3. Sending and Receiving Data

3.1 Encoding Information

3.1.1 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. Using that ability, we can encode the values of other (larger) primitive integer types. However, the sender and receiver have to agree on several things first. One is the size (in bytes) of each integer to be transmitted. For example, an int value in a Java program is represented as a 32-bit quantity. We can therefore transmit the value of any variable or constant of type int using four bytes. Values of type short, on the other hand, are represented using 16 bits and so only require two bytes to transmit, while longs are 64 bits or eight bytes.

3.1.2 Strings and Text

- Old-fashioned text—strings of printable (displayable) characters—is perhaps the most common way to represent information. Text is convenient because humans are accustomed to dealing with all kinds of information represented as strings of characters in books, newspapers, and on computer displays. Thus, once we know how to encode text for transmission, we can send almost any other kind of data: first represent it as text, then encode the text.

3.1.3 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 Java'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 int-sized bitmaps and masks (32 bits), but everything we say applies to other integer types as well.

3.2 Composing I/O Streams

- Java's stream classes can be composed to provide powerful capabilities. For example, we can wrap the OutputStream of a Socket instance in a BufferedOutputStream instance to improve performance by buffering bytes temporarily and flushing them to the underlying channel all at once. We can then wrap that instance in a DataOutputStream to send primitive data types.

I/O Class	Function
Buffered[Input/Output]Stream	Performs buffering for I/O optimization.
Checked[Input/Output]Stream	Maintains a checksum on data.
Cipher[Input/Output]Stream	Encrypt/Decrypt data.
Data[Input/Output]Stream	Handles read/write for primitive date types.
Digest[Input/Output]Stream	Maintains a digest on data.
GZIP[Input/Output]Stream	De/compresses a byte stream in GZIP format.
Object[Input/Output]Stream	Handles read/write objects and primitive data types.
PushbackInputStream	Allows a byte or bytes to be "unread."
PrintOutputStream	Prints string representation of data type.
Zip[Input/Output]Stream	De/compresses a byte stream in ZIP format.

3.3 Framing and Parsing

- Framing refers to the problem of enabling the receiver to locate the beginning and end of a message. 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.
- 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.

3.4 Java-Specific Encodings

- When you use sockets, generally either you are building the programs on both ends of the communication channel—in which case you also have complete control over the protocol—or you are communicating using a given protocol

3.5 Constructing and Parsing Protocol Messages

- To illustrate the different methods of encoding information, we present two implementations of VoteMsgCoder, one using a text-based encoding and one using a binary encoding. If you were guaranteed a single encoding that would never change, the toWire() and fromWire() methods could be specified as part of VoteMsg. Our purpose here is to emphasize that the abstract representation is independent of the details of the encoding.

3.5.1 Text-Based Representation

- Your program must always be prepared for any possible inputs, and handle them gracefully. In this case, the fromWire() method throws an exception if the expected string is not present. Otherwise, it gets the fields token by token, using the Scanner instance. Note that the number of fields in the message depends on whether it is a request (sent by the client) or response (sent by the server). fromWire() throws an exception if the input ends prematurely or is otherwise malformed.

3.5.2 Binary Representation

- 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. To demonstrate this, consider a vote server that

1) maintains a mapping of candidate IDs to number of votes,

2) counts submitted votes, and

3) responds to inquiries and votes with the current count for the specified candidate.