Typically you use sockets because your program needs to provide information to, or use information provided by, another program. Because in most real applications the behavior of clients and servers depends upon 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.

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.

**Encoding Information**

**Primitive Integers**

As we have already seen, TCP and UDP sockets give us the ability to send and receive sequences (arrays) of bytes, i.e., integer values in the range 0–255.

Let’s consider how we would encode a sequence of four integer values: a byte, a short, an int, and a long, in that order, for transmission from sender to receiver. We need a total of 15 bytes: the first contains the value of the byte, the next two contain the value of the short, the next four encode the value of the int, and the last eight bytes contain the long value

Are we ready to go? Not quite. For types that require more than one byte, we 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 integer, with the least significant byte—so-called little-endian order—or at the left end, with the most significant byte—big-endian order.

A k-bit, unsigned integer can encode values in the range 0 through 2k − 1 directly.

So how do we get the correct values into the byte array of the message? To allow you to see exactly what needs to happen, here’s how to do the encoding explicitly using “bitdiddling” (shifting and masking) operations.

**String and Text**

Old-fashioned text—strings of printable (displayable) characters—is perhaps the most common way to represent information.

Text is made up of symbols or characters. In fact every String instance corresponds to a sequence (array) of characters (type char[ ]). A char value in Java is represented internally as an integer

A mapping between a set of symbols and a set of integers is called a coded character set.

Sender and receiver need to agree on how those integers will be represented as byte sequences—that is, an encoding scheme.

The moral of the story is that sender and receiver must agree on the representation for strings of text. The easiest way for them to do that is to simply specify one of the standard charsets.

**Bit-Diddling: Encoding Boolean**

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.

**Composing I/O Streams**

Java’s stream classes can be composed to provide powerful capabilities.

**Framing and Parsing**

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.

Framing is an important consideration when using TCP sockets.

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, an explicit byte

sequence that the sender transmits immediately following the data. The marker must be

known not to occur in the data.

Explicit length: The variable-length field or message is preceded by a (fixed-size) length field that tells how many bytes it contains.

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 (using shutdownOutput() or close()) after sending the message. After the receiver reads the last byte of the message, it receives an end-of-stream indication (i.e., read() returns −1), 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 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, the sender is responsible for ensuring that this precondition is satisfied. The downside of such techniques is that both sender and receiver have to scan the message.

**Java-Specific Encoding**

RMI lets you invoke methods on different Java virtual machines, hiding all the messy details of argument encoding and decoding. Serialization handles conversion of actual Java objects to byte sequences for you, so you can transfer actual instances of Java objects between virtual machines.

These capabilities might seem like communication Nirvana, but in reality they are not always the best solution, for several reasons.

**Constructing and Parsing Protocol Messages**

In implementing a protocol, it is helpful to define a class to contain the information carried in a message. The class provides methods for manipulating the fields of the message—while maintaining the invariants that are supposed to hold among those fields.

**Text-Based Representation**

The protocol specifies that the text be encoded using the US-ASCII charset. The message begins with a so-called “magic string”—a sequence of 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.

**Binary Representation**

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 six bits. The two low-order bits of the first byte encode the two booleans. The second byte of the message always contains zeros, and the third and fourth bytes contain the candidateID. The final eight bytes of a response message (only) contain the vote count.

**Sending and Receiving**

Sending a message over a stream is as simple as creating it, calling toWire(), adding appropriate framing information, and writing it. Receiving, of course, does things in the opposite order. This approach applies to TCP; in UDP explicit framing is not necessary, because message boundaries are preserved.