Chapter 2: Basic sockets

2.1. Socket Address:

- IP uses 32-bit binary addresses to identify communicating hosts.

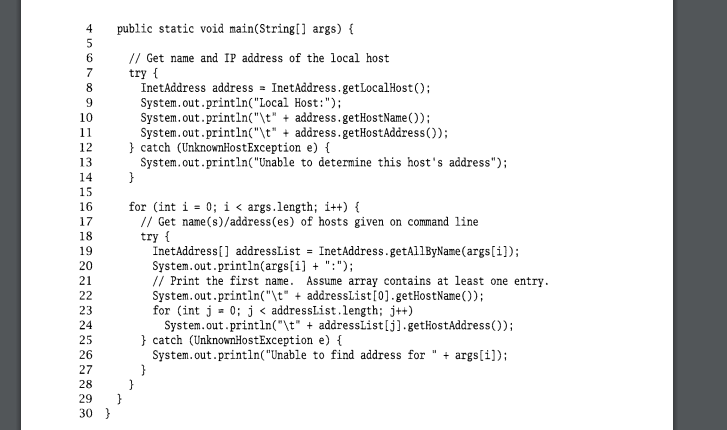
- A client must specify the IP address of the host running the server program when it initiates communication.

- Network infrastructure uses the 32-bit destination address to route the client's information.

- Addresses can be specified in Java using a string with dotted-quad (eg: 192.1.1.1) or name ( server.example.com)

- Java encapsulates the IP addresses abstraction in the InetAddress class.

- The program takes a list of names or IP addresses as commandline parameters and prints the name and an IP address of the local host, followed by names and IP addresses of the hosts specified on the command line.



InetAddress contains: Creators (line 4), Accessor (line 22), Operator.

2.2. TCP Sockets.

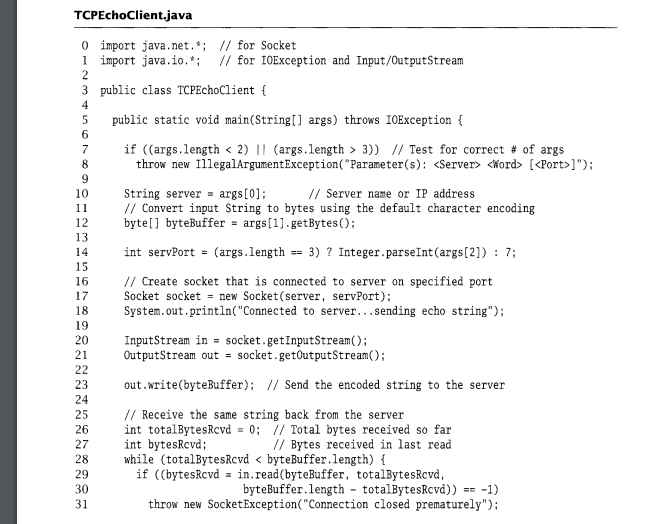
- Java provides two classes for TCP: Socket and ServerSocket. An instance of Socket represents one end of a TCP connection.

- A TCP connection is an abstract two-way channel, ends are each identified by an IP address and port number. The connection must go through a setup phase.

- Instance of ServerSocket listens for TCP connection requests and creates a new Socket instance to handle each incoming connection.

2.2.1. TCP client.

- The client initiates communication with a server that is passively waiting to be contacted. Contains 3 steps: Construct an instance of Socket, Communicate using the socket's I/O streams, Close the connection using the close() method of Socket.



- Application setup and parameter parsing: lines 0-14 (convert echo line 12, determine a port server for echo server line 14 (default port 7)).

- TCP socket creation: line 17

- Get socket input and output streams: lines 20-21.

- Send the string to echo server: line 23 ( write() method).

- Receive the reply from the echo server: lines 25-33. (This particular form of read() takes three parameters: 1) buffer to receive into, 2) byte offset into the buffer where the first byte received should be placed, and 3) the maximum number of bytes to be placed in the buffer).

- Print echoed string: line 35.

- Close socket: line 37.

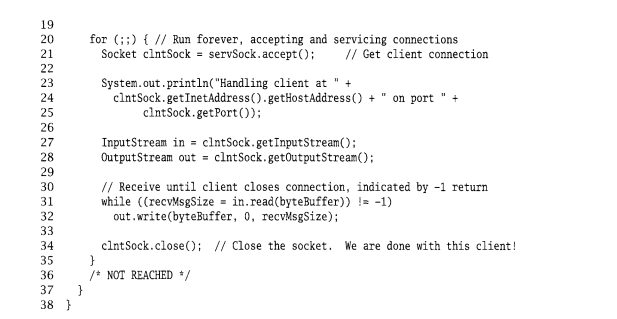
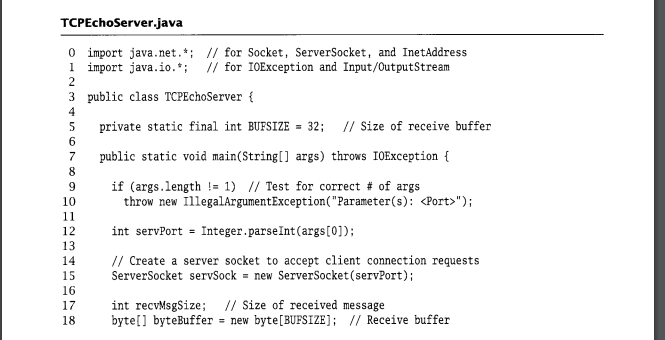
- Socket. Contains ( Constructors, Operators, Accessors/Mutators).

2.2.2. TCP Server:

- The server's job is to set up a communication endpoint and passively wait for connections from clients. The typical TCP server comprise two steps:

+ Construct a ServerSocket instance, specifying the local port.

+ Repeatedly: Call the accept () method of ServerSocket to get the next incoming client connection, Communicate with the client using the returned Socket's InputStream and OutputStream, Close the new client socket connection using the close() method of Socket.



- Application setup and parameter parsing: lines 0-12

- Server socket creation: line 15

- Loop forever, iteratively handling incoming connections: lines 20-35

+ Accept an incoming connection: line 21.

+ Report connected client: lines 23-25.

+ Get socket input and output streams: lines 27-28.

ServerSocket:

- Constructors: Construct a TCP socket that is ready to accept incoming connections to the specified local port.

+ Local port. A port of 0 allows the constructor to pick any available port.

+ Queue Limit: The maximum size of the queue of incomplete connections and sockets waiting to be accepted().

+ LocalAddr: The IP address to which connections to this socket should be addressed.

- Operator: Socket accept(): returns a connected Socket instance for the next new incoming connection to the server socket. void close(): Closes the underlying TCP socket.

- Accessors/Mutators: setSoTimeout(int timeout) Returns/sets the maximum amount of time (in milliseconds) that an accept() will block for this socket.

2.2.3. Input and Output Streams:

- OutputStream: abstract void write(int data), void write(byte[] data), void write(byte[ ] data, int offset, int length), void flush(), void close()

- InputStream : abstract int read(), int read(byte[] data), int read(byte[ ] data, int offset, int length), int available(), void close()

2.3) UDP Sockets

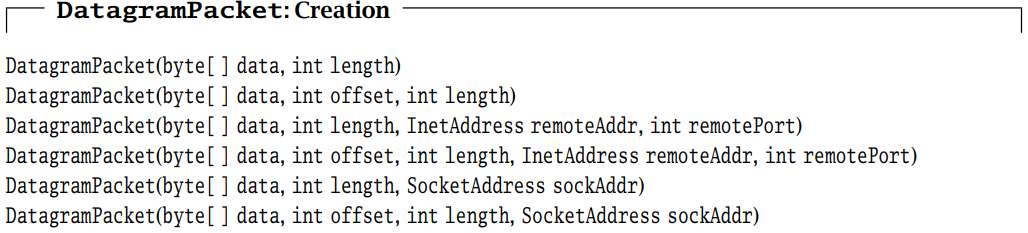
* Provides an end-to-end service different from that of TCP.
* Performs only two functions:
* It adds another layer of addressing (ports) to that of IP.
* It detects some forms of data corruption that may occur in transit and discards any corrupted messages.
* Another difference: UDP sockets preserve message boundaries
* A final difference is that the end-to-end transport service UDP provides is best-effort: there is no guarantee that a message sent via a UDP socket will arrive at its destination, and messages can be delivered in a different order than they were sent

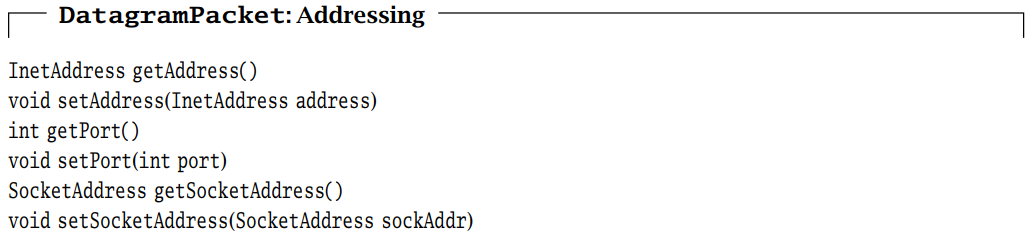
→ Be prepared to deal with loss and reordering.

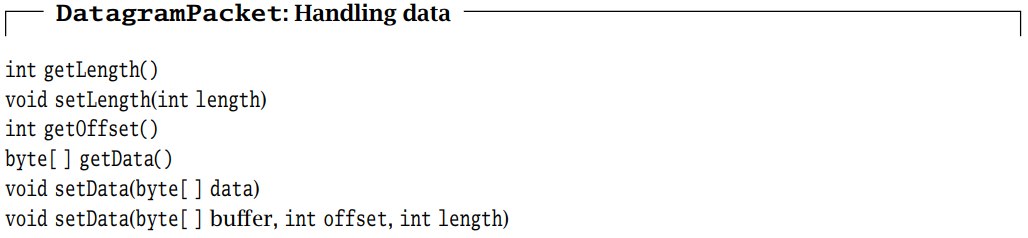
* Reasons to use UDP over TCP:
* Efficiency
* Flexibility

2.3.1) DatagramPacket

* UDP endpoints exchange self-contained messages → Datagram
* To send: constructs a DatagramPacket instance containing the data to be sent and passes it as an argument to the *send()* method of a DatagramSocket.
* To receive: constructs a DatagramPacket instance with preallocated space (a byte[]), into which the contents of a received message can be copied (if/when one arrives), and then passes the instance to the receive() method of a DatagramSocket.
* Each instance of DatagramPacket also contains address and port information:
* Sent: the address and port identify the destination.
* Received: they identify the source of the received message.
* Internally, a DatagramPacket also has length and offset fields, which describe the location and number of bytes of message data inside the associated buffer.

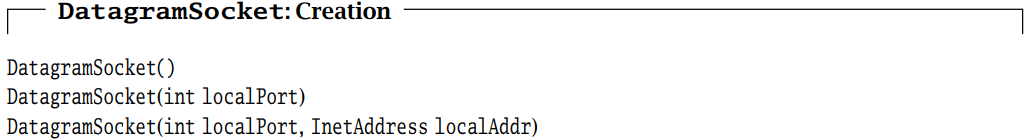


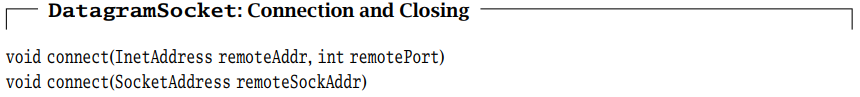


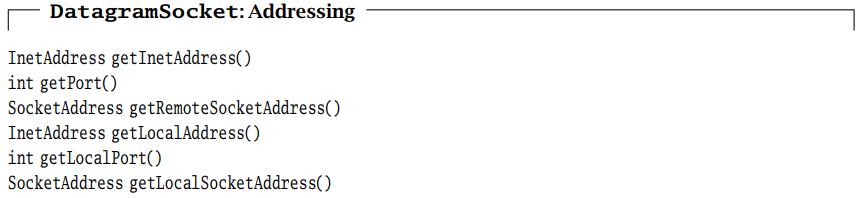


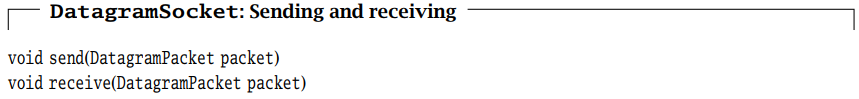
2.3.2) UDP Client

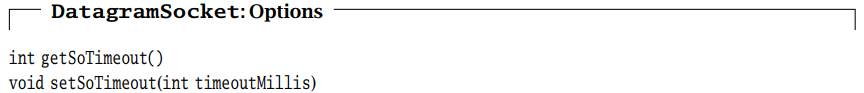
* The typical UDP client goes through three steps:
* Construct an instance of DatagramSocket, optionally specifying the local address and port.
* Communicate by sending and receiving instances of DatagramPacket using the send() and receive() methods of DatagramSocket.
* When finished, deallocate the socket using the close() method of DatagramSocket
* A UDP socket is not required to establish a connection before communication, and each datagram can be sent to or received from a different destination.
* One consequence of using UDP is that datagrams can be lost
* *setSoTimeout()* method of DatagramSocket to specify a maximum amount of time to
* block on *receive()*:
* Send the echo string to the server.
* Block on receive() for up to three seconds, starting over (up to five times) if the reply is not received before the timeout.
* Terminate the client.











2.3.3) UDP Server

* A UDP server’s job is to set up a communication endpoint and passively wait for clients to initiate communication
* The typical UDP server goes through three steps:
* 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.
* 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.
* Communicate by sending and receiving DatagramPackets using the *send()* and *receive()* methods of DatagramSocket.

2.3.4) Sending and Receiving with UDP Sockets

* UDP preserves message boundaries:
* Each call to *receive()* on a DatagramSocket returns data from at most one call to *send().*
* Different calls to *receive()* will never return data from the same call to *send()*.
* UDP does not provide recovery from network errors and, therefore, does not buffer data for possible retransmission.

→ by the time a call to send() returns, the message has been passed to

the underlying channel for transmission and is (or soon will be) on its way out the door.

* A UDP socket’s received data is kept in a queue of messages, each with associated information identifying its source.

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!

→ For this reason, a receiver should always supply a DatagramPacket that has enough space to hold the largest message allowed by the application protocol at the time it calls to receive()

* Each instance of DatagramPacket has an internal notion of message length that may be changed whenever a message is received into that instance (to reflect the number of bytes in the received message).

→ Applications that call receive() more than once with the same instance of DatagramPacket should explicitly reset the internal length to the actual buffer length before each subsequent call to receive().

* Another potential source of problems for beginners is the getData() method of DatagramPacket, which always returns the entire original buffer, ignoring the internal offset and length values.