE. There are a number of similarities between a circuit-switched telephone call and packet routing; accordingly, wide area networks are often labeled **packet switching networks** or **PSN**s. (The terms **Packet Switching Exchange**, or **PSE**, and **Packet Switch Node**, or **PSN**, are synonymous with node.)

The algorithms chosen to support packet routing in a network depend upon factors such as:

- the cost of sending a packet across a specific link. Some links may be more costly than others: terrestial links are typically cheaper than satellite links.
- the length of time it will take the packet to reach the intended destination. Time-critical information may be required to move through the network with a higher priority than other information.
- the volume of traffic on the network. As the number of packets on the network increases, **congestion** can occur, leading to delays and other problems (see below).
- the order the packets will arrive at the intended destination. If multiple paths exist in the network and packets are permitted to take any path; then packets taking a longer path may arrive at the intended destination later than other packets, even though the packet that arrived late was sent earlier.

- how lost or duplicate packets are handled. In any communication, information can be lost through some form of error. In a wide area network, mechanisms may be required to ensure that information always reaches the intended destination. Similarly, if multiple paths exist and a packet somehow is sent on two or more links, mechanisms may also be required to recognize the duplication.

When a packet is transmitted through a wide area network, the packet will be handled in three different ways:

1. the node attached to the source DTE will accept the DTE's information, store it in a packet, and transmit the packet onto the network.
2. the node attached to the destination DTE will remove the packet from the network. The information in the packet will be transmitted to the destination DTE using the DTE's protocol.
3. any intermediate nodes will route the packet to the final, destination node using some type of **routing strategy** or **routing algorithm**.

Routing Algorithms

There are many different algorithms that can be used to support packet routing through a network. At one extreme, the node can simply forward the packet out all of its links in the hope that eventually a copy of the packet will reach the intended destination node. This is known as **flooding** and although there is a very good chance that a copy of the packet will reach the destination, there is also a very good chance that many other copies of the packet will also reach the destination. In addition, copies of the packet could be travelling through the network for indefinite periods of time before reaching the destination (this is especially true if there are loops or cycles in the network). Two ways of improving this algorithm are:

- supply each packet with a **hop count**; that is, a counter which is given an initial value by the source node and decremented by each node that receives the packet. When the hop count reaches zero, the receiving node discards the packet. The hop count must be large enough to ensure that nodes at opposite ends of the network can communicate.
- transmitting the packet out all links *except* the one on which it was received. This helps to reduce the number of unnecessary transmissions between pairs of nodes. The assumption behind this algorithm is that if a node forwards a packet, it wasn't the intended destination, therefore there is no need to retransmit the packet back to the original node.

There are a number of limitations to the above approaches, the chief among them being the unnecessary routing of packets to nodes that aren't on the path to the intended destination.

An alternative to flooding and its variations is to supply each node with a data structure known as a **routing table**. In its simplest form, the routing table contains a list of all possible destination nodes and the link to the next node on the path to the specified destination. For example, using the network illustrated in Figure 13.14, station A's routing table as depicted in Table 13.1.

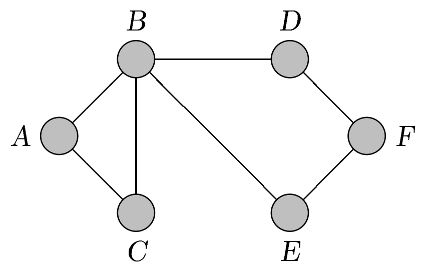


Figure 13.14: A wide area network of six stations

Table 13.1: Routing table for station A

Destination	A	B	C	D	E	F
Next node	-	B	C	B	B	B

In the routing table (Table 13.1), the Destination is the intended destination of the packet supplied to the node and Next node indicates the link to be taken to reach the destination (a '-' signals that there is no next node). For example, to reach nodes B, D, E, and F from node A, all packets would take the link to node B; however, to reach C, packets would be sent directly on the link to node C.

The choice of routes becomes somewhat more complex when considering node F. The routes from F to nodes D and E are readily apparent; however, there are two choices from F to B (either via node D or node E).

There are many ways in which the routing table can be constructed, organized, and used; some of which include:

- If the traffic patterns on the network are well known and the topology is stable (i.e., the addition or removal of a node or link is a rare event), it may be possible to decide upon the routes when the network is initially implemented. These routes can be placed in the routing tables of the different nodes and the network left to function. This is often referred to as **static allocation**.

  For example, it may be decided that the route from node F to node B is to go via node E, producing the routing table for node F shown in Table 13.2.

  Table 13.2: Routing table for node F

Destination	A	B	C	D	E	F
Next node	E	E	E	D	E	-

  The choice of routes can be based upon any number of factors. Typical reasons include the cost of transmitting packets across a link, traffic patterns (some nodes may be subject to delays), and distance.

If there exist multiple paths between nodes, the routing table could contain information about the different possible routes; this is known as **multipath routing**. For example, node F's routing table could be expanded to show that a route to node B also exists via node D (in addition to the route through node E). Multipath routing has a number of uses: traffic can be shared between the different routes; and should a route become inactive due to a link or node failure, an alternate route may be available.

- In many situations, the topology and traffic patterns of the network can change; for example, due to node or link failures and variable traffic patterns during certain times of day. For example, if node E fails, traffic from node F may have to be rerouted through D to reach B. Algorithms that allow the modification of routing tables are known as **dynamic** or **adaptive** routing strategies.

  A variation on the use of static allocation is to have a network control centre that periodically sends new routing tables to the nodes. These tables are based upon information that the network control centre receives from all of the nodes; each node sends packets containing network status information. For example, the network control center could be informed of node E's failure; it could then produce new routing tables that would have node F's traffic flowing via node D.

- The use of a centralized network control centre performing routing table updates can be costly to the network in that a percentage of the network bandwidth is lost to the exchange of status information. Ideally, the amount of bandwidth lost should be minimal. One such technique is known as **backward learning**.

  In backward learning, every packet contains a count that is increased as the packet is transmitted by each node (the further away a packet is from its source, the greater the value of the count). Each routing table entry has a distance count associated with every destination in the table (the count indicates the number of nodes the destination is away from this node). Initially, all routing table distance counts are set to infinity.

  The backward learning algorithm works as follows. Whenever a packet is received by a node, the destination entry in the routing table that corresponds with the *source* address in the packet is found. The routing table count is compared with the packet's count; if the packet count is less, the routing table entry is changed to the packet count, otherwise nothing is done.

  For example, if the link between B and D is not functioning, the routing table for node D might look like that shown in Table 13.3.

Table 13.3: Routing table for node D (Link B-D not functioning)

Destination	A	B	C	D	E	F
Next node	F	F	F	-	F	F
Count	4	3	4	-	2	1

Should the B – D link become active, a packet from B might arrive across the link at D with a count of 1. Since this count is lower than the existing routing table count of 3 for destination B, the Next node for Destination B changes to B and the Count changes to 1. This is shown in Table 13.4.

Table 13.4: Routing table for node D (Link B-D now functioning)

Destination	A	B	C	D	E	F
Next node	F	B	F	-	F	F
Count	4	1	4	-	2	1

As soon as a packet from D arrives at B, B's routing table count for D will change, meaning that messages from A and C can take the shorter route.

13.3.5 Methods of Packet Transmission

Packets are transmitted across a network one node at-a-time. Ideally, the packet will reach the intended destination with the minimum of difficulty. However, over a period of time, nodes may fail (i.e., crash due to a software or hardware fault) and links may fail (e.g., being cut by someone digging a trench). Should either of these types of fault occur, the packet transmitted might not reach its destination. The remedial action depends upon the type of service being offered by the network. Three common methods of service are now considered.

Datagram

A **datagram service** is a *best-effort* communication service that does not guarantee anything about the transmission of the packet. For example, a packet may be sent and then lost because of a network fault or the packet might simply be discarded due to heavy network traffic flow (i.e., network congestion). On the other hand, duplicate copies of the packet might arrive at the destination because of routing problems in the network or simply the routing algorithm itself (consider the possible effects of flooding).

In a datagram service, it is assumed that the DTEs are responsible for handling any errors associated with datagram transmission. For example, the DTEs could construct their own Go-Back-N protocol atop the datagram service.

Messaging

A **messaging system** is one in which a message is sent as a series of packets that are re-assembled into the complete message at each node before being forwarded onto the next link.

In a typical messaging application, a DTE divides a large set of data, the **message** (such as a file), into a series of packets. Packets are transmitted between nodes; the receiving node reassembles the message from the incoming packets. Once the message is assembled, the node performs its routing algorithm and determines the next link. The message might not be transmitted immediately, the node may perform some type of scheduling: typical scheduling criteria include the amount of buffer space available on the next node and the amount of traffic on a particular link. Networks that support messaging are also referred to as **store-and-forward** networks.

Messaging systems normally offer 'guaranteed' service, in which a message sent from one DTE will eventually arrive at the intended destination DTE. To achieve this level of service, the communications *between* each node must ensure that each packet has been successfully transmitted. Protocols similar to Bisync or SDLC are often used by the individual nodes when communicating across the link.

Messaging is often used in situations where interactive computing is not required and the arrival of the data at the destination is not time critical.

Virtual Circuits

There are applications that cannot function using datagrams or messaging. Consider, for example, an interactive terminal connected via a wide area network to a remote host. The user of the terminal wants to make sure that the data entered arrives at the remote host when, and in the order, it is entered. Imagine the fun of trying to edit a file in which each line is sent as a datagram: some lines might never arrive, others could arrive several times.

Network designers have recognized these limitations and have set about developing a system analogous to the public telephone network's 'switched circuit' known as a **virtual circuit**. A virtual circuit is a path through a wide area network that is maintained for the duration of the communication (as is a switched circuit in the telephone network). The data entered by the user follows the same path through the circuit until the communication is ended. If another communication to the same destination is started at a later time, the path taken can differ from the previous one because of traffic conditions and changing topology. Virtual circuits are reliable: the data supplied by a transmitting DTE arrives in the order sent, without loss, duplication, or error, at the receiving DTE. Over a period of time, there can be hundreds of simultaneous virtual circuits established across the network.

A virtual circuit consists of three distinct phases:

Call establishment. The calling party (a DTE) must indicate to its node that a virtual circuit is requested: a request that includes the address of the called party (normally a DTE attached to another node). This information is put into a **call request** packet that is routed through the network by the various nodes on the path between the two DTEs. The network's routing algorithm dictates the path taken by the call request packet.

As the call request packet progresses through the network, a logical path is established between the nodes on the network for this one specific call. Each node is responsible for maintaining its part of the virtual circuit for the duration of the call in its **virtual circuit table**. When the call request packet finally arrives at the destination node, the destination DTE is 'called', and if the call can be accepted, a **call connected** packet is returned to the calling DTE. However, if the call cannot be accepted (for example, if the DTE is inactive) the call will be cleared by the called node returning a **call clearing** packet to the calling DTE.

Although a virtual circuit itself is full duplex, the path created by the call request packet need not be full duplex. For example, the call request packet could set up a simplex channel; in these situations, the call connected packet establishes a return simplex channel. It is possible that the two halves of the virtual circuit take completely different paths through the network.

Data transfer. Once the call has been established, data transfer can take place. Data sent by either DTE passes through the network in a **data packet** using the virtual circuit created in the call establishment phase. Since there can be any number of virtual circuits in use, each packet is sent with an indication of its virtual circuit, the **virtual circuit number**. As the data packet arrives at each node, the virtual circuit number, used in conjunction with the virtual circuit table indicates the next leg of the packet's journey.

To ensure the reliability of the data transfer, data packets are acknowledged, typically as they move between nodes. In some cases, there can be **end-to-end** acknowledgements, confirming the arrival of the packet at the destination node.

Call clearing. As in any communication, there comes a time for the connection between the two DTEs to be terminated. Since there are a finite number of possible virtual circuits in the network (there is a limit to the size of the virtual circuit tables), both the DTEs *and* the nodes involved in a virtual circuit must be informed that the call is being terminated.

A call is terminated by either DTE sending a **call clearing** packet. The packet passes through the original path created during the call establishment phase; each node is responsible for freeing its part of the virtual circuit. The normal response to a call clearing packet is a confirmation packet.

Although it may appear that a single virtual circuit number is used to identify a virtual circuit through the network, this is seldom the case since the overheads associated with maintaining globally unique virtual circuit numbers outweigh their benefits. Instead, what appears to be a single virtual circuit is usually made up of a number of separate virtual circuits, allocated on a node-by-node basis. For example, consider the virtual circuits shown in Figure 13.15, that utilize the links connecting nodes X, Y, and Z, in which virtual circuits exist for DTEs A and D, as well as DTEs B and C.

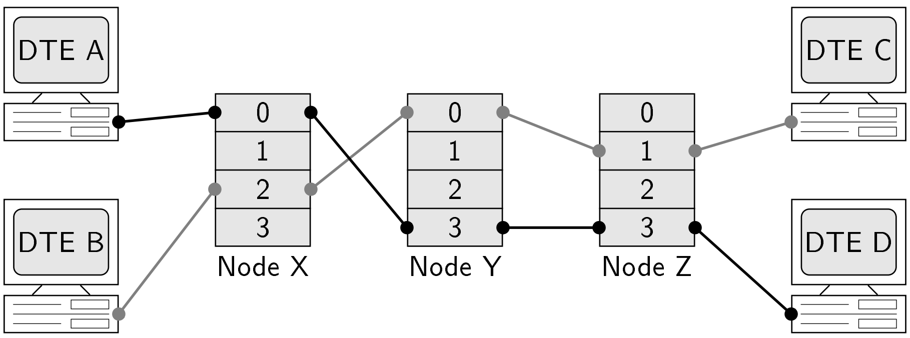


Figure 13.15: Virtual circuits connecting DTE A to DTE D and DTE B to DTE C

In the above example, process B communicates with process C through node X's virtual circuit number 2, whereas C uses virtual circuit number 1 on node Z. Since the virtual circuit numbers change from link-to-link, the node is for responsible changing the packet's virtual circuit number prior to being transmitted out the next link. Consider a packet sent from B to C: the initial virtual circuit number is 2, which is changed to 0 when the packet is sent to node Y; at node Y the number is changed again, to 1, and then sent to node Z, which forwards the packet to process C.

To distinguish between datagrams and virtual circuits, datagram services are often called **connectionless** while virtual circuits are called **connection-oriented**.

Many wide area networks offer combinations of the above services. For example, virtual circuits for interactive users may be needed during working hours, but during the quiescent periods at night, large volume transfers can be sent using a messaging service.

13.3.6 Node Design

The exact functions of a node depend upon a number of factors such as the protocol used by the wide area network and the protocols supported by the DTEs. However, there are a number of generic features that a node should support:

- the transmission and reception of packets from other nodes;
- the routing of packets from one node to the others; and
- controlling the DTEs' network access.

Although it is possible to design a node so that all of these functions are performed in one large maze of software, network designers have, over the years, come to recognize that dividing node software into **layers** or distinct parts offers a number of benefits, including:

- faults are easier to isolate and fix;
- software testing is simplified; and
- one layer can be modified while isolated from the other layers.

The most natural (and perhaps obvious) division of the node is between that part of the node which controls the network and the part that allows the DTEs access to the network. The network-part is responsible for forwarding packets and maintaining the virtual circuits; while the DTE-part allows DTEs to send and receive information across the wide area network. The entire structure is depicted in Figure 13.16.

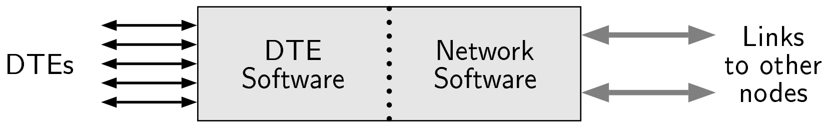


Figure 13.16: Functional divisions of a node

Within each of these two broad divisions, further layering is possible.

Network Software Layering

Consider the functions associated with the transmission and reception of packets in a network supporting virtual circuits or messaging:

- nodes are connected to other nodes by links. A single node may be associated with a number of links; not all of which will support the same medium (for example, some may be high speed synchronous lines, while others might be asynchronous, and still others could be satellite).

  In many wide area networks, the lowest layer is known as the **Physical Layer** and defines the electrical and mechanical properties of the link. The nodes at the end of each link must agree to the same Physical Layer protocol.

- Both virtual circuits and messaging require that packets sent across a link arrive undamaged and in the correct sequence at the receiving node. Since the underlying Physical Layer offers only a transmission medium, another layer is required, supporting the following features:

  - a device driver specific to the underlying medium. The device driver is responsible for the transmission and reception of packets.
  - a reliable, node-to-node protocol, such as stop-and-wait or Go-Back-N. This feature ensures that any packet sent across the link will either arrive correctly or will be received in error and a retransmission requested. (In a datagram network, the node-to-node protocol need not be reliable.)

  This layer, often referred to as the **Data Link Layer**, is intended to ensure that packets sent across the underlying link arrive at the next node in the correct order and error-free. Although the device driver will be unique to each different type of link, the reliable protocol software need not be unique to the link. That is, packets are formatted with the correct sequence number and control fields by the protocol software, while the underlying device driver and the Physical Layer simply transmit the information supplied. Once again, the argument for using a single set of software deals primarily with maintenance: by using one set of software, 'fixes' are universal.

- Finally, calls must be established and packets routed through the network. Since these functions cannot be justifiably performed by either the Physical Layer or the Data Link Layer, a third layer, the **Network Layer** is introduced. The Network Layer is responsible for a number of functions, including:

  - inspecting packets received by a Data Link Layer to determine the destination of the packet. The packet may be intended for a DTE attached to the local node, in which case the Network Layer supplies the information in the packet to the DTE. Or the packet may be destined for a remote DTE, requiring the Network Layer to forward the packet to the Data Link Layer that will eventually lead to the destination DTE.
  - taking information from the DTE (typically data and the address of the intended destination) and passing it in a packet to the correct Data Link Layer for eventual transmission. Additionally, call request and call clearing packets are also handled by the Network Layer, and forwarded to a Data Link Layer for transmission.

Should the destination DTE be attached to the same node as the transmitting DTE, the Network Layer need not attempt to transmit the information; instead, the information can be supplied directly to the specified DTE.

The Network Layer acts as a multiplexer both to the DTEs that are attached to node and to the links.

The relationship between the various layers, the DTEs and connections to other nodes is shown in Figure 13.17.

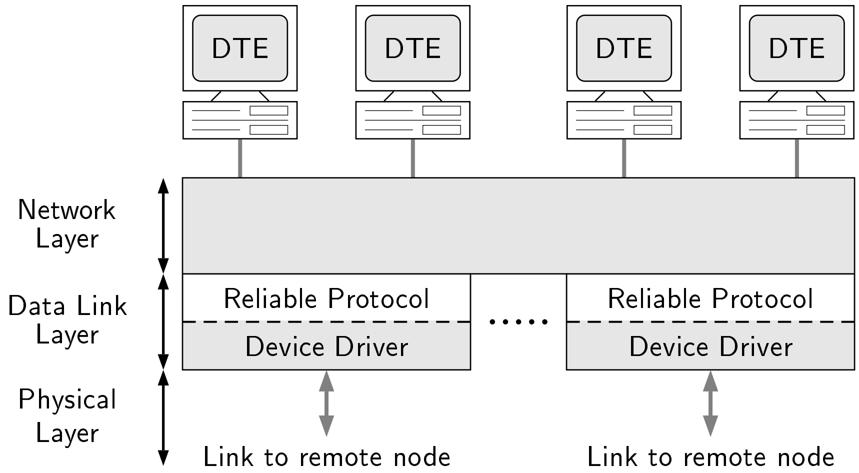


Figure 13.17: A node's layers and their relationships

The concept of layering involves more than simply dividing tasks into logical functions: it encompasses data structures and in some respects, programming style. Many layered communication systems define a single layer structure and apply this structure to all layers in the system:

- each layer has its own protocol and protocol conventions. Communications occur between **peer-entities** *within* a layer. For example, a wide area network's Network Layer has a set of protocols specific to the Network Layer. These protocols deal with the creation, maintenance, and termination of virtual circuits (or perhaps simply the routing of datagrams). Whereas in the Data Link Layer, protocols might exist for the reliable transfer of information between nodes.

- although communications occur within a layer between peer-entities, the entities within a layer use the **communication services** offered by the underlying layer. At a minimum, the services associated with a layer consist of the transmission and reception of information. The communication system may offer features that permit a lower layer to return error status information to the higher layer.

- since each layer has its own protocol and uses the services of the underlying layer, it is necessary to **encapsulate** one layer's packet structure in the packet structure of the underlying layer. The encapsulation involves adding control information specific to a layer (such as sequence numbering and error checking information). For example, a packet sent by a DTE in the above layered system could proceed through the sequence of encapsulation steps shown in Figure 13.18.

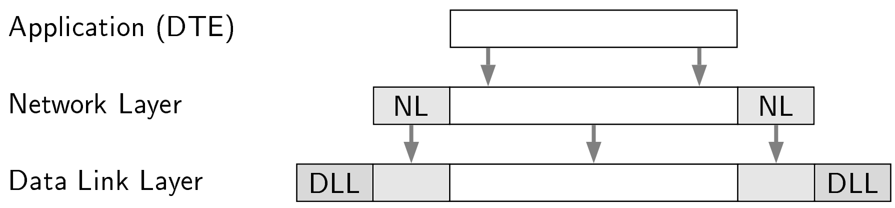


Figure 13.18: An example of encapsulation

Finally, the data within the packet is subject to the rules associated with transmitting information in the Physical Layer. For example, bit-stuffing or *DLE* insertion could occur.

When the packet is received, the control information associated with a specific layer is removed before passing the information to the layer above.

- the path taken by a DTE's packet through a wide area network may take it through many intermediate nodes. For example, consider Figure 13.19, in which an application on a DTE attached to node A is to communicate with an application attached to node C through an intermediate node, B.

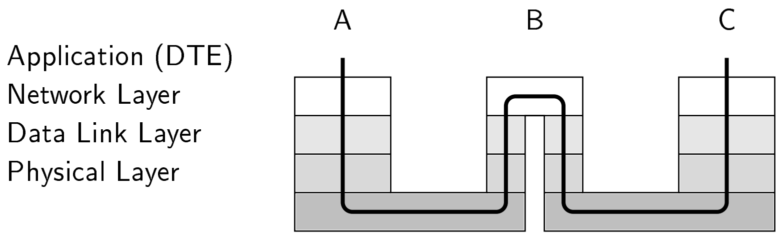


Figure 13.19: The path taken by a packet from node A to C via node B

A packet sent by the application attached to node A is encapsulated by each layer on node A and eventually sent to node B. The packet arrives at node B's network layer, which is responsible for routing the packet (by consulting either a routing table or a virtual circuit table). In either event, the packet is re-encapsulated and sent out the link attached to node C. At node C, the network layer supplies the packet to the destination application.

The Node-DTE Interface

The node and its DTEs must interact when communications across the wide area network are to take place. The concept of layering still applies between each DTE and the node; in this case, the DTE must inform the Network Layer of the action to be taken with the information supplied; for example:

DTE to Node. The DTE can request a call to be made to another DTE on the network, data can be sent and received from a remote DTE, and the DTE can request that the call be cleared.

Node to DTE. The node can supply the DTE with requests for setting up a call, data from a remote node can be supplied to the DTE, the node can indicate to the DTE that the call is being cleared.

The separation of the DTE from the node is a logical but somewhat arbitrary one since there are many situations where a DTE cannot communicate directly with a node and other situations in which the node and the DTE are indistinguishable. Consider the following:

DTE and Node Combined. In an attempt to reduce costs, many networks have node software that is intended to function on interconnected DTEs (i.e., the DTEs themselves are the nodes). The DTEs communicate via leased lines or, in some cases, dial-out facilities. Packets are formatted by software (typically within the operating system) and forwarded to the Network Layer software.

In this design of network, not only are costs reduced, but the features offered by the operating system (for example, file storage for messaging), can simplify the design of node software.

DTE and Node Separate. In most commercial wide area networks, the node and the DTE are separate and distinct. If the DTE is programmable, packets can be formatted by the DTE according to the Network Layer's requirements and sent to the node for transmission. However, if the DTE is simply an interactive terminal (or a computer emulating a terminal), a number of problems arise; for example, where is the data to be formatted into a packet and how does the node determine the intended destination of a packet?

The solution adopted by many wide area networks is to develop a software package known as a **PAD** or **Packet Assembler/Disassembler** that on the DTE-side emulates a host to which the DTE communicates and on the node-side, generates packets based upon the Network Layer's requirements (the node can be connected to other DTEs and PADs).

As shown in Figure 13.20, an asynchronous terminal could communicate with a PAD, which encodes and decodes Network Layer packets.

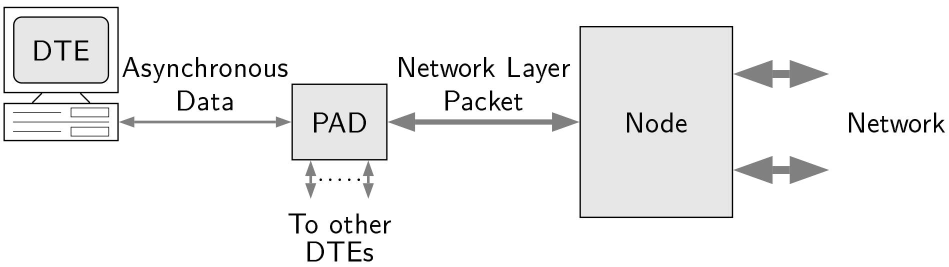


Figure 13.20: Connecting to a network via a Packet Assembler/Disassembler

An alternative to having a separate PAD is shown in Figure 13.21, in which the PAD is part of the node. The number of DTEs that could connect to an individual PAD would be implementation dependent.

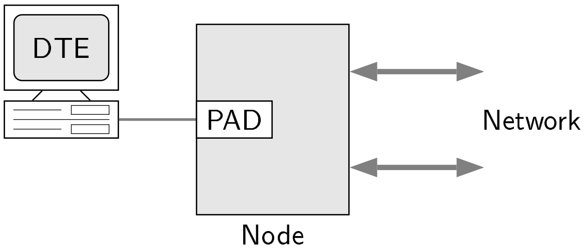


Figure 13.21: A PAD can be part of a node

PADs are not restricted to asynchronous DTE communications, PADs are available to support Bisync and other such protocols.

13.4 Wide Area Network Protocols

From the discussion in the previous section it should be apparent that there is no single wide area network protocol: each layer is associated its own set of rules, standards, and protocols. In some cases, there can be numerous choices of protocol within a single layer.

13.4.1 Physical Layer Standards

The physical layer connects pairs of nodes through a communication channel. Physical layer standards define the mechanical, electrical, functional, and procedural rules associated with the communication channel. Examples of physical layer standards include RS-232-C, RS-449 (RS-423-A and RS-422-A), X.21, and X.21*bis*.

13.4.2 Data Link Layer Protocols

The data link layer is responsible for formatting information for transmission across the underlying physical layer. Typically, the data link layer offers a reliable communication service to the higher layers, by maintaining error detection and flow control. In a network supporting datagrams only, there is not the same need for reliability.

There are a number of different data link layer protocols: some networks use Bisync or SDLC, neither of which are considered standards. Internationally adopted standards, all based upon SDLC (see Chapter 10), include:

ADCCP. The Advanced Data Communication Control Protocol, developed by the American National Standards Institute (ANSI) as ANSI standard X.366.

HDLC. The **High-Level Data Link Control** protocol, a point-to-point synchronous protocol intended for Data Link Layer peer communications developed by the International Organization for Standardization (ISO). HDLC is defined in ISO documents DIS 3309 and DIS 4335.

LAP-B. The Link Access Procedure-Balanced protocol is defined by ITU-T as part of the X.25 protocol (see below). LAP-B is a subset of HDLC.

13.4.3 Network Layer Protocols

Probably the best known Network Layer protocol is **X.25**, an ITU-T standard developed to handle virtual circuits between DTEs and a wide area network. In X.25, the node is referred to as the **DSE** (or Data Switching Exchange) and the connection between a DTE and a DSE is the **DCE** (or data circuit *terminating* equipment); the protocols used within the network are not defined by X.25.

X.25 is actually an ITU-T *recommendation* as opposed to a specific protocol, consisting of a detailed description of the Physical, Data Link, and Network Layers:

Level 1. The physical, electrical, functional, and procedural characteristics needed to establish, maintain and disconnect the physical link between the DTE and the DCE. The DTE/DCE interface characteristics are defined in Recommendations X.21 and X.21*bis*.

Level 2. the link access procedure for data interchange across the link between the DTE and the DCE. Level 2 access is defined by the LAP-B protocol.

Level 3. the packet format and control procedures for the exchange of packets containing control information and user data between the DTE and the DCE. Level 3 is also referred to as the **Packet Level DTE/DCE Interface** and describes X.25 virtual circuits. Many public data networks have adopted the X.25 Level 3 protocol for network access.

The structure of X.25 is shown in Figure 13.22.

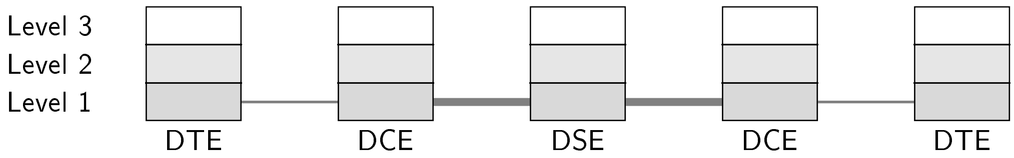


Figure 13.22: The structure of X.25

Level 3 of X.25 defines two types of virtual circuit: a **virtual call** (also known as a **switched virtual circuit** or SVC) and **permanent virtual circuit** or PVC. Switched virtual circuits are those described in previous sections, while permanent virtual circuits are those in which the virtual circuit is always available for data transfer. There is no call establishment nor call clearing phase associated with a permanent virtual circuit.

The three phases associated with an X.25 virtual circuit are as follows:

Call Establishment. A call can be established by any DTE that has an available virtual circuit number. The call must be sent in a Call Request packet; the format of the Call Request

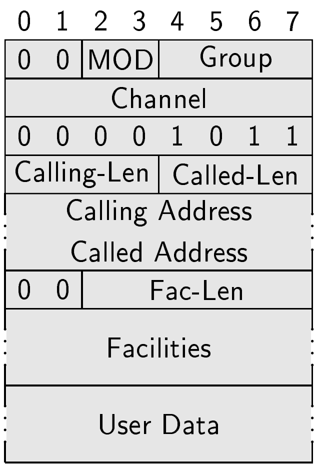


Figure 13.23: The X.25 Call-Request Packet

packet is shown in Figure 13.23 (note, ITU-T labels its bits from left-to-right, with bit 7 being the low-order bit; an 8-bit quantity is referred to as an **octet**).

The various fields in the Call Request packet are defined as follows:

MOD. MOD informs the called DTE which data packet sequence numbering scheme the calling DTE is using: either 8 or 128 (see Data Transfer, below). If the two bits are *01*, then modulus-8 is used; however, if the value of MOD is 10, modulus-128 is used. The two other possible bit patterns are not defined. All packets are sent with the MOD identifier.

Group/Channel. A 12-bit virtual circuit number made from a 4-bit Group number and an 8-bit Channel number. All packets are sent with the 12-bit virtual circuit number.

Calling-Len and Called-Len. The calling DTE's address length (Calling-Len) and called DTE's address length (Called-Len). Addresses can be up to 14 digits in length.

Calling and Called Address. The source and destination addresses. The network address format is defined by ITU-T X.121: three digits for the country, one for the network within the country, and ten to identify the DTE. Each digit is four-bits long (i.e., a binary coded decimal digit). Within a network, the organization of the ten digits is defined by the network administration. If the final octet contains a single digit (i.e., 4-bits), the remaining four bits are zero filled.

Fac-Len. The length, in octets, of the Facilities field.

Facilities. The Facilities field allows the calling DTE to either indicate or request one or more facilities. The exact facilities supported can vary from network to network, but some of the more common ones include reverse charging (the calling DTE wants the called DTE to accept the network charges for this call), use of a non-standard window or message size, and whether the calling DTE is a member of a specific **Closed-User Group** or **CUG** (a CUG is a facility which restricts communications to a specific group of DTEs).

Facilities are sent as sequences of octet pairs: the first octet, the facility code, indicates the facility requested; and the second octet, the facility parameter, is the value associated with the requested code. For example, to request reverse charging, the facility code is 00000001, while the parameter is either 00000000 (no reverse charging requested) or 00000001 (reverse charging requested). A specific closed-user group can be requested by supplying a facility code of 00000011 followed by the 8-bit closed-user group number.

User Data. Data supplied by the calling DTE that is passed, unchanged, to the called DTE.

The Call Request packet arrives at the DTE called as an Incoming Call packet. If the DTE called can accept the call, it returns a Call Accepted packet, which is received by the calling DTE as a Call Connected packet. The Call Accepted/Connected packet format is shown in Figure 13.24.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>0</td><td>0</td><td colspan="2">MOD</td><td colspan="4">Group</td></tr>
<tr><td colspan="8">Channel</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Figure 13.24: The X.25 Call-Connect/Call-Accept packet structure

The overall flow of information in the X.25 call establishment phase is presented in Figure 13.25.

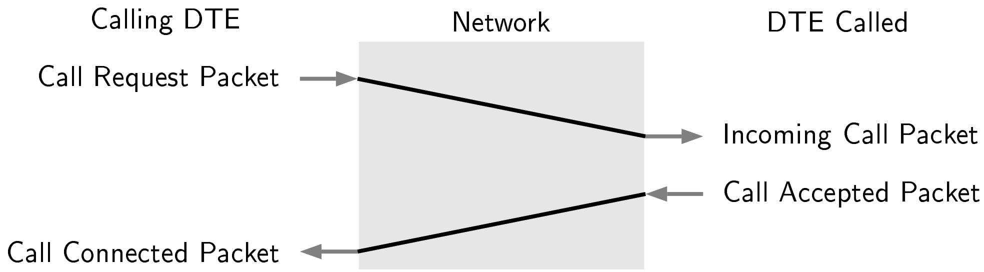


Figure 13.25: The steps in the call-establishment phase

At this point, data transfer may commence. However, if the call cannot be accepted, the DTE called responds with a Call Clearing packet (see below). Possible reasons for the call not being accepted include the refusal of the called DTE to accept reverse charging or that the called DTE itself was in the process of making a call.

Data Transfer. Once the virtual circuit has been established, data transfer can take place. Unless otherwise specified, the virtual circuit is full duplex, meaning that the DTEs may transmit at any time, as long as their transmit window is open. To ensure that packets are not lost and to permit flow control, X.25 implements a Go-Back-N sliding window protocol for Level 3 Data packets. Each Data packet is sent with both its sequence number and the sequence number of the next expected packet. The format and contents of a Data packet is shown in Figure 13.26.

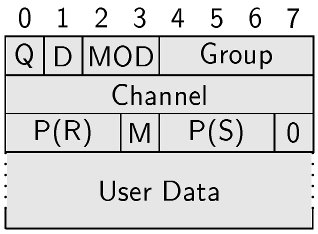


Figure 13.26: The X.25 Data packet

The Q-bit is the **data qualifier bit** and when cleared, signals the DTE to forward the User Data to the application associated with the virtual circuit number. However, when set, the Q-bit allows the DTEs to exchange DTE-specific control data; in these situations, the User Data is not supplied to the application associated with the virtual circuit number. The D-bit, when set, requests an end-to-end acknowledgement for this packet *after* is has been successfully delivered to the application by the DTE (X.25 acknowledgements are discussed below).

P(R), the packet receive sequence number, and P(S), the packet sent sequence number, are the next-expected and the current packet sequence numbers, respectively. Sequence numbering is performed modulo-8 (i.e., 0 though 7) or modulo-128 (i.e., 0 through 127), depending upon the value of MOD. Modulo-128 requires that P(R) and P(S) each occupy one octet. The sequencing allows the DTEs to perform end-to-end sequence number checking as well as flow control by withholding acknowledgements.

The User Data field holds a maximum of 128 octets unless otherwise specified by the network. Other valid maximum data lengths must be a power of 2 taken from the following list: 16, 32, 64, 256, 512, and 1024 octets. The M-bit is set if the current data packet is full and a subsequent packet contains data that is to be concatenated to the current data. For example, if an application sends a 132 byte message, two data packets are sent: the first, a packet of 128 octets (with the M-bit set), followed by a second packet of 4 octets (with the M-bit cleared). The remote DTE then assembles the 132 bytes before forwarding them to the remote application.

There are three end-to-end acknowledgement packets which correspond to the SDLC/HDLC Go-Back-N protocol acknowledgement frames: RR, RNR, and REJ. The packet formats are shown in Figure 13.27 (P(R) denotes the sequence number of the next expected packet).

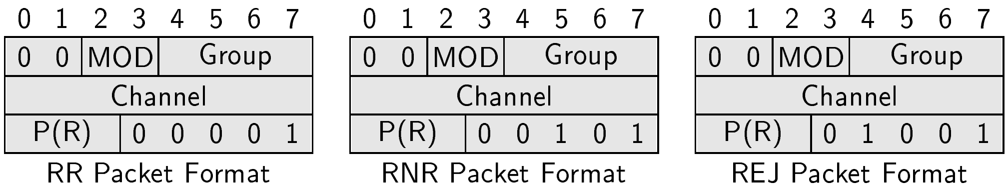


Figure 13.27: X.25 end-to-end acknowledgement packet formats

In addition to sending and receiving sequenced data packets, X.25 also allows for the transmission and reception of unsequenced Interrupt packets. Interrupt packets are intended to convey signalling information (such as a break signal) on a virtual circuit. The response from the DTE and DCE receiving the Interrupt packet is an Interrupt Confirmation packet; shown in Figure 13.28.

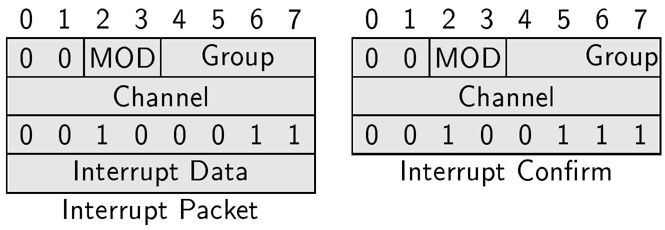


Figure 13.28: The X.25 Interrupt and Interrupt-Confirm packets

The exchange of data takes place through data packets across the virtual circuit. Acknowledgements can be piggybacked on Data packets or sent as RR packets; a typical cycle is shown in Figure 13.29.

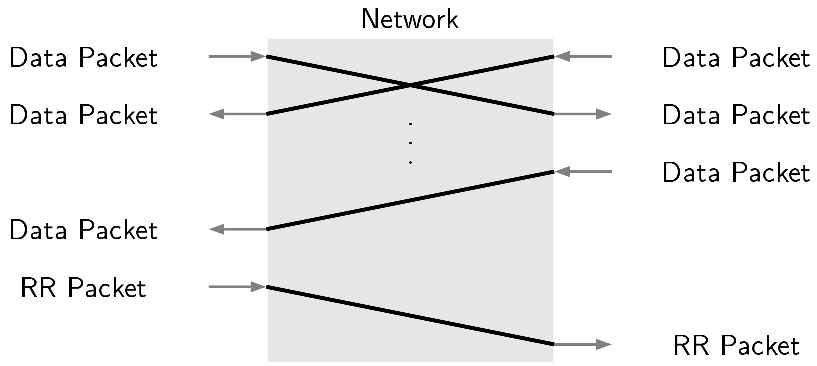


Figure 13.29: Data transfer using X.25

Call Clearing. Call clearing can be initialized by either DTE issuing a Clear Request packet. The DCE connected to the DTE issuing the Clear Request responds with Clear Confirmation, freeing the virtual circuit. When the packet arrives at the remote DCE, the DCE issues a Clear Indication packet to the remote DTE. The remote DTE then frees its virtual circuit and responds to its DCE with a Clear Confirmation.

The format of the Clear Request/Indication packets is given in Figure 13.30. The codes for the Clearing Cause field are given in Table 13.5.

The format of the DTE and DCE Clear Confirmation packet is presented in Figure 13.31.

The flow of packets for an X.25 call clearing is shown in Figure 13.32 (note that the Clear Confirmation packets are *local* rather than end-to-end).

0	1	2	3	4	5	6	7
0	0	MOD		Group			
Channel							
0	0	0	1	0	0	1	1
Clearing Cause							

Figure 13.30: The X.25 Clear-Request/Clear-Indication packet

Table 13.5: X.25 Call Clearing Cause field values

Code	0	1	2	3	4	5	6	7
DTE Clearing	0	0	0	0	0	0	0	0
Number Busy	0	0	0	0	0	0	0	1
Out of Order	0	0	0	0	1	0	0	1
Remote Procedure Error	0	0	0	1	0	0	0	1
Number Refuses Reverse Charging	0	0	0	1	1	0	0	1
Invalid Call	0	0	0	0	0	0	1	1
Access Barred	0	0	0	0	1	0	1	1
Local Procedure Error	0	0	0	1	0	0	1	1
Network Congestion	0	0	0	0	0	1	0	1
Not Obtainable	0	0	0	0	1	1	0	1

0	1	2	3	4	5	6	7
0	0	MOD		Group			
Channel							
0	0	0	1	0	1	1	1

Figure 13.31: The X.25 Clear-Confirmation packet

In addition to the three phases described above, X.25 also defines two other phases, **Reset** and **Restart**:

Reset. The Reset phase allows a DTE to reset a specific virtual circuit: all sequence numbers are cleared and any data or interrupt packets associated with the circuit are discarded. Either DTE can reset a virtual circuit by issuing a Reset Request packet; the remote DTE resets the remote virtual circuit and responds with a Reset Confirm.

The Reset Request/Indication packet format is presented in Figure 13.33. The codes for the Resetting Cause field are given in Table 13.6.

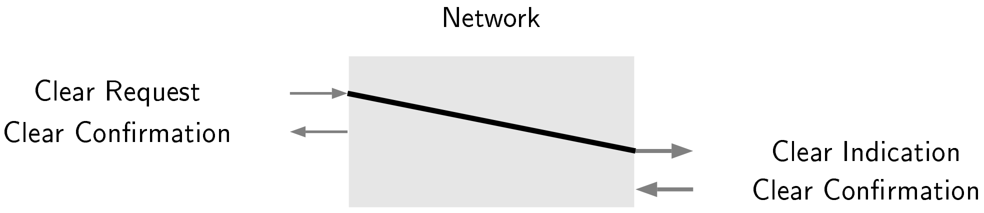


Figure 13.32: The steps in the Call-Clearing phase

0	1	2	3	4	5	6	7
0	0	MOD		Group			
Channel							
0	0	0	1	1	0	1	1
Resetting Cause							
Diagnostic Code							

Figure 13.33: The X.25 Reset-Request/Reset-Indication packet

Table 13.6: X.25 Resetting Cause field values

Code	0	1	2	3	4	5	6	7
DTE Reset	0	0	0	0	0	0	0	0
Out of Order	0	0	0	0	0	0	0	1
Remote Procedure Error	0	0	0	0	0	0	1	1
Local Procedure Error	0	0	0	0	0	1	0	1
Network Congestion	0	0	0	0	0	1	1	1

The Diagnostic Code has a value of zero. The format of the DTE and DCE Reset Confirmation packet is shown in Figure 13.34.

Restart. The Restart phase allows a DTE to clear *all* of its switched virtual circuits and to reset its permanent virtual circuits. Either the DTE or the DCE can request a restart by issuing a Restart Request packet; the response is a Restart Confirm. Any switched virtual circuits must be re-established after a restart.

The format of the Restart Request/Indication packet is as given in Figure 13.35.

The codes for the Restarting Cause field in the Restart Request/Indication packet given in Table 13.7.

The format of the DTE and DCE Restart Confirmation packet is shown in Figure 13.36.

0	1	2	3	4	5	6	7
0	0	MOD		Group			
Channel							
0	0	0	1	1	1	1	1

Figure 13.34: The X.25 Reset Confirmation packet

0	1	2	3	4	5	6	7
0	0	MOD		Group			
Channel							
1	1	1	1	1	1	1	1
Restarting Cause							

Figure 13.35: The X.25 Restart-Request/Restart-Indication packet

Table 13.7: X.25 Restarting Cause codes

Code	0	1	2	3	4	5	6	7
Local Procedure Error	0	0	0	0	0	0	0	1
Network Congestion	0	0	0	0	0	0	1	1

13.4.4 Network Access Protocols

From the above discussion on X.25, it is apparent that network access is handled by DTEs that support X.25. The various X.25 levels can be embedded in a DTE's operating system and users or applications requiring network access can follow the steps required by the specific operating system. Clearly, without some means of generating and recognizing X.25 packets, DTEs such as asynchronous terminals are unable to access an X.25 network.

Fortunately, ITU-T has also developed a series of PAD (Packet Assembler/Disassembler) protocols that permit non-X.25 DTEs access to wide area networks. These standards, **X.3** (PAD parameters), **X.28** (terminal-PAD interface) and **X.29** (PAD DTE interface) define how an asynchronous DTE (i.e., a terminal) can access a remote X.25 DTE through a packet switched network using a PAD.

Since different terminals and their users will have different requirements, X.3 specifies a number of features that can be tailored to a specific terminal. (These features are maintained in a **profile** which is associated with the terminal for as long as it is connected to the network.) For example, the X.3 PAD parameters allow either the remote DTE or the local terminal to modify things such as:

- whether the terminal can communicate with the PAD to change PAD parameters (PAD parameter 1).

0	1	2	3	4	5	6	7
0	0	MOD		Group			
Channel							
1	1	1	1	1	1	1	1

Figure 13.36: The X.25 Restart Confirmation packet

- whether the PAD is responsible for echoing characters to the terminal (PAD parameter 2). If the terminal expects remote echoing but the DTE does not support remote echo, the PAD can be requested to echo each character entered.
- the character(s) used to signal the PAD that a packet is to be forwarded (PAD parameter 3). Packets can be forwarded when 'full' (i.e., after 128 or 256 characters are entered), when a specific character is detected (such as a carriage return), or on each character (useful when working with a full-duplex screen editor).
- whether the PAD recognizes the flow control characters X-ON and X-OFF; allowing the user to suspend output to the terminal (PAD parameter 12).
- if the PAD supports line-at-a-time input (that is, the end-of-line character causes the line to be forwarded in a packet), the user should be allowed to make changes to the line before the line is sent. X.3 defines editing character selection, in which the character delete, line delete, and line redraw keys can all be defined (PAD parameters 15, 16, 17, and 18).

A terminal connected to a PAD operates in one of two modes: either the terminal is communicating with the PAD (for example, for call setup or call clearing) or the terminal is communicating with the remote DTE (for data transfer). When communication with the PAD is required (for example, to change one or more X.3 parameters), the user issues an escape sequence which is recognized by the PAD, data transfer then ceases while the user communicates with the PAD. The commands and associated responses are defined in X.28 and permit the user to establish a call, clear a call, query PAD parameters, and change PAD parameters. When data is to be sent to the DTE, the PAD is responsible for formatting the data in an X.25 packet.

The DTE can also communicate with the PAD through the X.29 protocol which defines a series of messages intended for use between the DTE and the PAD. These messages allow the DTE to query and select different PAD parameters as well as signalling that the call is to be cleared. The ability to change PAD parameters is useful when, for example, the user activates a full-screen editor. At this time, the DTE can request that the PAD forward every character while the editor is in use.

The Q-bit in the X.25 packet allows the PAD to determine which packets are intended for the terminal (Q-bit is cleared) and which packets are intended for the PAD (Q-bit is set). X.29 packets are sent with the Q-bit set.

13.5 Examples of Wide Area Networks

Wide area networks fall into two broad categories:

- Those that have been developed by a company that are intended to work with the company's equipment.

  A good example of this is **SNA**, or Systems Network Architecture, developed by IBM, which consists of both hardware and software. SNA is a layered network architecture consisting of a physical layer (Physical Control); a data link layer (Data Link Control) which supports SDLC; and a network layer (Path Control). The path control layer is responsible for packet routing and flow control.

  DECNET is another example of hardware and software products developed by a manufacturer (Digital Equipment Corporation) specifically for their own equipment. DECNET is part of Digital's **DNA** (Digital Network Architecture), a layered network architecture. DECNET has a number of objectives, including: the connection of Digital computers and operating systems, to support 'any' physical layer (i.e., full duplex, half duplex, synchronous, asynchronous, serial, parallel, and so on), and to offer facilities to support interprocess communications and file transfer. The data link layer is supported by **DDCMP** (Digital Data Communication Message Protocol), a sliding window protocol with a window size of up to 127.

- Those that conform to international standards, such as X.25.

  Most public packet switching networks, such as **Datapac** (Canada) and **Transpac** (France), support networking standards such as X.3, X.25, X.28, and X.29. There are a number of arguments for supporting international standards, rather than the products of a single vendor, the most compelling being that supporting international standards permits the interconnection of equipment from different manufacturers. Another argument, to be pursued in later Chapters, is that network interconnection can be an easier task if common protocols are adopted.

13.6 Summary

A wide area network is a collection of nodes interconnected by a series of links; there is no single wide area network topology, although most are partially connected meshes. The purpose behind a wide area network is to offer users a shared communication facility which is made available 'on demand'. This is different from the telephone system which must devote an entire circuit to a pair of DTEs for the duration of their communication.

Wide area networks achieve shared communication through the use of packet switching, a technique whereby information is transmitted in a data structure known as a packet that is routed through the network from node to node. Routing can be achieved through any number of techniques, although most commercial wide area networks use some form of routing table.

There are three types of packet service that a wide area network can offer:

- datagrams, in which the packet is put onto the network and the various nodes make a 'best effort' attempt at having the packet reach the intended destination. There is no guarantee that the datagrams will reach their intended destination in the order they are sent or even if they will reach the destination.

- messaging, in which a single message is broken into a series of packets and transmitted between nodes. A message, unlike a datagram, is reassembled at each node at which it arrives; the node is responsible for determining the next leg in the route and scheduling a time for the transmission of the message.

- virtual circuits, which closely resemble a telephone call, consisting of a number of distinct steps. The first, call establishment, sees a call request packet moving from node to node, defining a path (the virtual circuit) through the network. Once the path is established and both ends agree to communicate, the second step can be instituted: the transfer of data across the route defined by the call request packet. When either or both DTEs agree to terminate the call, a call clearing packet can be sent across the network, freeing the different stages across the network.

Wide area networks have also illustrated two important concepts in data communications: layering, in which software is divided into discrete parts, each performing a specific task; and standards, which ensures that DTEs from different manufacturers can communicate. A good example of a layered wide area network standard is ITU-T's X.25, a three layer architecture defining a physical layer, a data link layer, and a network layer.

13.7 Exercises

1. Why isn't is necessary for the data link layer protocol to be reliable in a datagram network?
2. How does X.25 distinguish data packets from other packets?
3. Show the layers necessary for a messaging system and for a virtual circuit.
4. If a station has several processes, how could a call request packet be routed to the correct process?

Chapter 14

Internetwork communications

14.1 Introduction

A computer can communicate with another computer if it is interconnected via some form of underlying network. In the 1970s and 1980s, many organizations relied upon **wide area networks** (developed and controlled by telephone companies) to support their computer communication requirements. The gradual acceptance of the **local area network** meant that by the late 1980s, many of these same organizations had their own private **intranetworks**.

However, some organizations could not afford the communication charges associated with wide area networks and their communication requirements would not permit sharing a intranetwork with other organizations. These organizations, notably a number of universities in the United States, with help from the U.S. military, developed the technology to address these communication issues.

The steps involved in creating an interconnected network or **internet** are considered in this Chapter. However, before creating an internet, it is instructive to consider the problems confronting the designers of the original Internet.

14.2 The Problem

Consider the following scenario, five independent networks, each with its own addressing scheme, packet size, and internal protocol, are to be organized so that any computer on any network can communicate with any computer on any other network. Table 14.1 highlights the differences between the networks, some of which are rings, others are buses.

Figure 14.1 shows the topology of the networks. Initially, none are interconnected.

There are at least four issues that must be addressed before a computer on one of these networks can communicate with a computer on a separate network:

1. *Packet size.* Different co-operating organizations may have different computer networks, meaning that a packet transmitted on one network might not 'fit' on another network.

Table 14.1: Network characteristics

Attribute	Network A	Network B	Network C	Network D	Network E
Topology	Ring	Ring	Bus	Bus	Ring
Packet size (bytes)	1024	512	1500	1500	3072
Address size (bites)	32	16	48	48	48
Speed (Mbps)	4	1	10	10	16

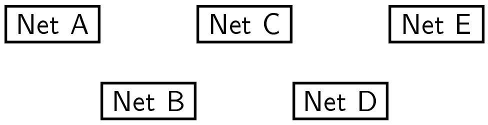


Figure 14.1: Network Layout

2. *Address structure and size.* As with packet size, different networks may support different address structures. Furthermore, if a computer on one network has the same physical address as a computer on a remote network, packets may be delivered to the wrong destination.

3. *Network protocol.* Each network may have a network protocol that is different from that of its neighbours, and the successful transmission of a packet on a network does not mean that the packet has been successfully received.

4. *Network interconnection.* The physical interconnection of networks can be achieved in a number of ways: a single computer may be shared by two networks or each network may have a computer with a connection to a computer on a remote network. The term **half-gateway** is often used to refer to computers that reside on one network and have a separate connection to a remote computer on another network (see Figure 14.2).

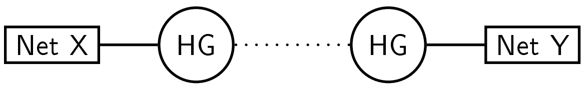


Figure 14.2: Interconnecting networks with a pair of **half-gateways**

Successful interconnection of these networks requires a solution to each of the above issues.

14.3 Network Interconnection

The first, and probably most obvious, step is to designate one or more computers on each network the task of sending packets to and receiving packets from neighbouring networks. These computers are referred to as **routers**. Although they exhibit similar characteristics to bridges and gateways, they are neither.

The topology chosen is dependent upon factors such as the distance between networks, whether a single router spans both networks or whether a router consists of a pair of half-gateways, the cost of each router, and projected traffic volumes.

If money is no issue, a fully-connected network may be possible; however, few organizations (even the U.S. military) have this luxury. Similarly, adopting a centralized system consisting of a single 'super-router' through which all communications pass, is usually avoided for reliability reasons. Most solutions opt for a **mesh** of links between the various networks, with one or more **routes** or **paths** between each network.

A network topology for the five networks described above is shown in Figure 14.3. It is important to note that in this interconnected mesh of networks, each network is 'willing' to transfer packets between remote networks. For example, network C transfers packets sent between computers on networks B and D; while these networks transfer packets sent by computers on network C.

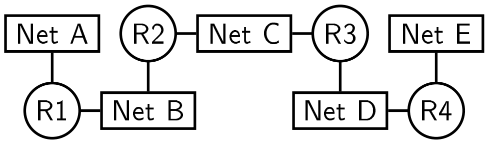


Figure 14.3: Network Interconnection creating an **internet**

Each network in an internet environment supports the transfer of packets across the network. Every packet will have a destination: some are destined for computers on the local network, while others are simply traversing the local network, en route to a computer on a remote or different network.

14.4 Identifying Computers

The interconnected network now consists of individual computers and specialized computers known as routers. Packets destined for computers on remote networks must pass through a network's router (or routers) in order to reach the intended destination.

In an ideal world, all computers would have the same address structure; however, since this may not be the case, it is necessary to develop a means whereby each computer can be identified in a uniform manner. There are actually two problems here: first, the size of the address, and second, the interpretation of the address.

14.4.1 Address Size

Prior to the acceptance of the Ethernet as a defacto LAN standard, different network manufacturers chose address structures that were in their own best interest. Since most addresses would be of fixed size and hardware specific, it would be impossible to adopt a common network-wide address structure.

This being the case, one possible solution would be to force all addresses to conform to the smallest possible physical address structure on the network. Those networks with addresses larger than the minimum would be forced to assign the unused bits a zero value (i.e., clear them). In networks with programmable NICs, this is a simple matter; however, networks with fixed-address NICs make this solution unworkable.

In the network interconnection example discussed above, there are three address sizes in use on the network: 48-, 32-, and 16-bits. If the smallest possible address size is to be used, all networks would be forced to adopt a 16-bit address structure.

Two more issues related to the problem of address size are:

- the addition of new networks with address sizes that are smaller than the pre-defined minimum. Networks with address sizes equal to or greater than the internetwork's minimum can be accommodated, whereas those with smaller address sizes cannot be supported.

- the decreasing number of possible computers as the address size gets smaller. For example, an address size of 16 bits permits no more than 65,536 possible computers, assuming that broadcast and multicast are omitted.

14.4.2 Address Structure

Broadly speaking, there are two ways in which a computer can be identified: by a **globally unique address** (somewhat like an Ethernet address) or by a **hierarchical address** (such as that found in the telephone system).

Globally Unique Addresses

Globally unique addresses are attractive in that once a computer has been allocated its address, it never need change it. Should the computer be moved to a new location, the same address is used.

This flexibility comes with a price. When a packet is sent to a computer on a remote network, each router must be able to examine the packet's destination address (i.e., the router must operate in promiscuous mode) and then determine whether the computer associated with the destination address is located on the local network or a remote one. The router can maintain one of two table structures:

- *a table of local addresses.* When a packet arrives at the router, the router can inspect the table; if the destination address is in the table, the packet can be discarded, otherwise it can be forwarded to an interconnected network. This approach ensures a small table, but can result in a flooded network as routers forward packets to any interconnected network, without regard to their possible destination network.

- *a table of remote addresses.* In this approach, the router maintains a list of all addresses and the router which leads to the packet's destination network. Although this will avoid flooding the network, in a large network, each router's table must contain sufficient space to hold the addresses of all possible destination computers.

Hierarchical Addresses

The best-known alternative to the global address structure is the hierarchical address structure. A hierarchical address consists of a number of **fields**; each field denotes part of the overall address. Examples of hierarchical addresses are commonplace, including a postal address such as:

```
struct postal_address
{
char addressee[32];
char street[32];
char city[32];
char state_or_province[32];
char country[32];
};
```

In the case of the `postal_address`, each letter is routed to the intended destination first using the `country` field, then the `state_or_province` field, the `city`, and finally the `street` field. (Once at the intended destination, further routing can take place, delivering the letter to the person specified in the `addressee` field.)

The hierarchical address has the advantage of exhibiting a structure that is shared by all participants, thereby making routing a simple task. Routing is specific to each level of the hierarchy; for example, once a letter has arrived at a specifice `state_or_province`, the postal clerk need only inspect the `city` field, as all other fields have no significance. This permits several cities in the same `state_or_province` to have the same `streets` and `addressees`.

A hierarchical address structure can be used in an interconnected computer network. In this case, two fields are required: a network field and a computer field, as shown in Figure 14.4. The network field specifies the computer's network, while the computer field specifies which computer on a given network.

Network Field	Computer Field

Figure 14.4: A hierarchical address for an internetwork computer

When compared to routing with global addresses, routing with hierarchical addresses is greatly simplified. First, the router need only inspect part of the address (the `network` field) as opposed to the entire address. Second, the table maintained by each router need only contain an indication as to the path the packet should take to reach the intended destination network. These tables, commonly referred to as **routing tables**, consists of at least two parts: the network name and the path that must be taken in order to reach the named network. Possible routing tables for the routers R1 and R2 are shown in Table 14.2.

The uniformity of the hierarchical address is also one of its limitations:

- Since both the `network` and `computer` fields have fixed lengths, each network is restricted to 2^c computers, where c is number of bits in the `computer` field.

Table 14.2: Possible routing tables for routers R1 (left) and R2 (right)

To Network	Via Network
A	–
B	–
C	B
D	B
E	B

To Network	Via Network
A	B
B	–
C	–
D	C
E	C

This can lead to an inefficient use of the available address space. The `computer` field must be large enough to accommodate the network with the largest number of computers; meaning that networks with a significantly smaller number of computers will have many unused addresses.

- A similar problem occurs with the `network` field, since it must be large enough to handle all possible networks. The space available for networks is also a power of 2, which can lead to an inefficient use of the address space if the number of networks is considerably less than 2^n (where n is the number of bits in the `network` field),
- By dividing the address into fields, the number of possible computers that can be on the interconnected network typically declines since the address space is not used efficiently.
- If network addresses are assigned arbitrarily, each router must have 2^n entries (where n is the number of bits associated with the `network` field). Entries that are not associated with a network must contain a flag that indicates that the network does not exist.

To illustrate these limitations, consider the sample network consisting of five networks. Assuming that the maximum number of computers on any one network will be 1024, then 10 bits must be reserved for the `computer` field (since 2^{10} is 1024). As a result, six bits are 'left over' for a total of 2^6 or 64 possible networks; with only five active networks, the maximum number of computers on the network is limited to 5120. If one of the networks is to support more than 1024 computers (but the size of the `network` field remains fixed), it may be necessary to assign more than one network number to a network.

Despite limitations such as these, most interconnected networks, such as the one being developed in this Chapter, use hierarchical addresses because it minimizes the size of the routing table.

14.5 Packet Size

The packet size issue is similar to that of the address size: the resulting interconnected network must adopt a common packet size to ensure that packets can be sent across network(s) with the smallest possible packet size.

It should be noted that adopting a minimum packet size is not as critical as adopting a common address structure. One technique that can be used is to allow a router to **fragment** any packet that it receives. Fragmenting, as the name suggests, involves dividing a packet into two or more smaller packets for subsequent transmission. It is the responsibility of another computer (typically the destination host as opposed to an intermediate router) to **reassemble** the packet (see Figure 14.5).

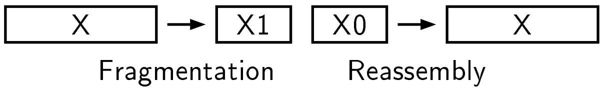


Figure 14.5: Packets can be fragmented and reassembled

Both fragmentation and the fact that a packet may be less than the maximum packet size suggests that it might be necessary to include additional control information in the packet. At a minimum, this control information should indicate the size (in bytes) of the packet.

14.6 Identifying Computers (Revisited)

The discussion in section 14.4 suggested that for an interconnected network to be successful, a common hierarchical address structure should be adopted by all routers and computers. A number of serious limitations were also highlighted in this section, notably that forcing all computers to program their NICs to the smallest possible address size could result in an extremely limited number of participating computers.

Before completely discarding this approach, it is worth reconsidering it in terms of a **virtual address** structure. That is, rather than forcing the underlying physical networks to conform to a common network structure, why not develop a (software) layer common to all computers that **maps** hierarchical addresses into physical addresses?

A virtual address consists of two parts: a **network number** and a **host** (i.e., computer) **number** (see Figure 14.6). The network number specifies the destination network, while the host number identifies a computer on the destination network. What is different about this approach is that no changes are required of the underlying network; the network keeps its packet and address structures. Each computer on each network is assigned a 'virtual' address consisting of a network number and a host number.

Network Number	Host Number

Figure 14.6: A virtual address

All computers attached to a common network share the same network number, while each host number is assigned to individual computers. No two computers *on the same network* share the same host number; similarly, no two networks share the same network number. This means that two computers on separate networks can safely be assigned the same host number.

14.7 Packet Routing

When a packet is to be transmitted, the transmitting software on the source computer is supplied with the packet and the virtual address of the destination. In an internetwork environment, the destination can be on either the local network or on a remote network.

14.7.1 Transmissions to the Local Network

The local transmission software can distinguish between the local and remote transmissions by inspecting the network number in the virtual destination address: if the network number is the same as that of the local network, then the transmission is to a local computer. The question is – which local computer?

Each computer on each network has its own unique host number; this host number corresponds to a physical address. If each computer on a network maintains a table of the host numbers and their corresponding physical addresses, determining the packet's virtual destination address is simply a matter of looking up the physical address in the table:

```
if (pkt -> destination . network_number == LOCAL_NETWORK)
{
     /* Transmission to local computer */
     dst_addr = map(pkt -> destination . host_number);
     send(pkt, dst_addr);
}
```

In the above code fragment, `map()` takes a host number and returns the corresponding (physical) destination address on the local network. The packet can then be transmitted to the destination using the `send()` function.

For example, a computer with a 32-bit physical address `0x3F401E24` on network number `0x06` could be assigned host number `0x173`. Assuming a 16-bit virtual address (6 bits for the network number and 10 bits for the host number), the computer's virtual address would be `0x1973`. The virtual address is independent of the physical address (and vice versa).

In this example, if the value of `LOCAL_NETWORK` is `0x06`, and the host number is `0x173`, then the `map()` function will return the physical address `0x3F401E24`.

14.7.2 Transmissions to a Remote Network

A transmission to a computer on a remote network begins with the local transmission software receiving a packet with a virtual destination address that has a network number specifying a network other than the local one. The transmission software can detect this easily by inspecting the network number:

```
if (pkt -> destination . network_number == LOCAL_NETWORK)
{
```

```
    /* Transmission to local computer */
}
else
{
    /* Transmission to remote computer */
}
```

When transmitting on the local network, the transmission software called the `map()` function to determine the address of the local computer. The same cannot take place when dealing with an address on a remote network, for a variety of reasons, most notably the fact that the physical address of the remote host has no meaning on the local network.

A transmission to a remote network involves forwarding the packet to a local router. The local router then retransmits the packet to the next router leading to the destination network, and so on until the destination network is reached. If routers are to be involved in the transmission of the packet, the following changes are required to the local transmission software and the packet structure:

- The local transmission software must map the destination network number into the physcial address of a *local router*. Only then can the packet be forwarded to the local router.

  The local mapping of network numbers into router physical addresses means that each computer must maintain (or have access to) a table consisting of network numbers and local router addresses:

  ```
  if (pkt -> destination . network_number == LOCAL_NETWORK)
  {
      /* Transmission to local computer */
  }
  else
  {
      /* Transmission to remote computer via local router */
      router_addr = net_map(pkt -> destination . network_number);
      send(pkt, router_addr);
  }
  ```

  The above example is essentially the same as that described earlier for local transmissions, with the exception that the call to `map()` is replaced by one to `net_map()`. `Net_map()` takes a network number and returns the address of a local router that leads to the destination; the packet is sent to the local router.

- When a packet arrives at a router, the router must determine the next stage of the packet's journey across the interconnected network. In order to do this, the router must have access to the virtual destination address.

  The router inspects the network number of the virtual destination address and compares it with the contents of the router's routing table. Assuming that the network number exists in the routing table, there are two possible outcomes of this inspection:

1. the router is not connected to the destination network, meaning that the routing table should return the address of the 'next' router that leads to the destination network; both the router and the next router share the same network, as shown in Figure 14.7. The packet is forwarded to the next router, across the intervening network, using whatever protocol the underlying network supports.

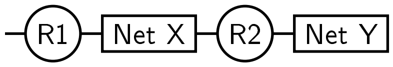


Figure 14.7: Packets for remote networks must traverse intermediate networks

2. the router is connected to the destination network, meaning that the routing table should return an indication that the destination network has been reached. When this occurs, the router must map the host number of the virtual destination address into the physical address of the destination computer; the packet can then be forwarded to the destination computer (see Figure 14.8). The packet is transmitted on the destination network using the protocols associated with the network.

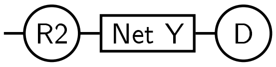


Figure 14.8: The 'last' router and the destination computer share the same network

When the packet arrives at the next router, the above two steps are repeated until the destination network is reached.

The steps involved in routing a packet that has been received by a router can be summarized as follows:

```
if (pkt -> destination . network_number == LOCAL_NETWORK)
{
     /* Destination network reached -- transmit to local computer */
     dst_addr = map(pkt -> destination . host_number);
     send(pkt, dst_addr);
}
else
{
     /* Destination not reached -- determine next router */
     router_addr = net_map(pkt -> destination . network_number);
     send(pkt, router_addr);
}
```

14.8 Packet Structure (Revisited)

From the discussion in the previous sections, it is apparent that if an interconnected network is to succeed, it must support its own packet structure, distinct from that of the underlying network. At a minimum, the internetwork packet should contain a length indicator and the virtual destination address.

Having said that, there are similarities between the interconnected network's packet structure and the packet structure of any underlying network: both packet structures contain a data field and a destination address. In addition to the destination address, strong arguments can be made to have the interconnected network include a source address; by supplying the virtual source address, the destination 'knows' where to send a reply. A possible format of the interconnected network's 'virtual packet' is shown in Figure 14.9.

Figure 14.9: A possible virtual packet for an interconnected network

The virtual packet contains four fields: VDA, the virtual destination address, VSA, the virtual source address, Length, the length of the packet, and Data, the data to be transmitted to the destination. The VDA is mapped into either the destination's physical address or the address of an intermediate router; it remains unchanged as the packet traverses the network.

14.9 Fault Tolerance and Reliability

Routers, like any other computer, can fail because of hardware or software faults. When a router failure occurs, the network should continue functioning, with as little impact upon the network users as possible. The addition of extra routers in the network helps overcome this problem, while at the same time leading to other problems, as discussed below.

14.9.1 Router Redundancy

Figure 14.3 illustrates a simple interconnected network that will split into two disjoint parts should any router or network fail. Figure 14.10 shows the effect of router R2 failing, resulting in two smaller networks: one consisting of networks A and B, the other C, D, and E.

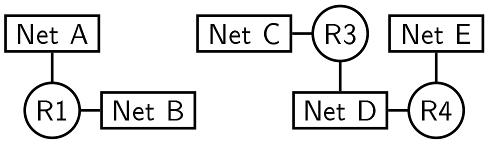


Figure 14.10: Router or network failure can result in network partition

The probability of network partition can be reduced by adding more routers between the various networks. For example, adding a router between networks B and D means that two router between networks B, C, and D must fail before the network is partitioned (see Figure 14.11).

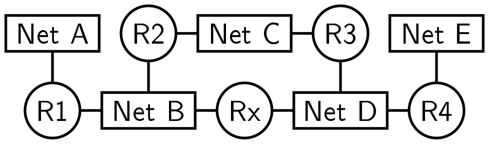


Figure 14.11: Additional routers make the internetwork more fault tolerant

14.9.2 Maintaining Routing Tables

As discussed in Chapter 13, there are a number of different ways in which routing tables can be maintained.

As well as being more fault tolerant, adding more routers to the interconnected network results in more paths through the network. However, with these benefits comes a problem – if multiple paths exist, which path should be taken to reach the destination?

14.9.3 Determining Physical Addresses

When a packet arrives at its destination network, the virtual address must be mapped into the corresponding physical address. In certain circumstances, the local host table may not include the physical address that corresponds to the virtual address. If this occurs, the transmitting computer (either a router or a local computer on the network) must determine which computer is associated with the virtual address. This can be achieved by transmitting a broadcast message to all computers on the network, essentially asking "who has virtual address '`X`'?" The computer with this virtual address is expected to respond to the transmitting computer with its physical address. With this, the transmitting computer can update its local host address table and forward the packet.

The above algorithm, known on the Internet as the **address resolution protocol** or **ARP**, requires that each computer know its own virtual address. An alternative approach, if programmable NICs are available, is to assign each computer on the network a physical address; all addresses are the same with the exception of a number of bits reserved for the host number. Whenever a packet is to be transmitted on the network, the host number is extracted from the virtual destination address and inserted into the reserved part of the physical address. In these implementations, there is no need for a routing table, nor is there a need for ARP.

14.9.4 Determining Virtual Addresses

The 'reverse' of asking for a computer's physical address is for a computer to ask for its virtual address. This **reverse address resolution protocol** or **RARP** was originally intended for 'diskless' computers that would boot, and during the boot process would ask for their virtual address. RARP, like ARP, is sent in a broadcast packet to all other computers on the network; the computer (or computers) that maintain the list of virtual addresses would respond with the virtual address associated with the computer.

RARP allows virtual addresses to be assigned dynamically. For example, if a network has more computers than virtual addresses, the addresses can be assigned on a demand basis. That is, when a computer requires an internetwork communication (to a local or remote network), it would send an RARP packet to a 'name' server. If an address was available, the server would respond with the virtual address. Once the communication had completed, the virtual address would be freed and made available to other computers on the local network.

14.10 The Internet

The basic Internet, as developed for the U.S. military, exhibits most of the characteristics of the internetwork developed in this Chapter. One of the major requirements of the military Internet was survivability; the failure of a router or a network was to be tolerated and the network was to continue functioning, albeit in a less than optimal mode.

The principal difference between the Internet and the internetwork developed in this Chapter is that the present generation of Internet use a 32-bit address. The 32-bit address is divided into *three* fields rather than two; the fields are a network number, a host number, and a **network class**. The network class allows different sized networks to co-exist in the same internetwork environment (this limitation was discussed in section 14.4.1).

14.10.1 Address Structure

The current version of the Internet (IPv4) supports *five* different classes of Internet address, all in the form class, net-id, host-id. The number of networks associated with each class are shown in Table 14.3.

Table 14.3: Internet address classes

Class	Number of hosts
A	more than 65,535
B	256 to 65,535
C	less than 256
D	multicast
E	reserved for research purposes

Figure 14.12 shows how a gateway can distinguish between the different address structures by examining the high order bits of the different IP address classes.

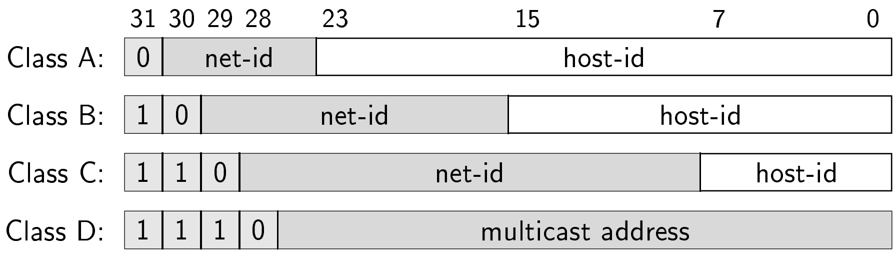


Figure 14.12: IP Address structures

14.10.2 Dot Notation

IP addresses are normally written in **dot notation**: each byte making up the 32-bit address is separated by a period, in the format m.n.o.p. Using the IP dot notation for address representation, class A networks have values 1.0.0.0 through 127.0.0.0 (the ARPANET is 10.0.0.0); while class B networks are in the range 128.0.0.0 through 191.255.0.0; and class C networks, 192.0.0.0 through 255.255.255.0.

14.10.3 Routing

The Internet is huge, consisting of literally hundreds of thousands of different sites and millions of hosts.

When an application has a message to send to an entity on a remote host, the message and destination IP address are supplied to the IP layer as usual. The packet is encapsulated in a datagram, but since the network identifier indicates a network other than the local network, the datagram is sent to a gateway on the network. The destination IP address is the address of the destination network/host, *not* an intermediate gateway.

When the gateway receives the datagram, the network identifier is extracted from the packet's destination IP address and used to determine the route of the datagram. If the network identifier indicates a network to which the gateway is directly connected, the datagram is forwarded to the destination on that network. However, if the gateway cannot reach the destination network directly, the gateway's routing table is accessed. Each entry in the routing table indicates the address of the *next* gateway on the path to the destination network. Should the specified destination network not exist in the routing table, the datagram is forwarded to a default gateway. This cycle is repeated until the datagram reaches a gateway which is connected to the destination network. Note that the source and destination IP addresses are never altered by the gateways; the IP addresses are used for routing purposes.

Each IP gateway is responsible for maintaining its own routing table. The routing table can be initialized from static storage or the gateway can communicate with other gateways, dynamically building the routing table. Only a subset of gateways maintain a list of routes to all networks, these gateways are known as **core** gateways. The remaining gateways are known as **non-core** gateways. Routing tables are updated by the gateways exchanging routing information consisting of the distances between gateways in terms of hop-counts.

14.10.4 The Future

As previously mentioned, the Internet was designed as a reliable communication system for the U.S. military. The number of computers to be interconnected, although large, was never envisaged to reach the numbers that connect to the Internet today. As a result, the number of Internet addresses is running out.

In the early-1990s, the Internet Engineering Task Force (IETF) embarked upon a redesign of the Internet address structure. In 1998 they unveiled a new, 128-bit address structure as well as a new, 64kb packet structure. The 128-bit address structure is hierarchical, and includes a class field.

14.11 Summary

This Chapter has shown how an **internet** can be built atop a collection of disjoint, independent, physical networks. At the physical level, an internet consists of **routers** connecting the physical network, creating an interconnected **mesh** of networks. Since the various physical networks may have different topologies, network addressing schemes, and protocols, it is necessary to design a virtual address structure that can uniquely identify any computer on any network.

The virtual address structure adopted for most internets is hierarchical, typically consisting of two fields: a network number (a unique number assigned to each physical network) and a host number. Each computer in an internet is assigned a unique virtual address; the network number is common to all computers in a single physical network.

All applications that require the use of the internet refer to the virtual address rather than the network's physical addresses. Before a packet is transmitted on the network, the virtual address is **mapped** into the corresponding physical address *or* the address of a router that leads to the intended destination computer. Each computer on a network maintains two tables, one that maps local host numbers to the corresponding (local) physical network address; the other that maps network numbers into the physical network address of a router that leads to the intended destination computer.

A **router** is a special computer that receives packets destined for computers on other networks. The router always inspects the packet's destination network number: if the router is connected to the destination network, it maps the host number into the destination's physical address and transmits the packet to the destination computer. However, if the router is not attached to the destination network, the router must map the network number into the address of the next router on the path to the intended destination. Routers also maintain two tables: a table of host numbers that correspond to physical addresses associated with each network to which it is connected; and a table of network numbers with the corresponding physical addresses of routers.

The Internet, as originally developed for the U.S. military, exhibits all the characteristics discussed in this Chapter. The need for robustness in the original Internet requires routers to be exchanging routing table information at periodic intervals. Should a router fail, another path through the network can be generated quickly.

It can be argued that many of today's Internet Service Providers (ISPs) function like wide area networks rather than members of a co-operating Internet. Some ISPs operate their own cross-country networks with few links to other ISPs, as a means to 'starve out' smaller, regional ISPs. (This can backfire, should their link to a region fail, cutting off their customers from the 'outside' world.)

14.12 Exercises

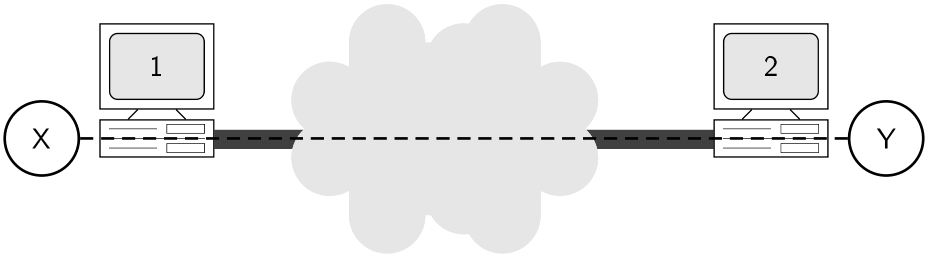
1. Explain why allocated a 16-bit address structure permits no more than 65,536 computers on an entire internetwork.

2. If a 16-bit address structure is chosen and there are a total of 65,536 computers on the network, explain why multicast and broadcast cannot be supported.

3. Assuming that the Ethernet approach of allocating one bit to indicate multicast and broadcast is adopted, what is the maximum number of computers allowed on the internetwork with a 16-bit address structure?

Part 7

Interprocess communication between different computers



The availability of a path between two computers on the same or separate networks does not ensure that two processes on these computers can communicate. For example, consider the following:

- The two computers may be connected to different network architectures, meaning that the addressing scheme used on one network may be completely different from the one used on the other.
- The two computers may have processors made by different manufacturers with incompatible architectures. A common problem in these situations is that one architecture supports big-endian byte ordering, while the other supports little-endian.
- The two computers may have the same hardware architecture, but may support different operating systems. If process identifier structures differ between the two operating systems, it may not be possible to use process identifiers in the communication.
- Using a process identifier to identify a destination process has its limitations, since in many operating systems, process identifiers are assigned randomly when a process is created. As a result, it may not be possible to use the same process identifier in subsequent communications.
- Even if all of the above issues can be overcome, there is still the problem of knowing the location of the destination process – without a destination address, communications may not be possible.

- Finally, requiring users to remember 32-bit process identifiers and 48-bit Ethernet addresses is not conducive to popularizing the technology.

Part 7 examines ways in which these problems have been overcome by network designers and software engineers through a technique known as **layering**. Several layered architectures are presented in Chapter 15, with a detailed discussion of Berkeley sockets.

Chapter 15

Layered Architectures

15.1 Introduction

With few exceptions, a number of assumptions have been made with respect to how communications are supported between computers, notably:

- how a network packet or frame is constructed.

  For the most part, users have been expected to build the network packet, identify the destination computer. Although this is useful in teaching the functioning of networks, it is unrealistic to expect the everyday user (other than perhaps someone working on a research project) to perform such tasks. Imagine the user's enthusiasm if every time a message on an 802.5 network is to be sent, the user is required to type the destination's 48-bit address explicitly.

- the destination is assumed to be a computer.

  Many operating systems support multiple processes on a single computer, meaning that a message arriving at a computer has little meaning unless a destination process can be explicitly identified.

This Chapter considers how communication systems overcome the problems of identifying entities and hiding network-specific features through the use of layered software, producing what is commonly known as a **layered architecture**.

15.2 Background

15.2.1 Identifying Remote Entities

Operating systems that support multiple processes require mechanisms whereby individual processes can be identified. For example, message-based operating systems, each process is associated with a unique identifier and at least one message queue. A source process sends a message to a destination process by supplying the message and the identifier of the destination process to the queue management software using the `send()` primitive. Similarly, when a process is to receive a message, the `recv()` primitive is invoked, supplying the queue management software with the process's identifier. The queue associated with the identifier is then accessed and the first available message on the queue is returned to the process.

This model can be extended to the transmission of messages between hosts. That is, a frame containing the process identifier of the destination process could be sent to the destination host. The destination address then consists of two parts: the destination host and the destination process. Upon receipt of the frame, the reception software on the destination host would perform a `send()` to the destination process specified in the destination address. Overall, the communication software functions like a multiplexer: messages to various destinations are multiplexed into separate frames and transmitted on the network, while frames are taken from the network and demultiplexed into messages using the process identifier. A typical sequence is shown in Figure 15.1.

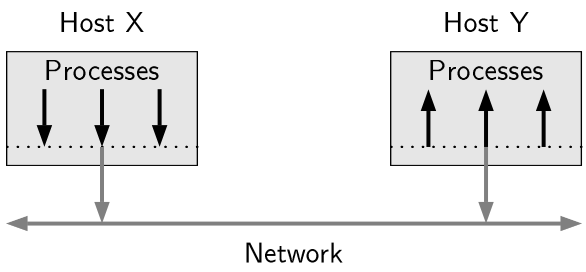


Figure 15.1: Communication software multiplexes and demultiplexes messages

Using process identifiers to identify remote processes will work successfully as long as:

- There is a one-to-one mapping between a process and its process identifier.

  In many operating systems supporting multiple processes, a pool of identifiers are recycled amongst the existing processes. Over a period of time, an identifier may be associated with a number of different processes; as processes terminate, the identifier is freed to be used by a newly created process. Should process identifiers be assigned dynamically, there is no guarantee that the destination process will be associated with the 'correct' identifier.

- All identifiers are the same format.

  Different operating systems often use different identifier formats and sizes. This can lead to problems should a network of heterogenous machines be established. For example, if a 16-bit identifier size is chosen and an operating system that uses 32-bit identifiers is attached to the network, how are 32-bit identifiers to be represented? Similarly, if a 32-bit identifier is taken as the standard, but all identifiers are 16-bits long, space in the frame will be wasted whenever an identifier is transmitted.

From the two situations described above, it should be apparent that process identifiers are not necessarily the best method of identifying a process on a remote host. Ideally, a common process identifier structure should be agreed upon by all operating systems; however, since this is an unlikely prospect, many communication systems have a second set of identifiers that are used for communications. These identifiers are commonly known as **ports** and have a structure that is agreed upon by all operating systems on the network. A process is associated with its operating system-specific process identifier and is *bound* to as many ports as necessary for the process to perform its communication functions; those processes not involved in network communications are not bound to a port.

For example, in the network shown in Figure 15.2, processes Py and Pz are bound to ports 2 and 3 respectively on host E; while on host S, processes P4, P9, and P1 are bound to ports 1, 2, and 4 respectively.

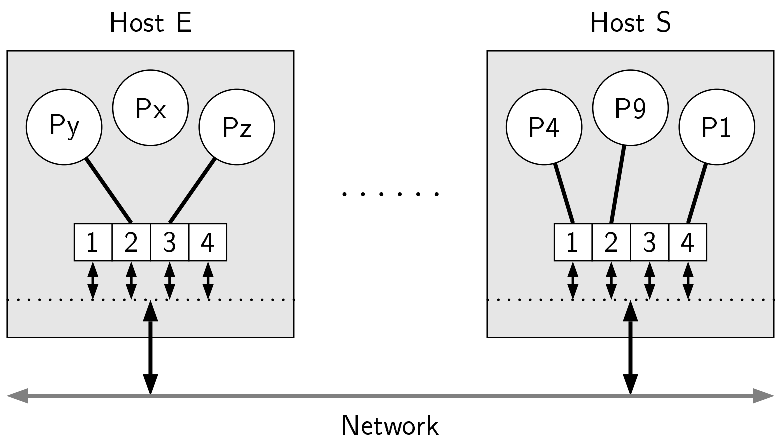


Figure 15.2: Processes are identified by their ports

The host address and port number are often referred to as a single data structure known as a **socket**; internally, the socket consists of a host-port pair.[1] Host uniquely identifies the process's host, while port identifies the process's port (the **host-port pair** is normally written in angle brackets as <host,port>). Should the process number change (for example, the host may crash and reboot, resulting in the software module obtaining a different process number), the system will still function as long as the software module gets the original socket.

Before a process can communicate, it must first bind to a socket. Thereafter, all messages that arrive on the host with the process's port number are returned to the process. When transmitting a message, a process supplies the communication system with the message and the socket of the destination process. The source communication system uses the host identifier to determine the destination host, while the destination communication system uses the port identifier to determine the destination process.

For example, using the above diagram, if process Py on Host E is to send a message to process P1 bound to port 4 on host S, the message is sent to socket address <S,4>.

Finally, it must be noted that sockets are not the only method of transmitting messages without using explicit process identifiers. For example, a **mailbox** is an intermediate entity that processes

[1]Sockets were originally developed at UC Berkeley for UNIX 4.2 as the standard software interface to the Internet.

can write to and read from. If a process A is to send a message to process B, the message is written to process B's mailbox. When process B reads its mailbox, the message from A is returned.

15.2.2 Layering

From the discussion in the previous section, it should be apparent that it is unrealistic to expect each process to support its own version of sockets and the communication software. Instead, most communication software has been developed to offer a range of communication services to processes and their related applications. Communication software is now typically written as a series of **layers**, each one offering services to the layer above it and using the services of the layer below it. There are many reasons for developing layered systems, including:

Enhancing features. Layers can add features or facilities to a network. For example, an unreliable, error-prone physical layer can be made reliable through the use of a data link layer supporting a Go-Back-N protocol.

Hiding features. The addition of a layer can also hide features. For example, it is possible to use names rather than network addresses when referring to a station; an application could supply a host name to an underlying layer which maps the name into a network address.

Ease of modification. Layered software is typically easier to maintain than monolithic software, since the functions performed by a layer are localized to the module associated with the layer in question. Fault detection and subsequent module retesting are also simplified. For example, module testing can be performed in a controlled manner if test software is written that generates the necessary error conditions. This is typically faster than waiting for the error to occur while the module is running.

Portability. Software written in a layered fashion is typically more portable than software that isn't. For example, if an application is written for a specific network (say an 802.3 bus) then transferring the application to another network may well require extensive modifications. By writing application software in a *network independent* fashion, moving the application between networks can be a simple task.

Layering has been used implicitly throughout the text; for example, wide area networks were discussed in terms of four distinct layers:

Physical. The physical layer is responsible for the transmission and reception of bytes across the channel.

Data Link. The data link layer defines the channel packet structure and is responsible for the orderly flow of information between the interconnected DTEs.

Network. The network layer is responsible for the establishment, maintenance, and eventual clearing of virtual circuits.

Application. The application layer in the wide area network example is a network testing tool that permits the user to establish a number of virtual circuits with various DTEs.

15.3 Terminology

Each host attached to the underlying subnet is referred to as a **system**. Within a single subnet, all systems typically support the same layers; the layers themselves are said to form a **layered architecture**. The highest layer of a layered architecture consists of applications and processes using the communication services offered by the underlying layers. The lowest layer deals with the physical connection of the systems making up the subnet.

Although the different layers in a layered architecture perform different functions, there are three underlying similarities:

- each layer is associated with its own internal protocol. For example, in a wide area network, the data link layer could support a Go-Back-N protocol, while the network layer may have a virtual circuit protocol such as X.25.
- each layer offers services to the layer above it. As an example, the data link layer of most commercial wide area networks offer reliable point-to-point communications to the network layer.
- each layer uses the services of the layer below it. For example, wide area network applications use the virtual circuit service offered by the underlying network layer.

Because of the widespread use of layering, many layered architectures use a more formal description of each layer and its interactions with its adjacent layers. For example, any layer N offers one or more **N-service**s to the layer above it, layer $N+1$. Similarly, layer N uses the services offered by the layer below it, layer $N-1$. Layer $N+1$ is referred to as the **service user** and layer N, the **service provider**.

Each layer N supports software (or hardware) modules known as a **protocol entity** that conforms to the **N-protocol** associated with the layer in question. Protocol entities that exist at the same layer are known as **peer protocol entities**. Peer protocol entities exchange information in **N-protocol data unit**s or **N-PDU**s, using the services of the $N-1$-layer. The service user that receives information is referred to as the **correspondent user**. This is represented diagrammatically in Figure 15.3.

These definitions are recursive; that is, any layer N becomes layer $N+1$ the further into the architecture one proceeds. At the lowest layer there is no layer $N-1$; the lowest layer must support the physical interconnection of the systems. Similarly, at the highest layer, there is no layer $N+1$ other than end users or application processes.

Users at layer $N+1$ access N-services through **service access point**s or **SAP**s. Since at any layer $N+1$ there are potentially multiple users of a particular N-service, each service user is assigned a unique SAP address. The boundary between the two layers is known as an **interface** and is crossed through an SAP.

The N-layer PDU also contains N-layer peer protocol entity control information (such as sequence numbers and checksums) in the **N-PCI** (or **N-protocol control information**) as well as the protocol data unit from layer $N+1$. At layer N, the protocol data unit from layer $N+1$ is referred to as the **N-SDU** (or **N-service data unit**). The N-PDU is said to **encapsulate** the (N+1)-PDU. The relationship between the $N+1$ and N layers is represented in Figure 15.4.

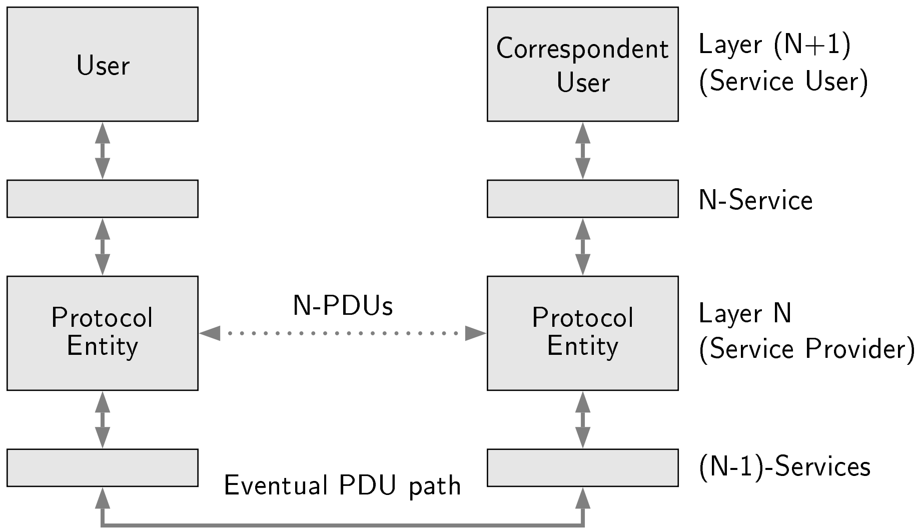


Figure 15.3: Terminology of a layered architecture

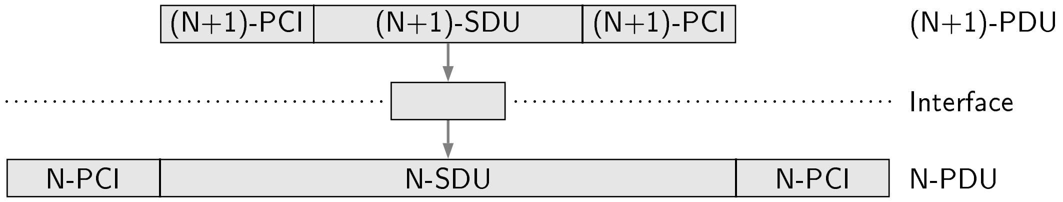


Figure 15.4: Encapsulation of the N-PDU

The services offered by any layer fall into two categories: confirmed and unconfirmed (corresponding to the connection-oriented and connectionless services discussed in previous Chapters). Support for the transfer of information is described by four primitives (the first two are used by both confirmed and unconfirmed services, while the last two are for confirmed services only):

Request. A service user initiates a transfer to a correspondent user by passing a request primitive to the service provider. The service provider takes this information (an SDU) and encapsulates it into a PDU. The PDU is sent to the correspondent service provider on the remote system using the services provided by the underlying layers.

Indication. Upon reception of the PDU, the correspondent service provider supplies the correspondent user with an indication primitive.

Response. The correspondent user is expected to acknowledge the reception of the indication with a response primitive. The correspondent service provider returns the response to the originating system in a PDU.

Confirm. When the original service provider receives the PDU, a confirm primitive is returned to the user.

The information flow between the $(N+1)$-layer users and the N-layer service is shown in Figure 15.5 (the response-confirm primitives are used only in confirmed services).

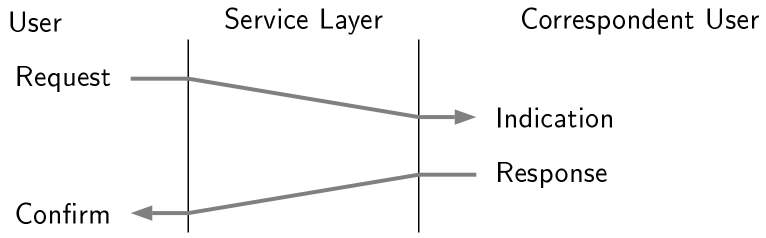


Figure 15.5: Information flow between a User and a Correspondent User

Many layered architectures have adopted the notation layer.service.primitive when referring to the services required by a user at a particular layer. Typical services include CONNECT (for establishing a connection), DATA (for the transfer of data), DISCONNECT or CLEAR (for clearing a connection), and RESET (for resetting a connection). For example, when establishing a virtual circuit in a wide area network, the application issues a CONNECT request, written as A.CONNECT.request and includes the address of the destination application.

15.4 Examples of Layered Architectures

Layered architectures are by no means a new phenomenon in data communications. For example, both IBM's System Network Architecture (SNA) and Digital's Digital Network Architecture (DNA) were released in the mid 1970s. There are several other well known architectures, three of which are examined in this section: the Open Systems Interconnection model, IEEE 802, and the Internet Protocol Architecture.

15.4.1 The Open Systems Interconnection Reference Model

Proprietary products such as SNA restrict users to a specific manufacturer's line of equipment. When SNA was introduced, both customers and manufacturers alike recognized this problem, and in the late 1970s began pushing for an 'open' architecture for which all manufacturers could develop products. The objective of such an architecture was to allow customers with equipment from different manufacturers to communicate on the same subnet, given adherence to a set of agreed-to standards.

Probably the best known example of an open architecture is the **Open Systems Interconnection** (or **OSI**) reference model proposed in the late 1970s by the International Organization for Standardization (or ISO). The OSI model consists of seven layers, each of which supports one or more services. Although the OSI model is a standard, it is not an implementation standard. That is, the model explains what each of the seven layers should do; however, the services and protocols associated with each layer are not specified in the model. (Section 15.3 described layering using OSI terminology.)

The main objective of the OSI model is to support communications between end-users on different systems. The end-users are not part of the OSI model, they simply use the facilities offered by the model. The four upper-most layers (**application**, **presentation**, **session**, and **transport**) support end-to-end protocols that are network independent. The lowest three layers (**network**, **data link**, and **physical**) support protocols that are network dependent.

The diagram shown in Figure 15.6 illustrates the logical and physical flow of information between a pair of end-users on different systems. Messages exchanged between end-users flow vertically through the system until they reach the Physical Layer, at which point the message is transmitted between systems (xPDU denotes the PDU used by a specific layer; Packets, Frames, and Bits normally are not referred to in terms of PDUs).

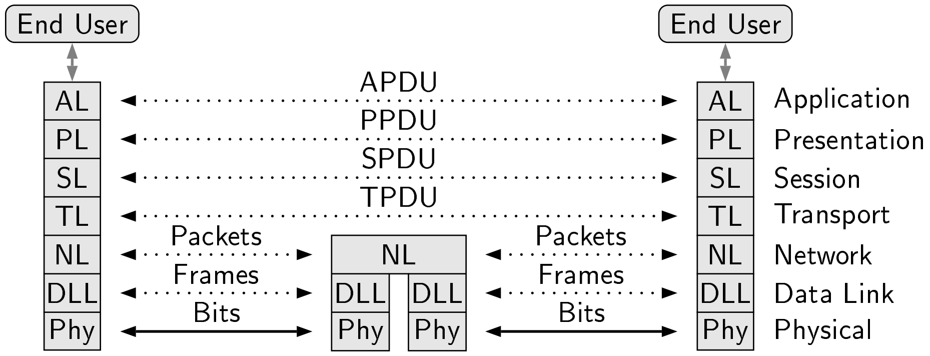


Figure 15.6: The Open Systems Interconnection Reference Model

Network Dependent Layers

The lowest three layers of the OSI model (physical, data link, and network) are network dependent, defining protocols for the transfer of information between systems.

The **physical layer** is the lowest layer in the OSI model (layer 1), responsible for the transmission of bits over a physical channel; covering issues such as signalling, modulation, as well as defining how components are physically interconnected. Examples of standards that are intended for the physical layer include RS-232-C, RS-449, and X.21. Physical layer standards for local area networks include IEEE 802.3 (ISO 8802.3), IEEE 802.4 (ISO 8802.4), and IEEE 802.5 (ISO 8802.5). Newer physical layer standards include those for ISDN networks, such as I430/1.

The second layer in the OSI model is the **data link layer**. This layer is responsible for the transfer of information in the form of frames across the underlying physical layer. The data link layer can be reliable, overcoming physical layer errors using Go-Back-N protocols. Examples of data link layer protocols include HDLC, SDLC, ADCCP, and LAP-B. In some networks, the data link layer is unreliable and each frame is treated as a datagram. Examples of unreliable data link layer protocols are the local area network protocols 802.3 and 802.5.

The **network layer** is the third layer of the OSI model and handles the routing of packets across the underlying subnet. In connection-oriented networks, the network layer is responsible for call establishment, data transfer, and call clearing. ITU X.25 is an example of a network layer protocol.

ISO standards include ISO 8473 and 8348.

Network Independent Layers

The lower three layers of the OSI model describe point-to-point communications between pairs of nodes on a subnet. The remaining layers assume that the underlying layers can carry information to a given remote system.

The fourth layer, the **transport layer**, defines how end-to-end communications can be established and maintained across the subnet. One of the functions of the transport layer is to make the network transparent to the remaining upper layers of the OSI model. OSI defines five different classes of connection-oriented transport service. Each class offers a specific quality of service based upon the underlying network layer's quality of service:

Simple class (Protocol class 0). The simple class is intended to operate over networks with low failure and error rates. A class 0 transport layer service therefore does not enhance the service offered by the network layer, leaving sequencing and flow control to the network layer. In class 0, there is one network connection for each transport connection; should the network connection fail, the transport connection will also fail.

Basic error recovery class (Protocol class 1). Protocol class 1 is similar to class 0, with the exception that it is intended for networks that are subject to network resets. For reset recovery, it is necessary for the transport service to maintain TPDU (Transport Protocol Data Unit) sequence numbering.

Multiplexing class (Protocol class 2). A class 2 transport service allows the multiplexing of one or more transport connections over a single network connection. The underlying network is assumed to be fully reliable.

Error recovery class (Protocol class 3). Protocol class 3 supports the features found in classes 1 and 2; notably, that it can recover from network resets and also supports multiplexing.

Error detection and recovery class (Protocol class 4). A class 4 network is intended to operate atop a datagram network; meaning that extensive error detection and handling features are necessary (i.e., sequence numbering, CRC checking, timeouts, and TPDU retransmission).

Transport layer protocols include ISO 8072 (OSI transport service), ISO 8073 (OSI transport protocols), and ITU X.214 and X.224.

The fifth layer of the OSI model is the **session layer**. Unlike the lower layers that are concerned with data transportation issues, the session layer deals with the management of interactions (or dialogs) between the two end-users. Each session connection has a corresponding transport connection. Should the transport connection fail, the session layer can re-establish the connection transparently (i.e., without end-user intervention). Similarly, once an end-user has completed a session, the session layer may choose to keep the transport connection active, initiating another session across the same transport connection.

The session layer may offer a number of different data transfer services. The session layer supports both full and half-duplex user dialogs. In half-duplex, the transmitting user is in possession of a permission-to-transmit token that can be exchanged by the users. Other data transfer services include **quarantining** and **synchronization**. Quarantining allows the local session layer to withhold delivery of a number of messages until explicitly instructed to do so by the remote session layer. Synchronization is intended to allow a data transfer to 'rewind' to a checkpoint specified by end-user application. Both the quarantining and synchronization services are useful in transaction-oriented applications where the loss or retransmission of information may lead to inconsistencies in, for example, a database.

In a truly open system, there is no guarantee that the equipment available to one end-user will be the same as that available to another. If the two end-users are to communicate, it may be necessary to convert the information format used on one system to that used by another. For example, one system may use ASCII and the other EBCDIC; similarly, the internal representation of integer or floating point numbers may differ, requiring some form of conversion. Layer 6 of the OSI model, the **presentation layer**, deals with the representation and transformation of the data between application layer entities.

In its simplest form, each system's presentation layer could support a translator for every other possible system on the network. If there are N different systems on the network, a total of $N \times (N-1)$ translators would be needed. However, if a common network-wide information representation is chosen, then only $2 \times N$ translators are needed: one from the machine's internal representation to the network external representation, and the other, from the external representation to the internal representation.

Presentation layer protocols include ITU X.409, Presentation Transfer Syntax, and GM's MAP Standard Message Format. In GM's presentation layer protocol, internal data is converted into a structure consisting of a type identifier (for example, Boolean, Integer, or Floating Point), an optional length indicator (depending upon the type), and the converted value. For example, Booleans are a single byte (`0x00` for true and `0xFF` for false), while a floating point number is converted into an ASCII string.

The presentation layer can also be responsible for the encryption and decryption of information, as well as for file compression.

The **application layer** is the uppermost layer of the OSI reference model, offering application services to the different end-users. Some of the more common application services include file transfer protocols, electronic mail (such as X.400), and virtual terminals (such as X.28 and X.29).

To avoid the duplication of work at the application layer, three types of application elements are proposed by the OSI model for use by end-user applications:

- Elements within the end-user application that deals with accessing OSI services (**User Elements** or **UE**).

- Elements that have capabilities that can be useful to a variety of applications (**Common Application Service Elements** or **CASE**).

- Elements that have capabilities for specified applications (**Specific Application Service Elements** or **SASE**).

15.4.2 IEEE 802

The lower three layers of the Open Systems Interconnection model are intended for connection-oriented wide area networks. These networks exhibit a number of common properties, including low speeds (typical maximums of about 56 kbps), high error rates (about 1 bit in 10^5), and the need to perform packet routing from node-to-node. Accordingly, the protocols proposed and developed for the lower levels of the OSI model overcome these limitations.

Since local area networks such as the Ethernet and the IBM Token Ring are faster (speeds of 10 Mbps and up), exhibit lower error rates (about 1 bit in 10^9), and do not require routing, many of the functions provided by the lower OSI layers are redundant or unnecessary in a LAN environment. In short, many of the protocols associated with the lower layers of the OSI model area considered to be too *heavy-weight* for local area networks.

The IEEE recognized these limitations and set about defining the 802 standard, intended for high-speed, low-error rate local area networks. 802 protocols are *light-weight* protocols. As with the OSI model, one of the objectives of the 802 standard is to ensure that equipment from different manufacturers can communicate if the standard is adhered to.

The 802 standard is a three-layer architecture that encompasses the physical and data link layers of the OSI model. Figure 15.7 shows the overall structure of the 802 protocol family and its relationship to the OSI model.

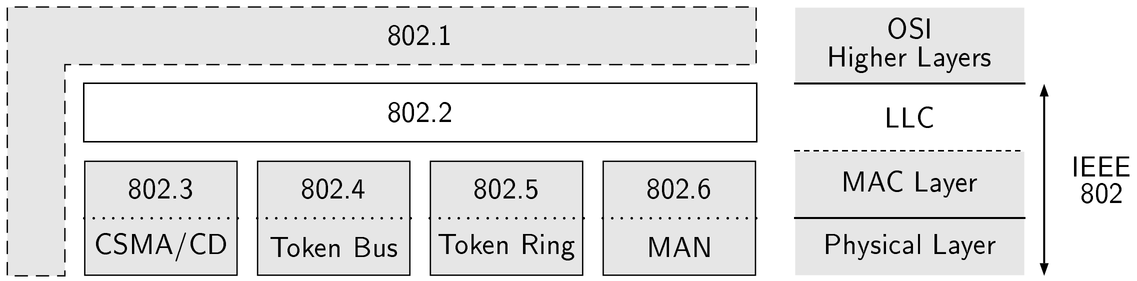


Figure 15.7: The three-layer 802 Protocol family

The IEEE-802 standard consists of the following parts:

802.1 Overview, internetworking, and systems management; defining the relationship between the various 802 protocols and the ISO higher-layer protocols.

802.2 The Logical Link Control (LLC) sublayer.

802.3 CSMA/CD bus access method (described in Chapter 5).

802.4 Token passing bus access method (described in Chapter 6).

802.5 Token passing ring access method (described in Chapter 6).

802.6 Metropolitan area network (MAN) access method (described in Chapter 6).

The various 802 physical layers have been described in Chapter 5 (802.3 CSMA/CD) and Chapter 6 (802.4 Token Bus, 802.5 Token Ring, and 802.6 MAN). Additional 802 physical layer protocols are currently under development, including 802.3u (100 Mb/s CSMA) and 802.12 (100 Mb/s Token Ring).

The OSI data link layer is divided into two parts: the MAC (or Media Access Control) layer and the LLC (or Logical Link Control) layer.

Media Access Control

The Media Access Control (MAC) sublayer provides three information exchange primitives to the LLC sublayer, regardless of the underlying network. The three service primitives are as follows (note that OSI terminology is used):

MA.DATA.request. The MA.DATA.request primitive takes three parameters from the LLC sublayer: the network address of the destination SAP (either a unicast, broadcast, or multicast address), the SDU to be transmitted (i.e., an LLC Protocol Data Unit) and a service class requesting a certain priority level. The SDU is transmitted by the MAC layer using the physical layer services of the underlying network.

The service class is used by networks that can support different levels of priority, such as the 802.5 token ring. The MAC sublayer transmits the frame with the specified priority. Networks such as 802.3 which do not support priorities simply ignore the service class.

MA.DATA.indication. The MA.DATA.indication returns the source and destination address as well as the SDU received to the correspondent LLC sublayer. The status of the reception is also returned, indicating whether the frame was received correctly or the reason for the failure of the reception.

MA.DATA.confirm. The status of the MA.DATA.request is returned by MA.DATA.confirm to the LLC user, verifying the completion of the transmission or specifying a reason for the failure.

The result returned by MA.DATA.confirm indicates only the success of the transmission at the MAC sublayer. In networks such as the 802.5 token ring, the confirmation is extracted from the status bits in the ring frame. However, in the 802.3 CSMA/CD network, the confirmation is generated locally; an example of an 802.3 transmission failure is when the collision limit is reached.

A time-sequence diagram of the primitives described above and the MAC and LLC sublayers is given in Figure 15.8.

Logical Link Control

The Logical Link Control (LLC) sublayer offers three types of service to the upper OSI layers: unacknowledged connectionless; connection-oriented; and acknowledged connectionless. The connectionless services are intended for applications that support their own error checking and recovery schemes.

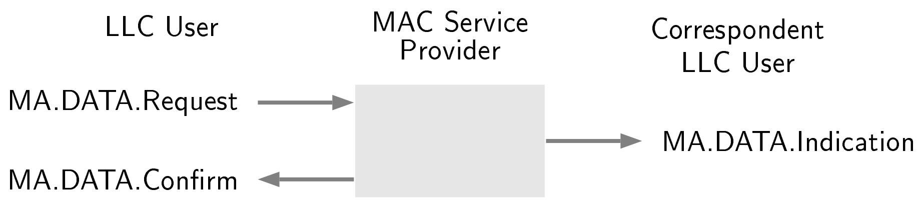


Figure 15.8: MAC Layer information interchange primitives

The unacknowledged connectionless service supports two service primitives: L.DATA.request, for the transfer of an SDU to a given remote address; and L.DATA.indication, signalling the receipt of an SDU. This service is a datagram service; there are no acknowledgements associated or supported with this service. The time sequence diagram for the unacknowledged connectionless service is shown in Figure 15.9.

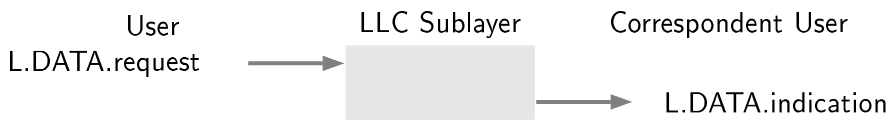


Figure 15.9: LLC Unacknowledged Connectionless Service

In the acknowledged connectionless service, an L.DATA_ACK.request contains an SDU that the delivery of which is acknowledged by the remote LLC sublayer, as shown in Figure 15.10.

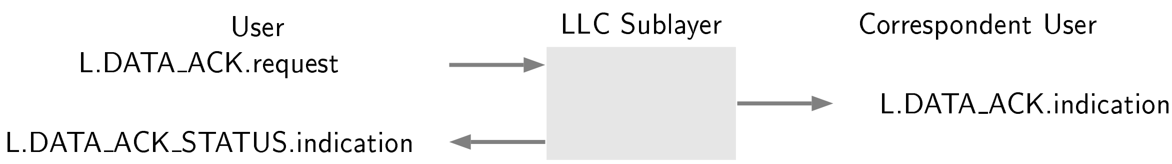


Figure 15.10: LLC Acknowledged Connectionless Service

The connection-oriented service shown in Figure 15.11 allows the service user to request the establishment of a connection using L.CONNECT.request (note that there is no L.CONNECT.response since the response is generated by the correspondent LLC sublayer).

Once the connection has been established, either service user may initiate a data transfer using L.DATA_CONNECT.request; see Figure 15.12.

The connection can be cleared by either user issuing an L.DISCONNECT.request; see Figure 15.13.

In all three types of service, the addresses supplied to the LLC SAP indicate the destination LLC SAP as well as the destination physical address.

Regardless of the type of service, there is a single LLC Protocol Data Unit (LLC PDU) format, as shown in Figure 15.14. The LLC PDU fields are described as follows:

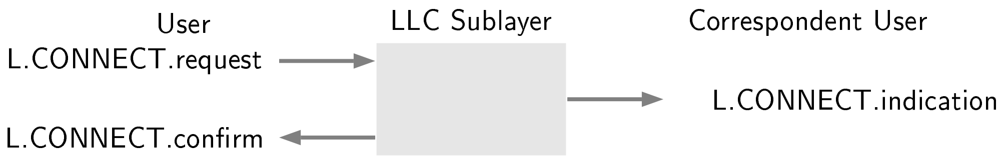


Figure 15.11: LLC Connection-Oriented Protocol: Connection Request

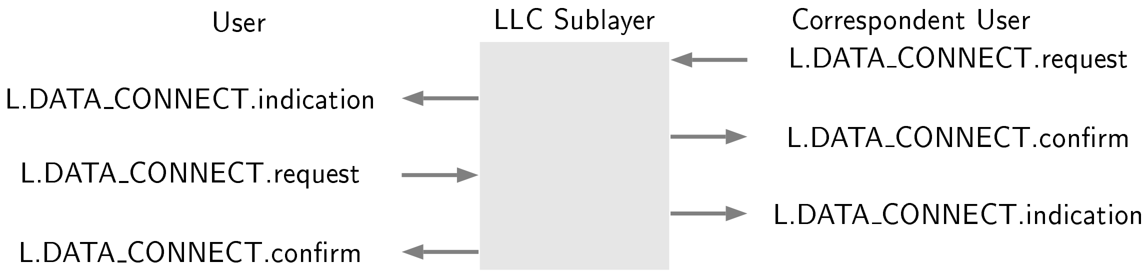


Figure 15.12: LLC Connection-Oriented Service: Data Transfer

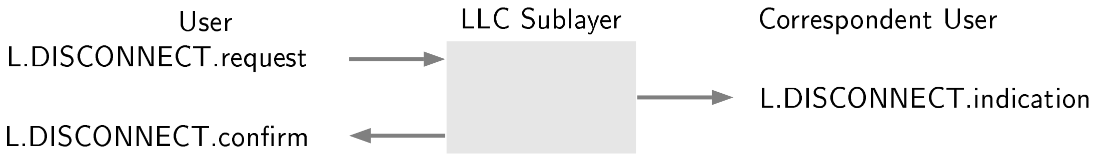


Figure 15.13: LLC Connection-Oriented Service: Call Clearing

DST SAP. The destination LLC service access point address. One bit indicates whether the SAP is an individual or group address, meaning that any one of 128 individual or 128 group LLC SAPs may be specified.

SRC SAP. The LLC service access point for the source of the Information.

Control. Each LLC PDU is sent with a control field; the format and contents of which is based upon the HDLC protocol control field (see Chapters 10 and 13). The connection-oriented service supports both Information and Supervisory control fields. On the other hand, the connectionless services use the Unnumbered Information (UI) control field.

Information. A variable length data field, M is a multiple of 8-bits. M can be zero; for example, when sending supervisory PDUs.

There is no LLC checksum, the MAC layer FCS is presumed to offer sufficient error detection capabilities.

DST SAP (1)	SRC SAP (1)	Control (1)	Information (M)

Figure 15.14: The LLC Protocol Data Unit (all sizes in bytes)

15.4.3 The Internet Protocol Architecture

The Internet Protocol Architecture (or simply, Internet) is an example of an open system architecture developed for a single (large) customer: the U.S. military. The Internet was originally developed in the mid-1970s as part of the U.S. military's DARPA (Defense Advanced Research Project Agency) study of internetwork communications. Earlier research funding from DARPA (or ARPA as it was known in the 1960s) went into the development of the ARPANET packet-switching network.

The Internet consists of three network independent layers, built atop the network access services of any underlying network. The layers and their relationship to the OSI model are shown in Figure 15.15.

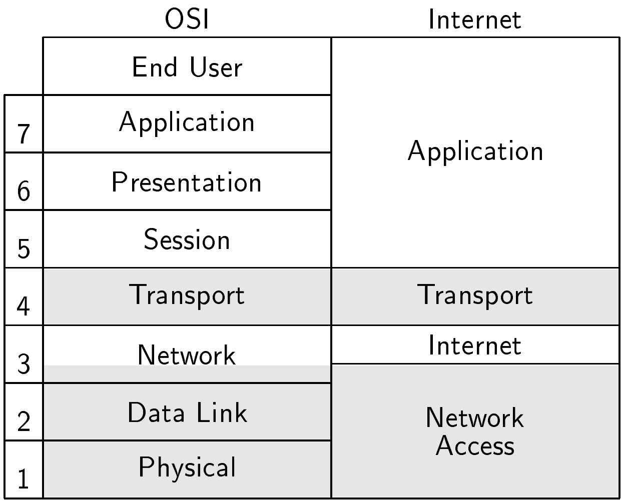


Figure 15.15: The relation between the OSI model and the Internet

The Internet Protocol Architecture is intended to operate atop almost any type of network, as long as the necessary software exists between the Internet and Network Access layers. The underlying network can be connection-oriented or connectionless, a wide area network or a local area network. The type of service offered by the underlying network is transparent to the top three layers.

In order to achieve this transparency, the designers of the Internet had to adopt several standards. For example, since byte ordering of an integer can vary from machine to machine (see Chapter 1), machines have been classified as either **little-endian** (in which the low-order byte is associated with the lowest memory location) or **big-endian** (where the low-order byte is stored in the highest memory location). The standard adopted by the Internet requires all machines to transmit information in big endian style (i.e., the most significant byte first).

Furthermore, the need for network independent host addresses was also recognized. Since different networks support different address structures, each host is assigned a unique 32-bit **Internet address**. When a packet is available for transmission, it is supplied with the destination host's Internet address in an **Internet datagram** to the Internet layer. The Internet layer maps the Internet address into a network-specific physical address; this address is then used by the Network Access layer as the destination address.

Should the Internet layer not recognize the Internet address, an **Address Resolution Protocol** (ARP) packet containing the unknown Internet address is broadcast on the network. The station with the address in question returns its physical address in an ARP reply packet.

The Internet layer offers a connectionless datagram service to the Transport layer (this service is known as **IP** or the **Internet Protocol (IP)**). The Internet datagram contains two parts: a header and a data area. The Internet layer can fragment datagrams if they are too large for the underlying network. The datagram header contains control information as well as the destination and source Internet addresses.

The Transport layer offers two levels of service to applications:

- a datagram protocol known as **UDP** or User Datagram Protocol. UDP messages are supplied to the Internet layer for transmission to a given Internet destination. Since a single host may have many processes transmitting and receiving UDP messages, each host has a number of UDP ports to which UDP messages are addressed.

  Upon receipt of a UDP message from the underlying network, the Internet layer supplies the message to the UDP service, which makes the message available to the process associated with the port specified in the UDP message.

  UDP is a 'best-effort' datagram (i.e., **connectionless**) service, meaning that a UDP packet can be lost, duplicated, or arrive out-of-order. Applications built atop UDP must implement their own reliable protocol (if reliability is required).

- a reliable, full-duplex connection-oriented, stream protocol known as **TCP** or Transmission Control Protocol. A message sent by TCP is sent in a **segment**, consisting of a TCP header and a data part. The header contains the source and destination ports as well as a 32-bit sequence number and a 32-bit acknowledgement number. TCP is a sliding window protocol, the sequence and acknowledgement numbers specify the current byte number and the number of the next expected byte, respectively. The window size is expressed in bytes and each segment is sent with a checksum.

  TCP/IP offers a **connection-oriented** service; however, it is *not* a virtual circuit. All sequence numbering, flow control, and error handling is done on an end-to-end basis (i.e, the two computers involved in the communication), as opposed to any of the intermediate gateways. The reason behind this approach can be summarized in one word: reliability. Without the use of virtual circuits, a router could fail and a new route through the network could be found. To help minimize bandwidth loss, TCP/IP uses selective retransmission (see Chapter 11).

  TCP is usually referred to as TCP/IP since the TCP protocol is interwoven with IP, meaning that structure (i.e., a slavish devotion to layering) was sacrificed for speed.

Operating systems such as Berkeley Unix and Linux support both TCP/IP and UDP (as does Microsoft Windows). In Berkeley Unix, the combination of the host Internet address and the port number was given the name **socket**, a name which has gained widespread acceptance.

The aforementioned three layers of the Internet architecture are commonly referred to as the **Internet**. Most users deal only with the highest layer of the Internet architecture, the Application layer. The Internet's Application layer consists of co-operating processes or tasks transferring data via TCP/IP and UDP. Most Internet applications are described in terms of the **client-server** model. A process offering a **service** is referred to as the **server**, while the process requiring the service is the **client**. In the client-server model, the client sends a **request** to the server; upon receipt of the request, the server performs the service. Upon completion, the server sends a **response** to the client. Since a process on a host is associated with a port, many services are associated with "well-known" ports, thereby facilitating the identification of the service by a client.

Some Internet application services include:

SMTP. SMTP (or Simple Mail Transfer Protocol), the Internet electronic mail protocol. This protocol operates atop TCP/IP and defines the steps involved in transferring electronic mail between two hosts.

Rlogin. A remote login service offered by most flavours of UNIX, allowing users on one UNIX machine to connect to a remote UNIX machine. Since multiple remote logins are possible, each login, once established, communicates through its own TCP/IP port.

TELNET. TELNET is similar to rlogin, with the exception that TELNET is intended for any remote Internet host; TELNET operates atop TCP/IP.

FTP. FTP or File Transfer Protocol, is a reliable file transfer protocol that uses both TCP/IP and TELNET. TCP/IP handles the transfer of the file, while TELNET allows the user to signon to the remote host from which the file transfer is to take place.

HTTP. HTTP or HyperText Transfer Protocol, is the most recent development of the **World Wide Web**, a tool intended to encourage and simplify access to the Internet. An HTTP-client allows a user to access and display **hypertext** documents with **links** to other documents; while an HTTP-server returns hypertext documents.

In all of the above examples, each service is associated its own well-known port; for example, SMTP is associated with port 25, TELNET with port 23, and HTTP, port 80.

The OSI model, unlike the Internet architecture, offers reliability at each of the four lowest layers. The Internet approach is to have the application service handle reliability on an end-to-end basis.

Although the Internet architecture is used widely for intranetwork communications (i.e., communications within a single network), it is actually intended for internetwork communications (i.e., communications between different networks).

15.5 Summary

This Chapter has considered some of the issues surrounding two closely related topics: the identification of entities other than hosts or nodes and the need for layering in communication systems.

Host (and node) identification is achieved by specifying a destination address in the network's frame. By itself, the destination host's unicast address cannot be used to uniquely identify an entity, such as a process, since a host may support tens or hundreds of processes. Instead, when transmitting a message to a remote entity, it is necessary to identify both the entity and its host. In some applications, such as electronic mail, the entity receiving the message may in turn distribute the message to other entities on the host.

Since it is unrealistic to expect a single process to handle functions such as mail distribution, the framing of packets, file transfer, and untold other tasks, many communication systems have been developed in a layered fashion. That is, the communication functions have been divided into a number of well-defined operations and assigned to a layer. Each layer offers the layer above it a number of services, while using the services of the layer below it.

There is another demand for layering: hiding network or manufacturer specific functions, thereby permitting the interconnection of software developed on different machines and on different networks. For layering to work properly, it is necessary to develop standards that are agreed to by the people that use the layers. A number of well-known layered architectures have been proposed and implemented since the late 1970s including the Open Systems Interconnection model, IEEE 802, and the Internet Protocol Architecture.

Since the late 1980s, there has been an explosive growth in the use of the Internet and the demand for client-server software. Although many applications seem novel, it is important to remember that ultimately, the information to be accessed is simply a stream of bits that utilizes an underlying communication system. Understanding the concepts associated with layered architectures makes it easier to understand these applications.

15.6 Exercises

1. Into which OSI layer should the following services be placed, justify each answer:

   (a) a time server.

   (b) a service that performs byte-swapping from Intel format to Motorola format (see Chapter 1).

   (c) the X.25 data-qualifier bit (the Q-bit) (see Chapter 13).

   (d) the functions of a ring network monitor.

2. UNIX sockets reserve a number of ports (typically the first 512) for 'well-known' services. Suggest reasons why this approach is taken.

   When answering this exercise, consider the effect on a process if the process arbitrarily bound to a reserved port number (such as TCP). What would happen if TCP messages started arriving at the process?

3. Write a simple connectionless socket system atop the Ethernet. Allow programs to bind to sockets and wait for incoming messages. Two possible test programs could be a time service and an echo service.

4. Many programmers feel that using message passing is an unnatural programming paradigm, preferring instead to use procedure calls. A programming paradigm known as **remote procedure calls** has been developed that permits programs on one machine to call a subroutine on a separate machine.

   The remote procedure call (or **RPC**) is implemented as a local procedure call to a *stub* procedure that creates a message containing the parameters and sends the message to a remote stub procedure. The remote stub unpacks the parameters and makes a local procedure call to the called (remote) procedure. Once the target procedure has completed its task, the results of the procedure are returned by the remote stub in a message to the local stub. The local stub unpacks the message, updates the necessary parameters and returns to the calling procedure.

   Implement RPCs atop an Ethernet layer. Issues that should be considered include how is call-by-value supported, how is call-by-address supported, and what happens if the remote procedure's host should fail.

   To test the software, write a remote procedure that accepts two numbers, adds them and returns the answer in a third number.

Appendix A

The ASCII Character Set

In its most basic form, all information within a computer is represented as a data structure consisting of one or more bits. The 'fundamental' data structure that most computer manufacturers refer to when describing their equipment is the **byte**.

The byte itself can be interpreted in a number of different ways; for example, as an integer or as a character. Various **character codes** have been developed over the past century that specify a standard to which various manufacturers conform when mapping the bits making up the byte into a 'character'. The character need not be printable; some characters are treated as **control codes** that are used in the transmission of information.

This Appendix is a brief overview of the ASCII character set.

A.1 7-bit ASCII

In the 1950s and 1960s, a number of organizations worked together to develop the 7-bit ASCII (American Standard Code for Information Interchange) code, formally known as ANSI (American National Standards Institute) standard X3.4-1977. ASCII supports 128 possible characters (i.e., 2^7): upper and lower case character, numbers, special symbols, and control characters. The ITU equivalent of ASCII is known as ITU International Alphabet Number 5 (or IA5).

The 7-bit ASCII code is shown in Table A.1. To read the bit pattern associated with a specific character, find the character in the table, concatenate the three bits at the top of the character's column with the four bits to the left of the character. For example, the letter 'S' is the bit pattern `101` concatenated with `0011`, or `10100111`.

The ASCII control characters are defined in Table A.2.

Originally, 7-bit ASCII was developed for machines supporting 7-bit bytes (actually eight bits in total: the eighth bit of a character was treated as a parity bit). However, since most bytes occupy eight bits within the machine, many manufacturers have extended the 7-bit ASCII character set to support a further 128 characters. For example, the PC treats bytes with values greater than 127 as special graphic characters, generated using the ALTMODE key.

Table A.1: The 7-bit ASCII character code

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Table A.2: The ASCII control characters

NUL The null character
SOH Start of header
STX Start of text
ETX End of text
EOT End of transmission
ENQ Enquiry
ACK Acknowledgement
BEL Bell
BS Backspace
HT Horizontal tab
LF Line feed
VT Vertical tab
FF Form feed
CR Carriage return
SO Shift out
SI Shift in

DLE Data Link Escape
DC1 Device Control 1
DC2 Device Control 2
DC3 Device Control 3
DC4 Device Control 4
NAK Negative Acknowledgement
SYN Synchronization
ETB End of transmission block
CAN Cancel
EM End of medium
SUB Substitution
ESC Escape
FS File separator
GS Group separator
RS Record separator
US Unit separator
DEL Delete

Index