

chapter 5

Sending and Receiving Data

Typically, you use sockets because your program needs to provide information to, or use information provided by, another program. There is no magic: any programs that exchange information must agree on how that information will be *encoded*—represented as a sequence of bits—as well as which program sends what information when, and how the information received affects the behavior of the program. This agreement regarding the form and meaning of information exchanged over a communication channel is called a *protocol*; a protocol used in implementing a particular application is an *application protocol*. In our echo example from the earlier chapters, the application protocol is trivial: neither the client’s nor the server’s behavior is affected by the *contents* of the messages they exchange. Because in most real applications the behavior of clients and servers depends on the information they exchange, application protocols are usually somewhat more complicated.

The TCP/IP protocols transport bytes of user data without examining or modifying them. This allows applications great flexibility in how they encode their information for transmission. Most application protocols are defined in terms of discrete *messages* made up of sequences of *fields*. Each field contains a specific piece of information encoded as a sequence of bits. The application protocol specifies exactly how these sequences of bits are to be arranged by the sender and interpreted, or *parsed*, by the receiver so that the latter can extract the meaning of each field. About the only constraint imposed by TCP/IP is that information must be sent and received in chunks whose length in bits is a multiple of eight. So from now on we consider messages to be sequences of *bytes*. Given this, it may be helpful to think of a transmitted message as a sequence or array of numbers, each between 0 and 255. That corresponds to the range of binary values that can be encoded in 8 bits: 00000000 for zero, 00000001 for one, 00000010 for two, and so on, up to 11111111 for 255.

When you build a program to exchange information via sockets with other programs, typically one of two situations applies: either you are designing/writing the programs on both sides of the socket, in which case you are free to define the application protocol yourself, or you are implementing a protocol that someone else has *already* specified, perhaps a protocol *standard*. In either case, the basic principles of encoding and decoding different types of information as bytes “on the wire” are the same. (By the way, everything in this chapter also applies if the “wire” is a file that is written by one program and then read by another.)

5.1 Encoding Integers

Let’s first consider the question of how integers—that is, groups of bits that can represent whole numbers—can be sent and received via sockets. In a sense, all types of information are ultimately encoded as fixed-size integers, so the ability to send and receive them is fundamental.

5.1.1 Sizes of Integers

We have seen that TCP and UDP sockets transmit sequences of *bytes*: groups of 8 bits, which can contain whole number values in the range 0–255. Sometimes it is necessary to send integers whose value might be bigger than 255; such integers must be encoded using multiple bytes. To exchange fixed-size, multibyte integers, the sender and receiver have to agree *in advance* on several things. The first is the *size* (in bytes) of each integer to be sent.

For example, an **int** might be stored as a 32-bit quantity. In addition to **int**, the C language defines several other integer types: **short**, **char**, and **long**; the idea is that these integers can be different sizes, and the programmer can use the one that fits the application. Unlike some languages, however, the C language does *not* specify the exact size of each of these primitive types. Instead, that is left up to the implementation. Thus, the size of a **short** integer can vary from platform to platform.¹ The C language specification does say that a **char** is no bigger than a **short**, which is no bigger than an **int**, which is no bigger than a **long**, which is no bigger than a **long long**. However, the specification does *not* require that these types actually be different sizes—it is technically possible for a **char** to be the same size as a **long**! On most platforms, however, the sizes do differ, and it is a safe bet that they do on yours, too.

So how do you determine the exact size of an **int** (or **char**, or **long**, or ...) on your platform? The answer is simple: use the `sizeof()` operator, which returns the amount of memory (in “bytes”) occupied by its argument (a type or variable) on the current platform. Here are a couple of things to note about `sizeof()`. First, the language specifies that `sizeof(char)` is

¹By “platform” in this book we mean the combination of compiler, operating system, and hardware architecture. The gcc compiler with the Linux operating system, running on Intel’s IA-32 architecture, is an example of a platform.

1—*always*. Thus in the C language a “byte” is the amount of space occupied by a variable of type **char**, and the units of `sizeof()` are actually `sizeof(char)`. But exactly how big is a C-language “byte”? That’s the second thing: the predefined constant `CHAR_BIT` tells how many bits it takes to represent a value of type **char**—usually 8, but possibly 10 or even 32.

Although it’s always possible to write a simple program to print the values returned by `sizeof()` for the various primitive integer types, and thus clear up any mystery about integer sizes on your platform, C’s lack of specificity about the size of its primitive integer types makes it a little tricky if you want to write portable code for sending integers of a specific size over the Internet. Consider the problem of sending a 32-bit integer over a TCP connection. Do you use an **int**, a **long**, or what? On some machines an **int** is 32 bits, while on others a **long** may be 32 bits.

The C99 language standard specification offers a solution in the form of a set of optional types: **int8_t**, **int16_t**, **int32_t**, and **int64_t** (along with their unsigned counterparts **uint8_t**, etc) all have the size (in bits) indicated by their names. On a platform where `CHAR_BIT` is eight,² these are 1, 2, 4 and 8 byte integers, respectively. Although these types may not be implemented on every platform, each *is* required to be defined if any native primitive type has the corresponding size. (So if, say, the size of an **int** on the platform is 32 bits, the “optional” type **int32_t** is *required* to be defined.) Throughout the rest of this chapter, we make use of these types to specify the precise size of the integers we want. We will also make use of the C99-defined type **long long**, which is typically larger than a **long**. Program `TestSizes.c` will print the value of `CHAR_BIT`, the sizes of all the primitive integer types, and the sizes of the fixed-size types defined by the C99 standard. (The program will not compile if any of the optional types are not defined on your platform.)

TestSizes.c

```

1  #include <limits.h>
2  #include <stdint.h>
3  #include <stdio.h>
4
5  int main(int argc, char *argv[]) {
6      printf("CHAR_BIT is %d\n\n", CHAR_BIT);           // Bits in a char (usually 8!)
7
8      printf("sizeof(char) is %d\n", sizeof(char)); // ALWAYS 1
9      printf("sizeof(short) is %d\n", sizeof(short));
10     printf("sizeof(int) is %d\n", sizeof(int));
11     printf("sizeof(long) is %d\n", sizeof(long));
12     printf("sizeof(long long) is %d\n\n", sizeof(long long));
13
14     printf("sizeof(int8_t) is %d\n", sizeof(int8_t));

```

²We are not aware of any modern general-purpose computing platform where `CHAR_BIT` differs from eight. Throughout the rest of this book, the value of `CHAR_BIT` is assumed, without comment, to be 8.

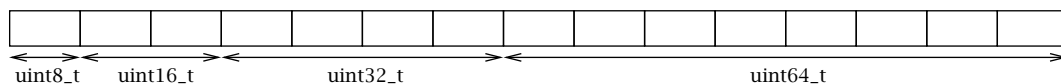
```

15  printf("sizeof(int16_t) is %d\n", sizeof(int16_t));
16  printf("sizeof(int32_t) is %d\n", sizeof(int32_t));
17  printf("sizeof(int64_t) is %d\n\n", sizeof(int64_t));
18
19  printf("sizeof(uint8_t) is %d\n", sizeof(uint8_t));
20  printf("sizeof(uint16_t) is %d\n", sizeof(uint16_t));
21  printf("sizeof(uint32_t) is %d\n", sizeof(uint32_t));
22  printf("sizeof(uint64_t) is %d\n", sizeof(uint64_t));
23  }

```

TestSizes.c

To make things a little more concrete, in the remainder of this section we'll consider the problem of encoding a sequence of integers of different sizes—specifically, of 1, 2, 4, and 8 bytes, in that order. Thus, we need a total of 15 bytes, as shown in the following figure.

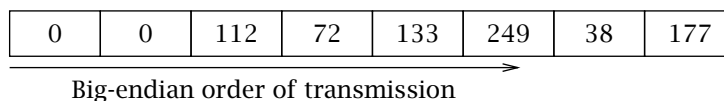


We'll consider several different methods of doing this, but in all cases we'll assume that the C99 fixed-size types are supported.

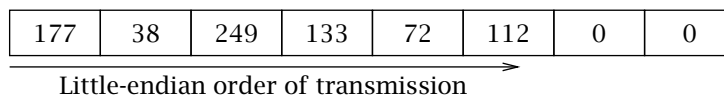
5.1.2 Byte Ordering

Once the sender and receiver have specified the sizes of the integers to be transmitted, they need to agree on some other aspects. For integers that require more than one byte to encode, they have to answer the question of which *order* to send the bytes in.


There are two obvious choices: start at the “right” end of the number, with the least significant bits—so-called *little-endian* order—or at the left end, with the most significant bits—*big-endian* order. (Note that the ordering of *bits within bytes* is, fortunately, handled by the implementation in a standard way.) Consider the **long long** value 123456787654321L. Its 64-bit representation (in hexadecimal) is 0x0000704885F926B1. If we transmit the bytes in big-endian order, the sequence of (decimal) byte values will look like this:



If we transmit them in little-endian order, the sequence will be:



The main point is that for any multibyte integer quantity, the sender and receiver need to agree on whether big-endian or little-endian order will be used.³ If the sender were to use little-endian order to send the above integer, and the receiver were expecting big-endian, instead of the correct value, the receiver would interpret the transmitted 8-byte sequence as the value 12765164544669515776L.

Most protocols that send multibyte quantities in the Internet today use big-endian byte order; in fact, it is sometimes called *network byte order*. The byte order used by the hardware (whether it is big- or little-endian) is called the *native byte order*. C language platforms typically provide functions that allow you to convert values between native and network byte orders; you may recall that we have already encountered `htons()` and `htonl()`. Those routines, along with `ntohl()` and `ntohs()`, handle the conversion for typical integer sizes. The functions whose names end in “l” (for “long”) operate on 32-bit quantities, while the ones ending in “s” (for “short”) operate on 16-bit quantities. The “h” stands for “host,” and the “n” for “network”. Thus `htons()` was used in Chapter 2 to convert 16-bit port numbers from host byte order to network byte order, because *the Sockets API routines deal only with addresses and ports in network byte order*. That fact is worth repeating, because beginning programmers are often bitten by forgetting it: **addresses and ports that cross the Sockets API are always in network byte order**. We will make full use of these order-conversion functions shortly. 

5.1.3 Signedness and Sign Extension

One last detail on which the sender and receiver must agree: whether the numbers transmitted will be *signed* or *unsigned*. We’ve said that bytes contain values in the range 0 to 255 (decimal). That’s true if we don’t need negative numbers, but they are necessary for many applications. Fortunately, the same 255-bit patterns can be interpreted as integers in the range -128 to 127 . *Two’s-complement* representation is the usual way of representing such signed numbers. For a k -bit number, the two’s-complement representation of the negative integer $-n$, $1 \leq n \leq 2^{k-1}$, is the binary value of $2^k - n$. The nonnegative integer p , $0 \leq p \leq 2^{k-1} - 1$, is encoded simply by the k -bit binary value of p . Thus, given k bits, we can represent values in the range -2^{k-1} through $2^{k-1} - 1$ using two’s-complement. Note that the most significant bit (msb) tells whether the value is positive (msb = 0) or negative (msb = 1). On the other hand, a k -bit *unsigned* integer can encode values in the range 0 through $2^k - 1$ directly. So for example, the 32-bit value `0xFFFFFFFF` (the all-ones value) when interpreted as a signed, two’s-complement number represents -1 ; when interpreted as an unsigned integer, it represents 4294967295. The signedness of the integers being transmitted should be determined by the range of values that need to be encoded.

Some care is required when dealing with integers of different signedness because of *sign-extension*. When a signed value is copied to any wider type, the additional bits are copied from the sign (i.e., most significant) bit. By way of example, suppose the variable `smallInt` is of type

³Other orders are possible for integers bigger than 2 bytes, but we know of no modern systems that use them.

int8_t, that is, a signed, 8-bit integer, and *widerInt* is of type **int16_t**. Suppose also that *smallInt* contains the (binary) value 01001110 (i.e., decimal 78). The assignment statement:

```
widerInt := smallInt;
```

places the binary value 0000000001001110 into *widerInt*. However, if *smallInt* has the value 11100010 (decimal -30) before the assignment statement, then afterward *widerInt* contains the binary value 111111111100010.

Now suppose the variable *widerUInt* is of type **uint16_t**, and *smallInt* again has the value -30, and we do this assignment:

```
widerUInt := smallInt;
```

What do you think the value of *widerUInt* is afterward? The answer is again 111111111100010, because the sign of *smallInt*'s value is extended *as it is widened* to fit in *widerUInt*, even though the latter variable is unsigned. If you print the resulting value of *widerUInt* as a decimal number, the result will be 65506. On the other hand, if we have a variable *smallUInt* of type **uint8_t**, containing the same binary value 11100010, and we copy its value to the wider unsigned variable:

```
widerUInt := smallUInt;
```

and then print the result, we get 226, because the value of an unsigned integer type is—reasonably enough—*not* sign-extended.

One final point to remember: when expressions are evaluated, values of variables are widened (if needed) to the “native” (**int**) size before any computation occurs. Thus, if you add the values of two **char** variables together, the type of the result will be **int**, not **char**:

```
char a,b;
printf("sizeof(a+b) is %d\n", sizeof(a+b));
```

On the platform used in writing this book, this code prints “sizeof(a+b) is 4”. The type of the argument to `sizeof()`—the expression `a + b`—is **int**. This is generally not an issue, but you need to be aware that *sign-extension also occurs during this implicit widening*.

5.1.4 Encoding Integers by Hand

Having agreed on byte ordering (we'll use big-endian) and signedness (the integers are all unsigned), we're ready to construct our message. We'll first show how to do it “by hand,” using shifting and masking operations. The program `BruteForceCoding.c` features a method `EncodeIntBigEndian()` that places any given primitive integer value as a sequence of the specified number of bytes at a specified location in memory, using big-endian representation. The method takes four arguments: a pointer to the starting location where the value is to be placed; the value to be encoded (represented as a 64-bit unsigned integer, which is big enough to hold any of the other types); the offset in the array at which the value should start; and the size in bytes of the value to be written. Of course, whatever we encode at the sender must be decodable

at the receiver. The `DecodeIntBigEndian()` method handles decoding a byte sequence of a given length into a 64-bit integer, interpreting it as a big-endian sequence.

These methods treat all quantities as unsigned; see the exercises for other possibilities.

BruteForceCoding.c

```

1  #include <stdint.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <limits.h>
5  #include "Practical.h"
6
7  const uint8_t val8 = 101; // One hundred and one
8  const uint16_t val16 = 10001; // Ten thousand and one
9  const uint32_t val32 = 100000001; // One hundred million and one
10 const uint64_t val64 = 10000000000001L; // One trillion and one
11 const int MESSAGELENGTH = sizeof(uint8_t) + sizeof(uint16_t) + sizeof(uint32_t)
12     + sizeof(uint64_t);
13
14 static char stringBuf[BUFSIZE];
15 char *BytesToDecString(uint8_t *byteArray, int arrayLength) {
16     char *cp = stringBuf;
17     size_t bufSpaceLeft = BUFSIZE;
18     for (int i = 0; i < arrayLength && bufSpaceLeft > 0; i++) {
19         int strl = snprintf(cp, bufSpaceLeft, "%u ", byteArray[i]);
20         bufSpaceLeft -= strl;
21         cp += strl;
22     }
23     return stringBuf;
24 }
25
26 // Warning: Untested preconditions (e.g., 0 <= size <= 8)
27 int EncodeIntBigEndian(uint8_t dst[], uint64_t val, int offset, int size) {
28     for (int i = 0; i < size; i++) {
29         dst[offset++] = (uint8_t) (val >> ((size - 1) - i) * CHAR_BIT);
30     }
31     return offset;
32 }
33
34 // Warning: Untested preconditions (e.g., 0 <= size <= 8)
35 uint64_t DecodeIntBigEndian(uint8_t val[], int offset, int size) {
36     uint64_t rtn = 0;
37     for (int i = 0; i < size; i++) {
38         rtn = (rtn << CHAR_BIT) | val[offset + i];
39     }
40     return rtn;

```

```

41 }
42
43 int main(int argc, char *argv[]) {
44     uint8_t message[MESSAGELENGTH]; // Big enough to hold all four values
45
46     // Encode the integers in sequence in the message buffer
47     int offset = 0;
48     offset = EncodeIntBigEndian(message, val8, offset, sizeof(uint8_t));
49     offset = EncodeIntBigEndian(message, val16, offset, sizeof(uint16_t));
50     offset = EncodeIntBigEndian(message, val32, offset, sizeof(uint32_t));
51     offset = EncodeIntBigEndian(message, val64, offset, sizeof(uint64_t));
52     printf("Encoded message:\n%s\n", BytesToDecString(message, MESSAGELENGTH));
53
54     uint64_t value =
55         DecodeIntBigEndian(message, sizeof(uint8_t), sizeof(uint16_t));
56     printf("Decoded 2-byte integer = %u\n", (unsigned int) value);
57     value = DecodeIntBigEndian(message, sizeof(uint8_t) + sizeof(uint16_t)
58         + sizeof(uint32_t), sizeof(uint64_t));
59     printf("Decoded 8-byte integer = %llu\n", value);
60
61     // Show signedness
62     offset = 4;
63     int iSize = sizeof(int32_t);
64     value = DecodeIntBigEndian(message, offset, iSize);
65     printf("Decoded value (offset %d, size %d) = %lld\n", offset, iSize, value);
66     int signedVal = DecodeIntBigEndian(message, offset, iSize);
67     printf("...same as signed value %d\n", signedVal);
68 }

```

BruteForceCoding.c

1. **Declarations and inclusions:** lines 1-12
 - **Library functions and constants:** lines 1-5
 - **Integer variables with values to be encoded:** lines 7-10
 - **Message length computation:** lines 11-12

The language spec says the initializer expression evaluates to 15; we include it for completeness.
2. **BytesToDecString():** lines 14-24

This auxiliary routine takes an array of bytes and its length, and returns a string containing the value of each byte as a decimal integer in the range 0 to 255.
3. **EncodeIntBigEndian():** lines 27-32

We iterate over the given value *size* times. On each iteration, the right-hand side of the assignment shifts the value to be encoded to the right, so the byte we are interested in is

in the low-order 8 bits. The resulting value is then *cast* to the type `uint8_t`, which throws away all but the low-order 8 bits, and placed in the array at the appropriate location. The ending value of *offset* is returned so that the caller does not have to recompute it when encoding a sequence of integers (as we will).

4. `DecodeIntBigEndian()`: lines 35–41

We construct the value in a 64-bit integer variable. Again we iterate *size* times, each time shifting the accumulated value left and bitwise-ORing in the next byte's value.

5. **Demonstrate methods**: lines 43–68

■ **Declare buffer (array of bytes) to receive series of integers**: line 44

■ **Encode items**: lines 47–51

The integers are encoded into the array in the sequence described earlier.

■ **Print contents of encoded array**: line 52

■ **Extract and display some values from encoded message**: lines 54–59

Output should show the decoded values equal to the original constants.

■ **Signedness effects**: lines 62–67

At offset 4, the byte value is 245 (decimal); because it has its high-order bit set, if it is the high-order byte of a signed value, that value will be considered negative. We show this by decoding the 4 bytes starting at offset 4 and placing the result into both a signed integer and an unsigned integer.

Note that we might consider testing several preconditions at the beginning of `EncodeIntBigEndian()` and `DecodeIntBigEndian()`, such as $0 \leq \text{size} \leq 8$ and `dst != NULL`. Can you name any others?

Running the program produces output showing the following (decimal) byte values:

101	39	17	5	245	225	1	0	0	0	232	212	165	16	1
← byte		← short		← int				← long						

As you can see, the brute-force method requires the programmer to do quite a bit of work: computing and naming the offset and size of each value, and invoking the encoding routine with the appropriate arguments. Fortunately, alternative ways to build messages are often available. We discuss these next.

5.1.5 Wrapping TCP Sockets in Streams

A way of encoding multibyte integers for transmission over a stream (TCP) socket is to use the built-in **FILE**-stream facilities—the same methods you use with *stdin*, *stdout*, and so on. To access these facilities, you need to associate one or more **FILE** streams with your socket descriptor, via the `fdopen()` call.

```
FILE *fdopen(int socketdes, const char *mode)
int fclose(FILE * stream)
int fflush(FILE * stream)
```

The `fdopen()` function “wraps” the socket in a stream and returns the result (or `NULL` if an error occurs); it is as if you could call `fopen()` on a network address. This allows buffered I/O to be performed on the socket via operations like `fgets()`, `fputs()`, `fread()` and `fwrite()`; the “mode” argument to `fdopen()` takes the same values as `fopen()`. `fflush()` pushes buffered data to underlying socket. `fclose()` closes the stream along with the underlying socket. `fflush()` causes any buffered to be sent over the underlying socket.

```
size_t fwrite(const void * ptr, size_t size, size_t nmemb, FILE * stream)
size_t fread(void * ptr, size_t size, size_t nmemb, FILE * stream)
```

The `fwrite()` method writes the specified number of objects of the given size to the stream. The `fread()` method goes in the other direction, reading the given number of objects of the given size from the given stream and placing them sequentially in the location pointed to by `ptr`. Note that the sizes are given in units of `sizeof(char)`, while the return values of these methods are the number of objects read/written, *not* the number of bytes. In particular, `fread()` never reads *part* of an object from the stream, and similarly `fwrite()` never writes a partial object. If the underlying connection terminates, these methods will return a short item count.


By giving different sizes to `fwrite()`, we can output our message to the stream sequentially. Assume that the variables `val8`, `val16`, and the rest are declared and initialized as in `BruteForceCoding.c` and that the integer variable `sock` is the descriptor of the connected TCP socket over which we want to write our message. Finally, assume that `htonll()` is a function that converts 64-bit integers from host to network byte order (see Exercise 2). Then we can write our message to the socket one integer at a time:

```
sock = socket(/*...*/);
/* ... connect socket ...*/
// wrap the socket in an output stream
FILE *outstream = fdopen(sock, "w");
// send message, converting each object to network byte order before sending
if (fwrite(&val8, sizeof(val8), 1, outstream) != 1) ...
val16 = htons(val16);
if (fwrite(&val16, sizeof(val16), 1, outstream) != 1) ...
val32 = htonl(val32);
if (fwrite(&val32, sizeof(val32), 1, outstream) != 1) ...
val64 = htonll(val64);
if (fwrite(&val64, sizeof(val64), 1, outstream) != 1) ...
```

```
fflush(outstream); // immediately flush stream buffer to socket
...                // do other work...
fclose(outstream); // flushes stream and closes socket
```

So much for the sending side. How does the receiver recover the transmitted values? As you might expect, the receiving side takes the analogous steps using `fread()`. Suppose now the variable `csock` contains the socket descriptor for a TCP connection, that the received values are to be placed in the variables `rcv8`, `rcv16`, `rcv32`, and `rcv64` of the expected types, and that `ntohll()` is a network-to-host byte order converter for 64-bit types.

```
/* ... csock is connected ...*/
// wrap the socket in an input stream
FILE *instream = fdopen(csock, "r");
// receive message, converting each received object to host byte order
if (fread(&rcv8, sizeof(rcv8), 1, instream) != 1) ...
if (fread(&rcv16, sizeof(rcv16), 1, instream) != 1) ...
rcv16 = ntohs(rcv16); // convert to host order
if (fread(&rcv32, sizeof(rcv32), 1, instream) != 1) ...
rcv32 = ntohl(rcv32);
if (fread(&rcv64, sizeof(rcv64), 1, instream) != 1) ...
rcv64 = ntohll(rcv64);
...
fclose(instream); // closes the socket connection!
```

Among the advantages of using buffered **FILE**-streams with sockets is the ability to “put back” a byte after reading it from the stream (via `ungetc()`); this can sometimes be useful when parsing messages. **We must emphasize, however, that FILE-streams can *only* be used with TCP sockets.** 

5.1.6 Structure Overlays: Alignment and Padding

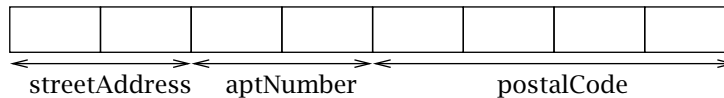
The most common approach to constructing messages containing binary data (i.e., multibyte integers) involves overlaying a C structure on a section of memory and assigning to the fields of the structure directly. This is possible because the C language specification *explicitly defines how structures are laid out in memory by the compiler*. It has this feature because it was designed for implementing operating systems, where efficiency is a primary concern, and it is often necessary to know precisely how data structures are represented. In combination with the facilities for byte order conversion that we saw above, this makes it rather simple to construct and parse messages made up of integers of particular sizes. (There is a significant caveat, however, that we will encounter shortly.)

Suppose for a moment that we are dealing with three integer components of an address: the number on the street, which ranges from 1 to about 12000; an apartment number that never exceeds 8000 (a nonapartment address uses an apartment number of -1); and a postal code, which is a five-digit number between 10000 and 99999. The first two components can be represented using 16-bit integers; the third is too big for that, so we’ll use a 32-bit integer for

it. If we want to pass those quantities around inside a program, we could declare a structure and pass around pointers to it:

```
struct addressInfo {
    uint16_t  streetAddress;
    int16_t   aptNumber;
    uint32_t   postalCode;
} addrInfo;
```

Clearly, such a structure is useful for passing data around in a program, but can we use it to pass information between programs over the Internet? The answer is yes. The C specification says that this structure will be laid out as follows in memory:



To exchange information between programs, we can simply use the layout of the structure as the message format, and take the contents directly from the structure and send them over the network (after any needed byte-order conversions). If the variable `addrInfo` declared above has been initialized to contain the values we want to send, and `sock` represents a connected socket as usual, we could send the 8-byte message using the following code:

```
// ... put values in addrInfo ...
// convert to network byte order
addrInfo.streetAddress = htons(addrInfo.streetAddress);
addrInfo.aptNumber = htons(addrInfo.aptNumber);
addrInfo.postalCode = htonl(addrInfo.postalCode);
if (send(sock, &addrInfo, sizeof(addrInfo), 0) != sizeof(addrInfo)) ...
```

On the receiving end, we can again use a buffered input stream (see previous section) and `fread()` to handle getting the right number of bytes into the structure. (Otherwise we have to use a loop because, as we saw earlier, there is no guarantee that all the bytes of the message will be returned in a single call to `recv()`). Once that's done, all we have to do is take care of byte ordering:

```
struct addressInfo addrInfo;
// ... sock is a connected socket descriptor ...
FILE *instream = fdopen(sock, "r");
if (fread(&addrInfo, sizeof(struct addressInfo), 1, instream) != 1) {
    // ... handle error
}
// convert to host byte order
addrInfo.streetAddress = ntohs(addrInfo.streetAddress);
addrInfo.aptNumber = ntohs(addrInfo.aptNumber);
addrInfo.postalCode = ntohl(addrInfo.postalCode);
// use information from message...
```

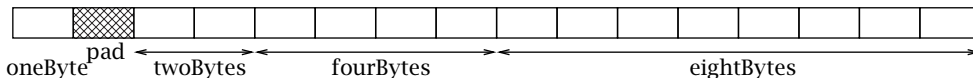
Now, it would seem that we could construct our 15-byte message using a declaration like the following:

```
struct integerMessage {
    uint8_t oneByte;
    uint16_t twoBytes;
    uint32_t fourBytes;
    uint64_t eightBytes;
}
```

Alas, this doesn't work, because the C language rules for laying out data structures include specific *alignment* requirements, including that the fields within a structure begin on certain boundaries based on their type. The main points of the requirements can be summarized as follows:

- Data structures are maximally aligned. That is, the address of any instance of a structure (including one in an array) will be divisible by the size of its largest native integer field.
- Fields whose type is a multibyte integer type are aligned to their size (in bytes). Thus, an `uint32_t` integer field's beginning address is always divisible by four, and a `uint16_t` integer field's address is guaranteed to be divisible by two.

To enforce these constraints, the compiler may add *padding* between the fields of a structure. Because of this, the size of the structure declared above is not 15, but 16. To see why, let's consider the constraints that apply. First, the whole structure must begin on an address divisible by 8. The *twoBytes* field must be at an even address, *fourBytes* must be at an address divisible by four, and *eightBytes*' address must be divisible by eight. All of these constraints will be satisfied if a single byte of padding is inserted between the *oneByte* field and the *twoBytes* field, like this:



The contents of bytes added by the compiler as padding are undefined. Thus if the declaration above is used, and the receiver is expecting the original unpadded layout, incorrect behavior is the likely result.

The best solution is to avoid the need for padding by laying out messages so that it is not needed. Unfortunately (and somewhat surprisingly), that's not always possible. In the case of our four-integer example, if we arranged the message with fields in the opposite order:

```
struct backwardMessage {
    uint64_t eightBytes;
    uint32_t fourBytes;
    uint16_t twoBytes;
    uint8_t oneByte;
}
```

no padding would be required between the fields. However, `sizeof(struct backwardMessage)` would nevertheless return 16, not 15. This is because one byte of padding would be required *between* instances of the structure (in an array, for example), in order to satisfy the first (maximal-alignment) constraint. (The invariant that must be maintained is that in an array of structures, the address of one element plus the `sizeof()` the structure yields the address of the subsequent element.) As a consequence, there is no way to specify a structure containing any multibyte integer for which `sizeof()` returns an odd number.

If we can't eliminate compiler-required padding, we can, as a last resort, include it in the message format specification. For our example, the message format could be defined with an *explicit* padding byte after the first field:

```
struct integerMessage2 {
    uint8_t oneByte;
    uint8_t padding;    // Required for alignment
    uint16_t twoBytes;
    uint32_t fourBytes;
    uint64_t eightBytes;
}
```

This structure is laid out in memory *exactly* like the originally declared **integerMessage**, except that the contents of the padding byte can now be controlled and accessed by the programmer. We'll see more examples of the use of structures to encode data in Section 5.2.3.

5.1.7 Strings and Text

Old-fashioned *text*—strings of printable (displayable) characters—is perhaps the most common way to represent information. We've already seen examples of sending and receiving strings of text in our echo client and server; you are probably also accustomed to having programs you write generate text output.

Text is convenient because we deal with it all the time: information is represented as strings of characters in books, newspapers, and on computer displays. Thus, once we know how to encode text for transmission, it is straightforward to send most any other kind of data: simply represent it as text, then encode the text. You know how to represent numbers and boolean values as strings of text—for example "123478962", "6.02e23", "true", "false". In this section we deal with the question of encoding such strings as byte sequences.

To that end, we first need to recognize that text is made up of sequences of symbols, or *characters*. The C language includes the primitive type **char** for representing characters, but does not include a primitive type for strings. Strings in C are traditionally represented as arrays of **char**. A **char** value in C is represented internally as an integer. For example, the character 'a', that is, the symbol for the letter 'a', corresponds to the integer 97. The character 'X' corresponds to 88, and the symbol '!' (exclamation mark) corresponds to 33.

A mapping between a set of symbols and a set of integers is called a *coded character set*. You may have heard of the coded character set known as *ASCII*—American Standard Code for Information Interchange. ASCII maps the letters of the English alphabet, digits, punctuation, and some other special (nonprintable) symbols to integers between 0 and 127. It has been used

for data transmission since the 1960s, and is used extensively in application protocols such as HTTP (the protocol used for the World Wide Web), even today. The C language specifies a *basic character set* that is a subset of ASCII. The importance of ASCII (and the C basic character set) is that strings containing only characters from the basic set can be encoded using one byte per character. (Note that in our echo client and server from Chapter 2 the encoding was *irrelevant*, because the server did not interpret the received data at all.)

This is well and good if you speak and use a language (like English) that can be represented using a small number of symbols. However, consider that no more than 256 distinct symbols can be encoded using one byte per symbol, and that a large fraction of the world's people use languages that have more than 256 symbols, and therefore have to be encoded using more than 8 bits per character. Clearly, the use of C presents significant challenges for implementing code that is “internationalizable.” Mandarin is just one prominent example of a language that requires thousands of symbols.

The C99 extensions standard defines a type **wchar_t** (“wide character”) to store characters from charsets that may use more than one byte per symbol. In addition, various library functions are defined that support conversion between byte sequences and arrays of **wchar_t**, in both directions. (In fact, there is a wide character string version of virtually every library function that operates on character strings.) To convert back and forth between wide strings and encoded char (byte) sequences suitable for transmission over the network, we would use the `wcstombs()` (“wide character string to multibyte string”) and `mbstowcs()` functions.

```
#include <stdlib.h>
size_t wcstombs(char *restrict s, const wchar_t *restrict pwcs, size_t n);
size_t mbstowcs(wchar_t *restrict pwcs, const char *restrict s, size_t n);
```

The first of these converts a sequence of wide characters from the array pointed to by `pwcs` into a sequence of multibyte characters, and stores these multibyte characters into the array pointed to by `s`, stopping if the next conversion would exceed the limit of `n` total bytes or if a null character is stored. The second does the same thing in the other direction.

As we have seen, for a coded character set that requires larger integer values, there is more than one way to encode those values for transmission over the network. Thus, it is necessary that sender and receiver agree on how those integers will be encoded as byte sequences.

As an example, consider a platform that uses 16-bit integers internally to represent characters, and whose character set includes ASCII as a subset—that is, the character ‘a’ maps to 97, ‘X’ to 88, and so on. Using the wide character facilities, we can declare a wide character string:

```
wchar_t testString[] = "Test!";
```

The internal representation of this string uses six 16-bit integers (including one for the terminating null (0) character). However, there are several ways to encode those integers as a sequence of 8-bit bytes:

- Each character can be represented using 2 bytes, in big-endian order. In that case, the result of calling `wcstombs()` on the wide string “Test!” would be the following sequence: 0, 84, 0, 101, 0, 115, 0, 116, 0, 33.

- Alternatively, we could use 2 bytes in little-endian order, which would give: 84, 0, 101, 0, 115, 0, 116, 0, 33, 0.
- We could use an encoding that maps symbols in the original ASCII character set to single bytes (the same as their ASCII encoding), and uses 2 bytes for symbols beyond ASCII. If the high-order bit of a byte is set, it indicates the next symbol is encoded with 2 bytes; otherwise it is an ASCII symbol encoded with a single byte. In this case the result after calling `wcstombs()` would be the 5-byte sequence 84, 101, 115, 116, 33.

Note that in these examples we did *not* include the terminating null (0) character. The terminating null is an artifact of the language, and not part of the string itself. It therefore should not be transmitted with the string unless the protocol explicitly specifies that method of marking the end of the string.

The bad news is that C99's wide character facilities are *not* designed to give the programmer explicit control over the encoding scheme. Indeed, they assume a single, fixed charset defined according to the "locale" of the platform. Although the facilities support a variety of charsets, they do not even provide the programmer any way to learn which charset or encoding is in use. In fact, the C99 standard states in several situations that the effect of changing the locale's charset at runtime is undefined. What this means is that if you want to implement a protocol using a particular charset, you'll have to implement the encoding yourself.

5.1.8 Bit-Diddling: Encoding Booleans

Bitmaps are a very compact way to encode boolean information, which is often used in protocols. The idea of a bitmap is that each of the bits of an integer type can encode one boolean value—typically with 0 representing false and 1 representing true. To be able to manipulate bitmaps, you need to know how to set and clear individual bits using C's "bit-diddling" operations. A *mask* is an integer value that has one or more specific bits set to 1, and all others cleared (i.e., 0). We'll deal here mainly with 32-bit maps and masks, but everything we say applies to types of other sizes as well.

Let's number the bits of an integer's binary representation from 0 to 31, where bit 0 is the least significant bit. In general, the `uint32_t` value that has a 1 in bit position i , and a zero in all other bit positions, is just 2^i . So bit 5 corresponds to the number 32, bit 12 to 4096, and so forth. Here are some example mask declarations:

```
const int BIT5 = (1<<5);
const int BIT7 = 0x80;
const int BITS2AND3 = 12;    // 8+4
int bitmap = 128;
```

To set a particular bit in an `int` variable, combine it with the mask for that bit using the bitwise-OR operation (`|`):

```
bitmap |= BIT5;
// bit 5 is now one
```


To *clear* a particular bit, bitwise-AND it with the *bitwise complement* of the mask for that bit (which has ones everywhere except the particular bit, which is zero). The bitwise-AND operation in C is `&`, while the bitwise-complement operator is `~`.

```
bitmap &= ~BIT7;
// bit 7 is now zero
```

You can set and clear multiple bits at once by OR-ing together the corresponding masks:

```
// clear bits 2, 3 and 5
bitmap &= ~(BITS2AND3|BIT5);
```

To test whether a bit is set, compare the result of the bitwise-AND of the mask and the value with zero:

```
bool bit6Set = (bitmap & (1<<6)) != 0;
```

5.2 Constructing, Framing, and Parsing Messages

We close this chapter with an example illustrating the application of the foregoing techniques in implementing a protocol specified by someone else. The example is a simple “voting” protocol as shown in Figure 5.1. Here a client sends a *request* message to the server; the message contains a candidate ID, which is an integer between 0 and 1000. Two types of requests are supported. An *inquiry* asks the server how many votes have been cast for the given candidate. The server sends back a *response* message containing the original candidate ID and the vote total as of the time the request was received for that candidate. A *voting* request actually casts a vote for the indicated candidate. The server again responds with a message containing the candidate ID and the vote total (which now includes the vote just cast). The protocol runs over stream sockets using TCP.

Normally, the “wire format” of the protocol messages would be precisely defined as part of the protocol. As we have seen, we might encode it in a number of ways: we could represent the information using strings of text, or as binary numbers. In order to illustrate all the

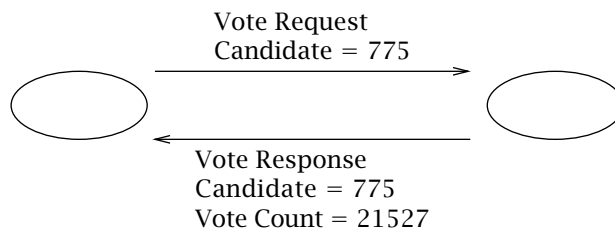


Figure 5.1: Voting protocol.

techniques described above, we will specify several different versions of the “wire format.” This also helps us to make a point about protocol implementation.

When implementing any protocol, it is good practice to hide the details of the way messages are encoded from the main program logic. We’ll illustrate that here by using a generic structure in the main programs to pass information to/from functions that process messages sent/received over the socket. This allows us to use the same client and server code with different implementations of the “wire format” processing. The **VoteInfo** structure, defined in `VoteProtocol.h`, contains everything needed to construct a message: the candidate ID number (an integer), the count of votes for that candidate (a 64-bit integer), a boolean indicating whether the message is an “inquiry” (inquiries do not affect the vote count), and another boolean indicating whether the message to be sent is a response sent from client to server (true), or a request (false). Also defined in this file are constants for the largest allowable candidate ID number and the maximum length of an encoded message on the wire. (The latter helps the programs to size their buffers.)

```
struct VoteInfo {
    uint64_t count;    // invariant: !isResponse => count==0
    int candidate;    // invariant: 0 <= candidate <= MAX_CANDIDATE
    bool isInquiry;
    bool isResponse;
};

typedef struct VoteInfo VoteInfo;

enum {
    MAX_CANDIDATE = 1000,
    MAX_WIRE_SIZE = 500
};
```

Note that we have defined a single **VoteInfo** structure for the information contained in *both* request messages and response messages. This, along with a proper control structure, allows reuse of the same message-processing code for both client and server, and both request and response.

The message-processing code is responsible for encoding the information from a **VoteInfo** structure and transmitting it over a stream socket, as well as for receiving data from a TCP socket, parsing the incoming vote protocol message (if any), and filling in a **VoteInfo** structure with the received information.

A clean design further decomposes the process into two parts. The first is concerned with *framing*, or marking the boundaries of the message, so the receiver can find it in the stream. The second is concerned with the actual encoding of the message, whether it is represented using text or binary data. Notice that these two parts can be independent of each other, and in a well-designed protocol they *should* be separated. In other words, we can specify the mechanism for framing the message as a whole separately from the encoding of its different fields. And that is what we shall do. Our job will be somewhat easier if we can use stream-processing functions, so we will have the client and server wrap the connected socket in a **FILE** stream for input and another for output.

The interface to the framing code is defined as follows in `Framer.h`.

```
int GetNextMsg(FILE *in, uint8_t *buf, size_t bufSize);
int PutMsg(uint8_t buf[], size_t msgSize, FILE *out);
```

The method `GetNextMsg()` reads data from the given stream and places it in the given buffer until it runs out of room or determines that it has received a complete message. It returns the number of bytes placed in the buffer (all framing information is stripped). The method `PutMsg()` adds framing information to the message contained in the given buffer, and writes both message and framing information to the given stream. Note that neither of these methods needs to know *anything* about the message content.

The interface to the encoding and parsing code is defined in `VoteEncoding.h` as follows:

```
bool Decode(uint8_t *inBuf, size_t mSize, VoteInfo *v);
size_t Encode(VoteInfo *v, uint8_t *outBuf, size_t bufSize);
```

The `Encode()` method takes a **VoteInfo** structure as input and converts it to a sequence of bytes according to a particular wire format encoding; it returns the size of the resulting byte sequence. The `Decode()` method takes a byte sequence of a specified size and parses it as a message according to the protocol, filling in the **VoteInfo** with the information from the message. It returns `TRUE` if the message was successfully parsed, and `FALSE` otherwise.

Given these interfaces to the framing and parsing code, we can now describe the voting client and server programs that will use these methods. The client is straightforward: the candidate ID is given as a command-line argument, along with a flag indicating that the transaction is an inquiry (by default it is a vote request). Upon sending the request, the client waits for the response and then closes the connection when it is received.

VoteClientTCP.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <stdint.h>
5 #include <unistd.h>
6 #include <errno.h>
7 #include <sys/socket.h>
8 #include <netinet/in.h>
9 #include <arpa/inet.h>
10 #include <netdb.h>
11 #include "Practical.h"
12 #include "VoteProtocol.h"
13 #include "Framer.h"
14 #include "VoteEncoding.h"
15
16 int main(int argc, char *argv[]) {
17     if (argc < 4 || argc > 5) // Test for correct # of args
```

```

18     DieWithUserMessage("Parameter(s)", "<Server> <Port/Service> <Candidate> [I]");
19
20     char *server = argv[1];    // First arg: server address/name
21     char *service = argv[2];  // Second arg: string to echo
22     // Third arg: server port/service
23     int candi = atoi(argv[3]);
24     if (candi < 0 || candi > MAX_CANDIDATE)
25         DieWithUserMessage("Candidate # not valid", argv[3]);
26
27     bool inq = argc > 4 && strcmp(argv[4], "I") == 0;
28
29     // Create a connected TCP socket
30     int sock = SetupTCPClientSocket(server, service);
31     if (sock < 0)
32         DieWithUserMessage("SetupTCPClientSocket() failed", "unable to connect");
33
34     FILE *str = fdopen(sock, "r+"); // Wrap for stream I/O
35     if (str == NULL)
36         DieWithSystemMessage("fdopen() failed");
37
38     // Set up info for a request
39     VoteInfo vi;
40     memset(&vi, 0, sizeof(vi));
41
42     vi.isInquiry = inq;
43     vi.candidate = candi;
44
45     // Encode for transmission
46     uint8_t outbuf[MAX_WIRE_SIZE];
47     size_t reqSize = Encode(&vi, outbuf, MAX_WIRE_SIZE);
48
49     // Print info
50     printf("Sending %d-byte %s for candidate %d...\n", reqSize,
51           (inq ? "inquiry" : "vote"), candi);
52
53     // Frame and send
54     if (PutMsg(outbuf, reqSize, str) < 0)
55         DieWithSystemMessage("PutMsg() failed");
56
57     // Receive and print response
58     uint8_t inbuf[MAX_WIRE_SIZE];
59     size_t respSize = GetNextMsg(str, inbuf, MAX_WIRE_SIZE); // Get the message
60     if (Decode(inbuf, respSize, &vi)) { // Parse it
61         printf("Received:\n");
62         if (vi.isResponse)

```

```

63     printf(" Response to ");
64     if (vi.isInquiry)
65         printf("inquiry ");
66     else
67         printf("vote ");
68     printf("for candidate %d\n", vi.candidate);
69     if (vi.isResponse)
70         printf(" count = %llu\n", vi.count);
71 }
72
73 // Close up
74 fclose(str);
75
76 exit(0);
77 }

```

VoteClientTCP.c

1. **Access to library functions and constants:** lines 1-14
2. **Argument processing:** lines 17-27
3. **Get connected socket:** lines 30-32
4. **Wrap the socket in a stream:** lines 34-36
We open a stream using `fdopen()` (mode “r+” opens for both reading and writing).
5. **Prepare and send request message:** lines 38-55
 - **Prepare a `VoteInfo` structure with the candidate ID:** lines 39-43
 - **Encode into wire format:** line 47
 - **Print encoded message before framing:** lines 50-51
 - **Add framing and send over the output stream:** lines 54-55
6. **Receive, parse, and process response:** lines 58-71
 - **Call `GetNextMsg()`:** line 59
The method handles all the messy details of receiving enough data through the socket to make up the next message; like `recv()`, it may block indefinitely.
 - **Pass the result to be parsed:** line 60
 - **Process the response if it is correctly formed:** lines 61-70
Invalid responses are ignored.
7. **Close the stream:** line 74

Turning now to the server, it needs a way to keep track of the vote counts of all the candidates. Because there are at most 1001 candidates, an array of 64-bit integers will serve nicely. The server prepares its socket and waits for incoming connections like the other servers we have seen. When a connection arrives, our program receives and processes messages over

that connection until the client closes it. Note that, because of the very basic interface to the framing/parsing code, our client and server are quite simple-minded when it comes to handling errors in received messages; the server simply ignores any message that is malformed and immediately closes the connection.

VoteServerTCP.c

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdbool.h>
4  #include <stdint.h>
5  #include <unistd.h>
6  #include <errno.h>
7  #include <sys/socket.h>
8  #include <arpa/inet.h>
9  #include "Practical.h"
10 #include "VoteProtocol.h"
11 #include "VoteEncoding.h"
12 #include "Framer.h"
13
14 static uint64_t counts[MAX_CANDIDATE + 1];
15
16 int main(int argc, char *argv[]) {
17     if (argc != 2) // Test for correct number of arguments
18         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
19
20     int servSock = SetupTCPServerSocket(argv[1]);
21     // servSock is now ready to use to accept connections
22
23     for (;;) { // Run forever
24
25         // Wait for a client to connect
26         int clntSock = AcceptTCPConnection(servSock);
27
28         // Create an input stream from the socket
29         FILE *channel = fdopen(clntSock, "r+");
30         if (channel == NULL)
31             DieWithSystemMessage("fdopen() failed");
32
33         // Receive messages until connection closes
34         int mSize;
35         uint8_t inBuf[MAX_WIRE_SIZE];
36         VoteInfo v;
37         while ((mSize = GetNextMsg(channel, inBuf, MAX_WIRE_SIZE)) > 0) {
38             memset(&v, 0, sizeof(v)); // Clear vote information

```

```

39     printf("Received message (%d bytes)\n", mSize);
40     if (Decode(inBuf, mSize, &v)) { // Parse to get VoteInfo
41         if (!v.isResponse) { // Ignore non-requests
42             v.isResponse = true;
43             if (v.candidate >= 0 && v.candidate <= MAX_CANDIDATE) {
44                 if (!v.isInquiry)
45                     counts[v.candidate] += 1;
46                 v.count = counts[v.candidate];
47             } // Ignore invalid candidates
48         }
49         uint8_t outBuf[MAX_WIRE_SIZE];
50         mSize = Encode(&v, outBuf, MAX_WIRE_SIZE);
51         if (PutMsg(outBuf, mSize, channel) < 0) {
52             fputs("Error framing/outputting message\n", stderr);
53             break;
54         } else {
55             printf("Processed %s for candidate %d; current count is %llu.\n",
56                 (v.isInquiry ? "inquiry" : "vote"), v.candidate, v.count);
57         }
58         fflush(channel);
59     } else {
60         fputs("Parse error, closing connection.\n", stderr);
61         break;
62     }
63 }
64 puts("Client finished");
65 fclose(channel);
66 } // Each client
67 // NOT REACHED
68 }

```

VoteServerTCP.c

1. **Access to library functions and constants:** lines 1–12
2. **Local declarations:** line 14
Array to store vote counts.
3. **Declarations and argument processing:** lines 16–18
4. **Set up listening socket:** line 20
5. **Repeatedly accept and handle clients:** lines 23–66
 - **Wait for connection:** line 26
AcceptTCPConnection() prints client info.
 - **Wrap the socket in a stream:** lines 29–31
 - **Receive and process messages until connection closes:** lines 34–63

Note that the server uses the same code as the client to parse, frame, and encode messages.

6. **Close client connection:** line 65

5.2.1 Framing

Application protocols typically deal with discrete messages, which are viewed as collections of fields. *Framing* refers to the general problem of enabling the receiver to locate the boundaries of a message (or part of one). Whether information is encoded as text, as multibyte binary numbers, or as some combination of the two, the application protocol must specify how the receiver of a message can determine when it has received all of the message.

Of course, if a complete message is sent as the payload of a UDP datagram, the problem is trivial: every send/receive operation on a datagram socket involves a single message, so the receiver knows exactly where that message ends. For messages sent over TCP sockets, however, the situation can be more complicated, because TCP has no notion of message boundaries. If the fields in a message all have fixed sizes and the message is made up of a fixed number of fields, then the size of the message is known in advance and the receiver can simply read the expected number of bytes into a buffer. (This technique was used in `TCPEchoClient.c`, where we knew the number of bytes to expect from the server.) However, when the message can vary in length—for example, if it contains some variable-length arbitrary text strings—we do not know beforehand how many bytes to read.

If a receiver tries to receive more bytes from a socket than were in the message, one of two things can happen. If no other message is in the channel, the receiver will block and will be prevented from processing the message; if the sender is also blocked waiting for a reply, the result will be *deadlock*: each side of the connection waiting for the other to send more information. On the other hand, if another message is already in the channel, the receiver may read some or all of it as part of the first message, leading to other kinds of errors. Therefore framing is an important consideration when using TCP sockets.

Note that some of the same considerations apply to finding the boundaries of the individual *fields* of the message: the receiver needs to know where one ends and another begins. Thus, pretty much everything we say here about framing messages also applies to fields. However, as was pointed out above, the cleanest code results if we deal with the problem of locating the end of the message separately from that of parsing it into fields.

Two general techniques enable a receiver to unambiguously find the end of the message:

- *Delimiter-based*: The end of the message is indicated by a *unique marker*, a particular, agreed-upon byte (or sequence of bytes) that the sender transmits immediately following the data.
- *Explicit length*: The variable-length field or message is preceded by a *length* field that tells how many bytes it contains. The length field is generally of a fixed size; this limits the maximum size message that can be framed.

A special case of the delimiter-based method can be used for the last message sent on a TCP connection: the sender simply closes the sending side of the connection after sending the message. After the receiver reads the last byte of the message, it receives an end-of-stream indication (i.e., `recv()` returns 0 or `fread()` returns EOF) and thus can tell that it has reached the end of the message.

The delimiter-based approach is often used with messages encoded as text: A particular character or sequence of characters is defined to mark the end of the message. The receiver simply scans the input (as characters) looking for the delimiter sequence; it returns the character string preceding the delimiter. The drawback is that *the message itself must not contain the delimiter*; otherwise the receiver will find the end of the message prematurely. With a delimiter-based framing method, someone has to be responsible for ensuring that this precondition is satisfied. Fortunately, so-called *stuffing* techniques allow delimiters that occur naturally in the message to be modified so the receiver will not recognize them as such. The sending side performs a transformation on delimiters that occur in the text; the receiver, as it scans for the delimiter, also recognizes the transformed delimiters and restores them so that the output message matches the original. The downside of such techniques is that *both* sender and receiver have to scan every byte of the message.

The length-based approach is simpler but requires a known upper bound on the size of the message. The sender first determines the length of the message, encodes it as an integer, and prefixes the result to the message. The upper bound on the message length determines the number of bytes required to encode the length: 1 byte if messages always contain fewer than 256 bytes, 2 bytes if they are always shorter than 65536 bytes, and so on.

The module `DelimFramer.c` implements delimiter-based framing using the “newline” character (“\n”, byte value 10) as the delimiter. Our `PutMsg()` method does *not* do stuffing, but simply fails (catastrophically) if the byte sequence to be framed already contains the delimiter. The `GetNextMsg()` method scans the stream, copying each byte into the buffer until it reads the delimiter or runs out of space. It returns the number of bytes placed in the buffer. If the message is truncated, that is, the method returns a full buffer without encountering a delimiter, the returned count is negative. If some bytes of a message are accumulated and the stream ends without finding a delimiter, it is considered an error and a negative value is returned (i.e., this protocol does *not* accept end-of-stream as a delimiter). Thus, an empty but correctly framed message (length zero is returned) can be distinguished from the stream ending.

DelimFramer.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include "Practical.h"
5
6 static const char DELIMITER = '\n';
7
8 /* Read up to bufSize bytes or until delimiter, copying into the given
9  * buffer as we go.
```

```

10  * Encountering EOF after some data but before delimiter results in failure.
11  * (That is: EOF is not a valid delimiter.)
12  * Returns the number of bytes placed in buf (delimiter NOT transferred).
13  * If buffer fills without encountering delimiter, negative count is returned.
14  * If stream ends before first byte, -1 is returned.
15  * Precondition: buf has room for at least bufSize bytes.
16  */
17  int GetNextMsg(FILE *in, uint8_t *buf, size_t bufSize) {
18      int count = 0;
19      int nextChar;
20      while (count < bufSize) {
21          nextChar = getc(in);
22          if (nextChar == EOF) {
23              if (count > 0)
24                  DieWithUserMessage("GetNextMsg()", "Stream ended prematurely");
25              else
26                  return -1;
27          }
28          if (nextChar == DELIMITER)
29              break;
30          buf[count++] = nextChar;
31      }
32      if (nextChar != DELIMITER) { // Out of space: count==bufSize
33          return -count;
34      } else { // Found delimiter
35          return count;
36      }
37  }
38
39  /* Write the given message to the output stream, followed by
40   * the delimiter. Return number of bytes written, or -1 on failure.
41   */
42  int PutMsg(uint8_t buf[], size_t msgSize, FILE *out) {
43      // Check for delimiter in message
44      int i;
45      for (i = 0; i < msgSize; i++)
46          if (buf[i] == DELIMITER)
47              return -1;
48      if (fwrite(buf, 1, msgSize, out) != msgSize)
49          return -1;
50      fputc(DELIMITER, out);
51      fflush(out);
52      return msgSize;
53  }

```

1. **Declare constant delimiter value:** line 6
2. **Input method:** `GetNextMsg()`: lines 17–37
 - **Initialize byte count:** line 18
 - **Iterate until buffer is full or EOF:** lines 20–37

We get the next byte from the input stream, compare it to EOF, then the delimiter. On EOF we abort if an incomplete message is in the buffer, else return `-1`. On delimiter we break out of the loop. If the buffer becomes full (`count==bufSize`), we return the negation of the number of bytes read as an indication that the channel is not empty.
 - **Return the count of bytes transferred to buffer:** line 35
3. **Output method:** `PutMsg()`: lines 42–53
 - **Scan the input message looking for the delimiter:** lines 43–47

If we find it, (we rather unhelpfully) kill the program.
 - **Write message to output stream:** line 48
 - **Write delimiter byte to output stream:** line 50
 - **Flush the output stream:** line 47

This ensures that the message is sent over the underlying socket.

Although our implementation makes it fairly easy to change the single character used as a delimiter, some protocols make use of multiple-character delimiters. The HTTP protocol, which is used in the World Wide Web, uses text-encoded messages delimited by the four-character sequence `\r\n\r\n`. Extending the delimiter-based framing module to support multicharacter delimiters and to handle stuffing is left as an exercise.

The module `LengthFramer.c` implements the framing interface using length-based framing. It works for messages up to $65535 (2^{16} - 1)$ bytes in length. The `PutMsg()` method determines the length of the given message and writes it to the output stream as a 2-byte, big-endian integer, followed by the complete message. On the receiving side, the `fread()` method is used to read the length as an integer; after converting it to host byte order, that many bytes are read from the channel. Note that, with this framing method, the sender does not have to inspect the content of the message being framed; it needs only to check that the message does not exceed the length limit.

LengthFramer.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdint.h>
4 #include <netinet/in.h>
5 #include "Practical.h"
6
7 /* Read 2-byte length and place in big-endian order.
```

```

8  * Then read the indicated number of bytes.
9  * If the input buffer is too small for the data, truncate to fit and
10 * return the negation of the *indicated* length. Thus a negative return
11 * other than -1 indicates that the message was truncated.
12 * (Ambiguity is possible only if the caller passes an empty buffer.)
13 * Input stream is always left empty.
14 */
15 int GetNextMsg(FILE *in, uint8_t *buf, size_t bufSize) {
16     uint16_t mSize = 0;
17     uint16_t extra = 0;
18
19     if (fread(&mSize, sizeof(uint16_t), 1, in) != 1)
20         return -1;
21     mSize = ntohs(mSize);
22     if (mSize > bufSize) {
23         extra = mSize - bufSize;
24         mSize = bufSize; // Truncate
25     }
26     if (fread(buf, sizeof(uint8_t), mSize, in) != mSize) {
27         fprintf(stderr, "Framing error: expected %d, read less\n", mSize);
28         return -1;
29     }
30     if (extra > 0) { // Message was truncated
31         uint8_t waste[BUFSIZE];
32         fread(waste, sizeof(uint8_t), extra, in); // Try to flush the channel
33         return -(mSize + extra); // Negation of indicated size
34     } else
35         return mSize;
36 }
37
38 /* Write the given message to the output stream, followed by
39 * the delimiter. Precondition: buf[] is at least msgSize.
40 * Returns -1 on any error.
41 */
42 int PutMsg(uint8_t buf[], size_t msgSize, FILE *out) {
43     if (msgSize > UINT16_MAX)
44         return -1;
45     uint16_t payloadSize = htons(msgSize);
46     if ((fwrite(&payloadSize, sizeof(uint16_t), 1, out) != 1) || (fwrite(buf,
47         sizeof(uint8_t), msgSize, out) != msgSize))
48         return -1;
49     fflush(out);
50     return msgSize;
51 }

```

1. **Input method:** `GetNextMsg()`: lines 15–36
 - **Read the prefix length:** lines 19–20
The `fread()` method reads 2 bytes into the unsigned 16-bit integer *mSize*.
 - **Convert to host byte order:** line 21
 - **Truncate message if necessary:** lines 22–25
If the indicated size is bigger than the buffer provided by the caller, we truncate the message so it will fit, and remember so we can indicate via the return value that we did so.
 - **Read the message:** line 26
 - **Flush channel:** lines 30–34
If we are returning early because the buffer is full, we remove the extra bytes from the channel and return the negation of the header size.
 - **Return the size:** line 35
2. **Output method:** `PutMsg()`: lines 42–51
 - **Verify input length:** lines 43–44
Because we use a 2-byte unsigned length field, the length cannot exceed 65535 (the value of `UINT16_MAX`).
 - **Convert length to network byte order:** line 45
 - **Output length and message:** lines 46–48
 - **Flush to ensure the message is sent:** line 49

5.2.2 Text-Based Message Encoding

Now we turn to the representation of voting messages as text. Because we only have to represent numbers and a couple of indicators, we can use the basic C charset US-ASCII. The message consists of text fields separated by one or more occurrences of the ASCII space character (decimal value 32). The message begins with a so-called magic string—a sequence of ASCII characters that allows a recipient to quickly recognize the message as a Voting protocol message, as opposed to random garbage that happened to arrive over the network. The Vote/Inquiry boolean is encoded with the character “v” for a vote or “i” for an inquiry. The message’s status as a response is indicated by the presence of the character “R.” Then comes the candidate ID, followed by the vote count, both encoded as strings of decimal digits. The module `VoteEncodingText.c` implements this text-based encoding.

VoteEncodingText.c

```

1 /* Routines for Text encoding of vote messages.
2  * Wire Format:
3  *   "Voting <v|i> [R]  <candidate ID>  <count>"
4  */

```

```

5  #include <string.h>
6  #include <stdint.h>
7  #include <stdbool.h>
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <string.h>
11 #include "Practical.h"
12 #include "VoteProtocol.h"
13
14 static const char *MAGIC = "Voting";
15 static const char *VOTESTR = "v";
16 static const char *INQSTR = "i";
17 static const char *RESPONSESTR = "R";
18 static const char *DELIMSTR = " ";
19 enum {
20     BASE = 10
21 };
22
23 /* Encode voting message info as a text string.
24  * WARNING: Message will be silently truncated if buffer is too small!
25  * Invariants (e.g. 0 <= candidate <= 1000) not checked.
26  */
27 size_t Encode(const VoteInfo *v, uint8_t *outBuf, const size_t bufSize) {
28     uint8_t *bufPtr = outBuf;
29     long size = (size_t) bufSize;
30     int rv = snprintf((char *) bufPtr, size, "%s %c %s %d", MAGIC,
31         (v->isInquiry ? 'i' : 'v'), (v->isResponse ? "R" : ""), v->candidate);
32     bufPtr += rv;
33     size -= rv;
34     if (v->isResponse) {
35         rv = snprintf((char *) bufPtr, size, " %llu", v->count);
36         bufPtr += rv;
37     }
38     return (size_t) (bufPtr - outBuf);
39 }
40
41 /* Extract message information from given buffer.
42  * Note: modifies input buffer.
43  */
44 bool Decode(uint8_t *inBuf, const size_t mSize, VoteInfo *v) {
45
46     char *token;
47     token = strtok((char *) inBuf, DELIMSTR);
48     // Check for magic
49     if (token == NULL || strcmp(token, MAGIC) != 0)

```

```

50     return false;
51
52     // Get vote/inquiry indicator
53     token = strtok(NULL, DELIMSTR);
54     if (token == NULL)
55         return false;
56
57     if (strcmp(token, VOTESTR) == 0)
58         v->isInquiry = false;
59     else if (strcmp(token, INQSTR) == 0)
60         v->isInquiry = true;
61     else
62         return false;
63
64     // Next token is either Response flag or candidate ID
65     token = strtok(NULL, DELIMSTR);
66     if (token == NULL)
67         return false; // Message too short
68
69     if (strcmp(token, RESPONSESTR) == 0) { // Response flag present
70         v->isResponse = true;
71         token = strtok(NULL, DELIMSTR); // Get candidate ID
72         if (token == NULL)
73             return false;
74     } else { // No response flag; token is candidate ID;
75         v->isResponse = false;
76     }
77     // Get candidate #
78     v->candidate = atoi(token);
79     if (v->isResponse) { // Response message should contain a count value
80         token = strtok(NULL, DELIMSTR);
81         if (token == NULL)
82             return false;
83         v->count = strtoll(token, NULL, BASE);
84     } else {
85         v->count = 0L;
86     }
87     return true;
88 }

```

VoteEncodingText.c

The `Encode()` method uses `sprintf()` to construct a string containing all the fields of the message, separated by white space. It fails only if the caller provides an insufficient amount of space to hold the string.

The `Decode()` method uses the `strtok()` method to break the received message into tokens (fields). The `strtok()` library function takes a pointer to a character array and a string containing characters to be interpreted as delimiters. The first time it is called, it returns the largest initial substring consisting entirely of characters not in the delimiter string; the trailing delimiter of that string is replaced by a null byte. On subsequent calls with a NULL first argument, tokens are taken left to right from the original string until there are no more tokens, at which point NULL is returned.



`Decode()` first looks for the “Magic” string; if it is not the first thing in the message, it simply fails and returns FALSE. **Note well:** This illustrates a very important point about implementing protocols: **never assume anything about any input from the network.** Your program must always be prepared for any possible inputs, and handle them gracefully. In this case, the `Decode()` method simply ignores the rest of the message and returns FALSE if some expected part is not present or improperly formatted.⁴ Otherwise, `Decode()` gets the fields token by token, using the library functions `atoi()` and `strtol()` to convert tokens into integers.

5.2.3 Binary Message Encoding

Next we present a different way to encode the Voting protocol message. In contrast with the text-based format, the binary format uses fixed-size messages. Each message begins with a one-byte field that contains the “magic” value 010101 in its high-order 6 bits. As with the text format, this little bit of redundancy provides the receiver with a small degree of assurance that it is receiving a proper voting message. The two low-order bits of the first byte encode the two booleans; note the use of the bitwise-or operations shown earlier, in Section 5.1.8, to set the flags. The second byte of the message always contains zeros (it is effectively padding), and the third and fourth bytes contain the candidateID. The final 8 bytes of a response message (only) contain the vote count.

VoteEncodingBin.c

```

1  /* Routines for binary encoding of vote messages
2   * Wire Format:
3   *
4   *      1 1 1 1 1 1
5   * 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
6   * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
7   * |      Magic      |Flags|      ZERO      |
8   * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
9   * |                  Candidate ID          |
10  * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+
11  * |
```

⁴This also illustrates a key reason why it is best to do framing before parsing: recovery from a parse error when a message has only been partially received is much more complex because the receiver has to get “back in sync” with the sender.


```

11  * |          Vote Count (only in response)          |
12  * | |
13  * | |
14  * +---+---+---+---+---+---+---+---+---+---+---+---+
15  *
16  */
17
18 #include <string.h>
19 #include <stdbool.h>
20 #include <stdlib.h>
21 #include <stdint.h>
22 #include <netinet/in.h>
23 #include "Practical.h"
24 #include "VoteProtocol.h"
25
26 enum {
27     REQUEST_SIZE = 4,
28     RESPONSE_SIZE = 12,
29     COUNT_SHIFT = 32,
30     INQUIRE_FLAG = 0x0100,
31     RESPONSE_FLAG = 0x0200,
32     MAGIC = 0x5400,
33     MAGIC_MASK = 0xfc00
34 };
35
36 typedef struct voteMsgBin voteMsgBin;
37
38 struct voteMsgBin {
39     uint16_t header;
40     uint16_t candidateID;
41     uint32_t countHigh;
42     uint32_t countLow;
43 };
44
45 size_t Encode(VoteInfo *v, uint8_t *outBuf, size_t bufSize) {
46     if ((v->isResponse && bufSize < sizeof(voteMsgBin)) || bufSize < 2
47         * sizeof(uint16_t))
48         DieWithUserMessage("Output buffer too small", "");
49     voteMsgBin *vm = (voteMsgBin *) outBuf;
50     memset(outBuf, 0, sizeof(voteMsgBin)); // Be sure
51     vm->header = MAGIC;
52     if (v->isInquiry)
53         vm->header |= INQUIRE_FLAG;
54     if (v->isResponse)
55         vm->header |= RESPONSE_FLAG;

```

```

56  vm->header = htons(vm->header); // Byte order
57  vm->candidateID = htons(v->candidate); // Know it will fit, by invariants
58  if (v->isResponse) {
59      vm->countHigh = htonl(v->count >> COUNT_SHIFT);
60      vm->countLow = htonl((uint32_t) v->count);
61      return RESPONSE_SIZE;
62  } else {
63      return REQUEST_SIZE;
64  }
65 }
66
67 /* Extract message info from given buffer.
68  * Leave input unchanged.
69  */
70 bool Decode(uint8_t *inBuf, size_t mSize, VoteInfo *v) {
71
72     voteMsgBin *vm = (voteMsgBin *) inBuf;
73
74     // Attend to byte order; leave input unchanged
75     uint16_t header = ntohs(vm->header);
76     if ((mSize < REQUEST_SIZE) || ((header & MAGIC_MASK) != MAGIC))
77         return false;
78     /* message is big enough and includes correct magic number */
79     v->isResponse = ((header & RESPONSE_FLAG) != 0);
80     v->isInquiry = ((header & INQUIRE_FLAG) != 0);
81     v->candidate = ntohs(vm->candidateID);
82     if (v->isResponse && mSize >= RESPONSE_SIZE) {
83         v->count = ((uint64_t) ntohl(vm->countHigh) << COUNT_SHIFT)
84             | (uint64_t) ntohl(vm->countLow);
85     }
86     return true;
87 }

```

VoteEncodingBin.c

The Decode() method's job is especially simple in this version—it simply copies the values from the message into the **VoteInfo** structure, converting byte order along the way.

5.2.4 Putting It All Together

To get a working vote server, we simply compile together VoteServerTCP.c, one of the two framing modules, one of the two encoding modules, and the auxiliary modules DieWithMessage.c, TCPClientUtility.c, TCPServerUtility.c, and AddressUtility.c. For example:

```

% gcc -std=gnu99-o vs VoteServerTCP.c DelimFramer.c VoteEncodingBin.c \
  DieWithMessage.c TCPServerUtility.c AddressUtility.c

```

```
% gcc -std=gnu99-o vc VoteClientTCP.c DelimFramer.c VoteEncodingBin.c \
DieWithMessage.c TCPClientUtility.c
```

All four possible combinations of framing method and encoding will work—provided the client and server use the same combination!

5.3 Wrapping Up

We have seen how primitive types can be represented as sequences of bytes for transmission “on the wire.” We have also considered some of the subtleties of encoding text strings, as well as some basic methods of framing and parsing messages. We saw examples of both text-oriented and binary-encoded protocols.

It is probably worth reiterating something we said in the Preface: this chapter will by no means make you an expert! That takes a great deal of experience. But the code from this chapter can be used as a starting point for further explorations.

Exercises

1. If the underlying hardware platform is little-endian, and the “network” byte order is big-endian, is there any reason for the implementations of `htonl()` and `ntohl()` to be different?
2. Write the function `uint64_t htonl(uint64_t val)`, which converts a 64-bit integer from little-endian to big-endian byte order.
3. Write the little-endian analogs of the method given in `BruteForceEncoding.c`, that is, `EncodeLittleEndian()` and `DecodeLittleEndian()`.
4. Write methods `EncodeBigEndianSigned()` and `DecodeBigEndianSigned()`, which return signed values. (The input buffers are still unsigned types. *Hint*: use explicit type-casting.)
5. The `EncodeIntBigEndian()` method in `BruteForceEncoding.c` only works if several preconditions such as $0 \leq \text{size} \leq 8$ are satisfied. Modify the method to test for these preconditions and return an error indication of some sort if any are violated. What are the advantages and disadvantages of having the program check the preconditions, versus relying on the caller to establish them?
6. Assuming all byte values are equally likely, what is the probability that a message consisting of random bits will pass the “magic test” in the binary encoding of the Voting protocol? Suppose an ASCII-encoded text message is sent to a program expecting a binary-encoded `voteMsg`. Which characters would enable the message to pass the “magic test” if they are the first in the message?
7. Suppose we use `DelimFraming.c` and `VoteEncodingBin.c` in building the Voting Client. Describe circumstances in which the client fails to send a message.

8. Extend the delimiter-based framing implementation to perform “byte stuffing,” so that messages that contain the delimiter can be transmitted without the caller of `PutMsg()` having to worry about it. That is, the framing module transparently handles messages that contain the delimiter. (See any decent networking text for the algorithm.)
9. Extend the delimiter-based framing implementation to handle arbitrary multiple-byte delimiters. Be sure your implementation is efficient. (**Note:** this problem is *not* trivial! A naive approach will run very inefficiently in the worst case.)
10. Both `GetNextMsg()` implementations truncate the received message if the caller fails to provide a big enough buffer. Consider the behavior on the next call to `GetNextMsg()` after this happens, for both implementations. Is the behavior the same in both cases? If not, suggest modifications so that both implementations behave the same way in all cases.