**Chapter 2: Basic sockets**

2.1 Socket Addresses

* A client must specify the IP address of the host running the server program when it initiates communication. The network infrastructure then uses this destination address to route the client’s information to the proper machine.

2.2 TCP Sockets

2.2.1 TCP Client

* The client initiates communication with a server that is passively waiting to be contacted. The typical TCP client goes through three steps:

1. Construct an instance of Socket: The constructor establishes a TCP connection to the specified remote host and port.

2. Communicate using the socket’s I/O streams: A connected instance of Socket contains an InputStream and OutputStream that can be used just like any other Java I/O stream.

3. Close the connection using the close() method of Socket.

2.2.2 TCP Server

* The server’s job is to set up a communication endpoint and passively wait for connections fromc lients. The typical TCP server goes through two steps:

1. Construct a ServerSocket instance, specifying the local port. This socket listens for incoming connections to the specified port.

2. Repeatedly:

a. Call the accept() method of ServerSocket to get the next incoming client connection. Upon establishment of a new client connection, an instance of Socket for the new connection is created and returned by accept().

b. Communicate with the client using the returned Socket’s InputStream and OutputStream.

c. When finished, close the new client socket connection using the close() method of Socket.

2.2.3 Input and Output Streams

* The basic I/O paradigm for TCP sockets in Java is the stream abstraction.
* A stream is simply an ordered sequence of bytes. Java input streams support reading bytes, and output streams support writing bytes. In our TCP client and server, each Socket instance holds an InputStream and an OutputStream instance. When we write to the output stream of a Socket, the bytes can (eventually) be read from the input stream of the Socket at the other end of the connection.
* OutputStream is the abstract superclass of all output streams in Java. Using an OutputStream, we can write bytes to, flush, and close the output stream.
* InputStream is the abstract superclass of all input streams. Using an InputStream, we can read bytes from and close the input stream.

2.3 UDP Sockets

Provides an end-to-end server different from that of TCP. UDP performs only two functions:

1. Adds another layer of addressing (ports) to that of IP
2. Detects some forms of data corruption that may occur in transit and discards any corrupted messages.

Characteristics of UDP sockets:

* UDP sockets do not have to be connected before being used.
* UDP is analogous to communicating by mail
* Each message—called a datagram—carries its own address information and is independent of all others
* UDP socket is like a mailbox into which letters or packages from many different sources can be placed
* UDP sockets preserve message boundaries.
* There is no guarantee that a message sent via a UDP socket will arrive at its destination
* Messages can be delivered in a different order than they were sent

Why would an application use UDP instead of TCP?

* Efficiency: if the application exchanges only a small amount of data
* Flexibility: when something other than a reliable byte-stream service is required, UDP provides a minimal-overhead platform on which to implement whatever is needed.

2.3.1 DatagramPacket

UDP endpoints exchange self-contained messages, called datagrams, which are represented in Java as instances of DatagramPacket

2.3.2 UDP Client

A UDP client begins by sending a datagram to a server that is passively waiting to be contacted. The typical UDP client goes through three steps:

1. Construct an instance of DatagramSocket, optionally specifying the local address and port.
2. Communicate by sending and receiving instances of DatagramPacket using the send() and receive() methods of DatagramSocket.
3. When finished, deallocate the socket using the close() method of DatagramSocket.

2.3.3 UDP Client

The typical UDP server goes through three steps:

1. Construct an instance of DatagramSocket, specifying the local port and, optionally, the local address. The server is now ready to receive datagrams from any client.
2. Receive an instance of DatagramPacket using the receive() method of DatagramSocket. When receive() returns, the datagram contains the client’s address so we know where to send the reply
3. Communicate by sending and receiving DatagramPackets using the send() and receive() methods of DatagramSocket.

2.3.4 Sending and Receiving with UDP Sockets

Between the time a message arrives from the network and the time its data is returned via read() or receive(), the data is stored in a first-in, first-out (FIFO) queue of received data. With a connected TCP socket, all received-but-not-yet-delivered bytes are treated as one continuous sequence of bytes.

For a UDP socket, however, the received data may have come from different senders. A UDP socket’s received data is kept in a queue of messages, each with associated information identifying its source. A call to receive() will never return more than one message.

However, if receive() is called with a DatagramPacket containing a buffer of size n, and the size of the first message in the receive queue exceeds n, only the first n bytes of the message are returned. The remaining bytes are quietly discarded, with no indication to the receiving program that information has been lost