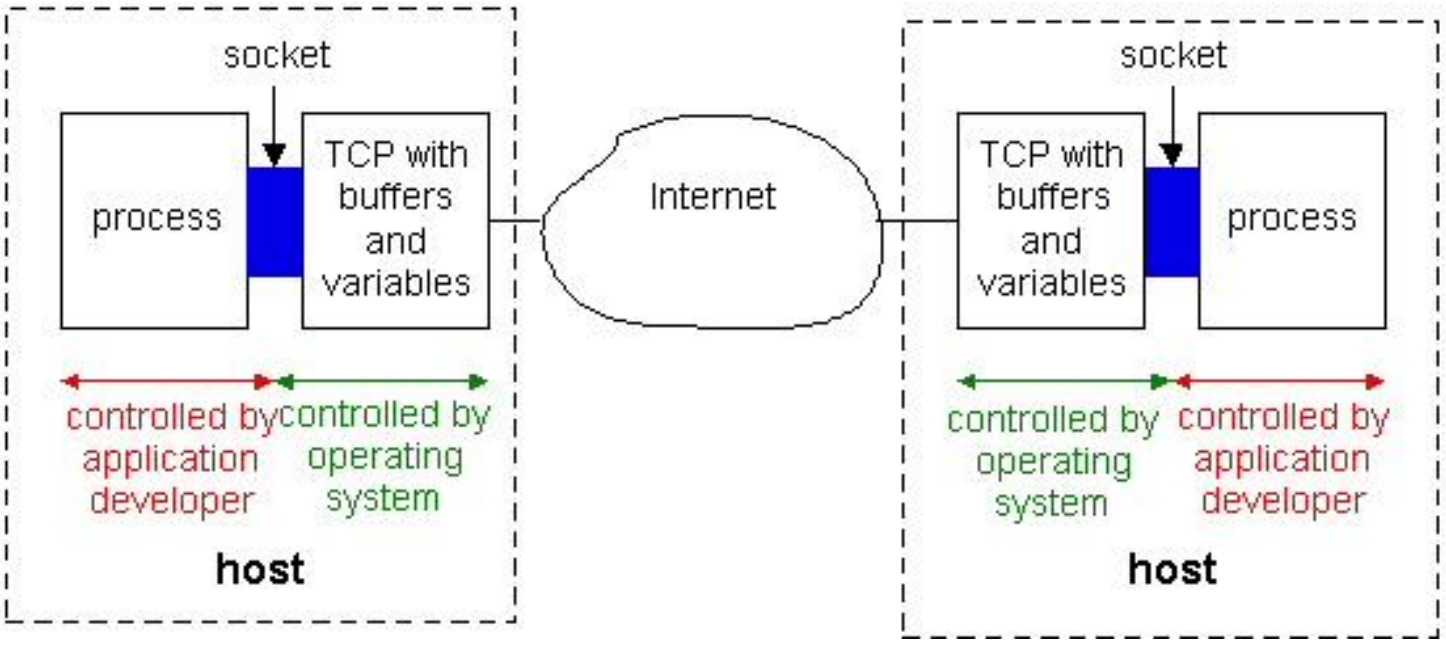
JAVA SOCKET API

Content

- Socket
- Stream Socket
- Datagram Socket
- APIs for managing names and IP addresses

Socket Programming with TCP



The application developer has the ability to fix a few TCP parameters, such as maximum buffer and maximum segment sizes.

Socket

- What is a socket ?
- *Sockets* (in plural) are an application programming interface (API) application program and the TCP/IP stack
- A *socket* is an abstraction through which an application may send and receive data
- A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network.

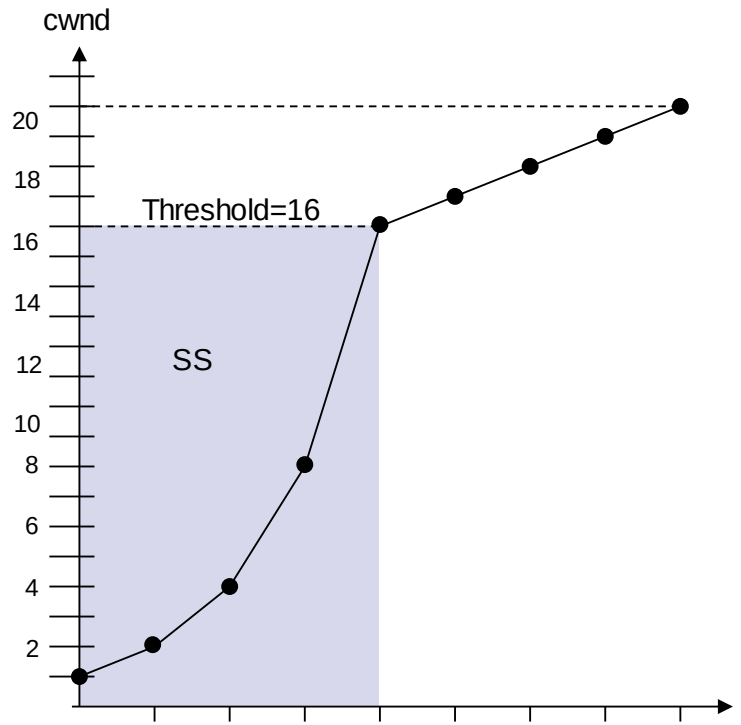
Socket (cont)

- The main types of sockets in TCP/IP are
  - *stream sockets :* use TCP as the end-to-end protocol (with IP underneath) and thus provide a reliable byte-stream service
  - *datagram sockets :* use UDP (again, with IP underneath) and thus provide a ***best-effort*** datagram service
- Socket Address: include host name and port

Stream sockets (TCP)

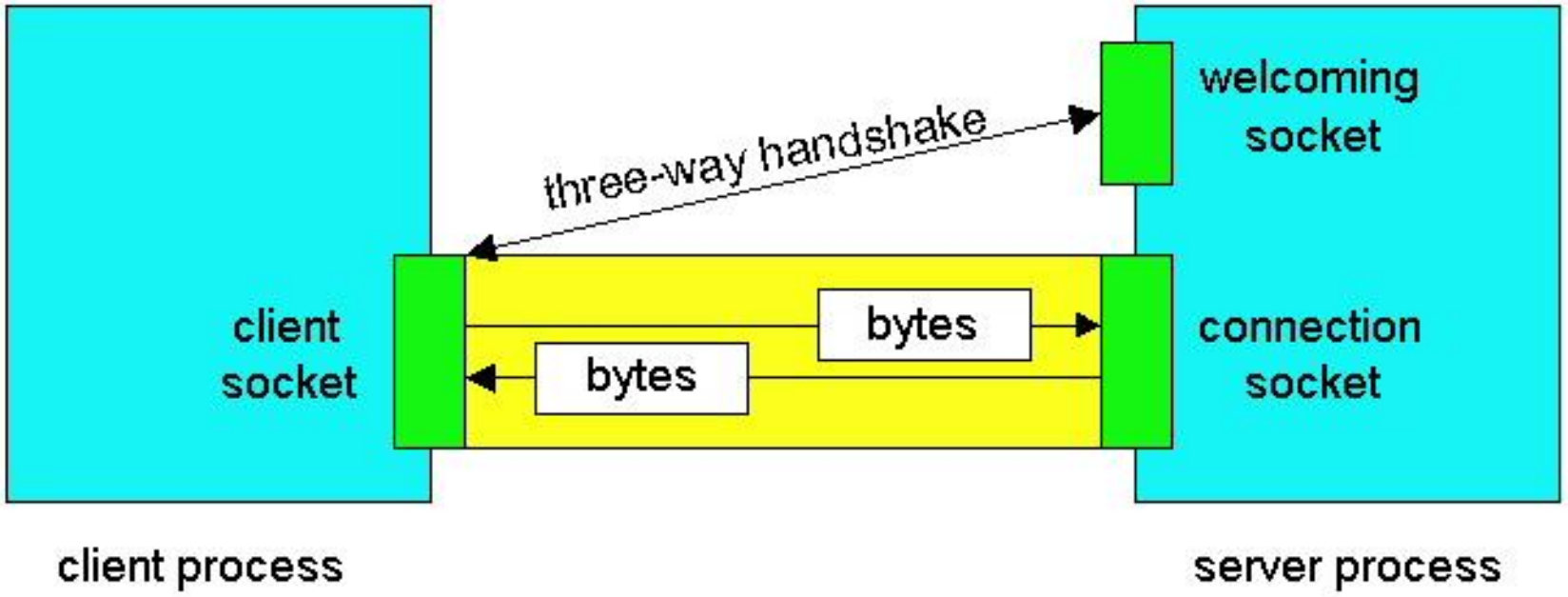
- TCP provides connections between clients and servers
- TCP also provides reliability : When TCP sends data to the other end, it requires an acknowledgment in return
- TCP connection is full-duplex: each TCP connection supports a pair of byte streams, in each direction. It means exchanging data (sending and receiving) between two entities at the same time
- TCP provides flow control

TCP flow control

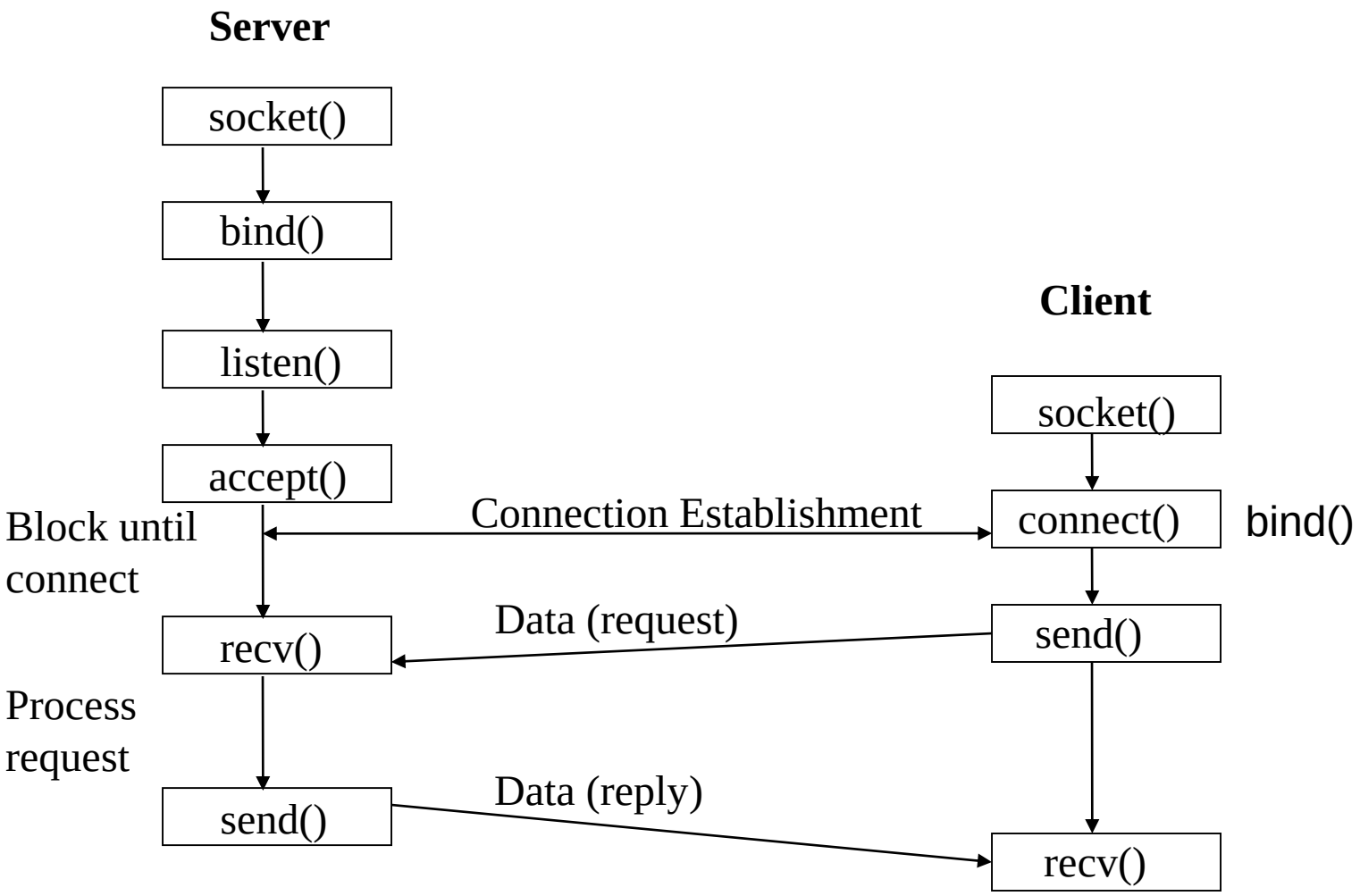


- A mechanism to ensure the sender is not overwhelming the receiver with more data than it can handle;
- With every ack message the receiver advertises its current receive window;
- TCP use a sliding window protocol to make sure it never has more bytes in flight than the window advertised by the receiver;
- When the window size is 0, TCP stop transmitting data and will start the persist timer;
- It will then periodically send a small WindowProbe message to the receiver to check if it can start receiving data again;
- When it receives a non-zero window size, it resumes the transmission.

Sockets Working Model



Stream Sockets (TCP)



Stream Socket APIs

- *socket*: creates a socket of a given domain, type, protocol (buy a phone)
- *bind*: assigns a name to the socket (get a telephone number)
- *listen*: specifies the number of pending connections that can be queued for a server socket. (call waiting allowance)
- *accept*: server accepts a connection request from a client (answer phone)
- *connect*: client requests a connection request to a server (call)
- *send*: write to connection (speak)
- *recv* : read from connection (listen)
- *close*: close a socket descriptor (end the call)

Stream Socket APIs (cont)

- socket()
  - creates a socket of a given domain, type, protocol (buy a phone)
  - Returns a file descriptor (called a socket ID)
- bind()
  - Assigns a name to the socket (get a telephone number)
  - Associate a socket with an IP address and port number (Eg : 192.168.1.1:80)
- connect()
  - Client requests a connection request to a server
  - This is the first of the client calls

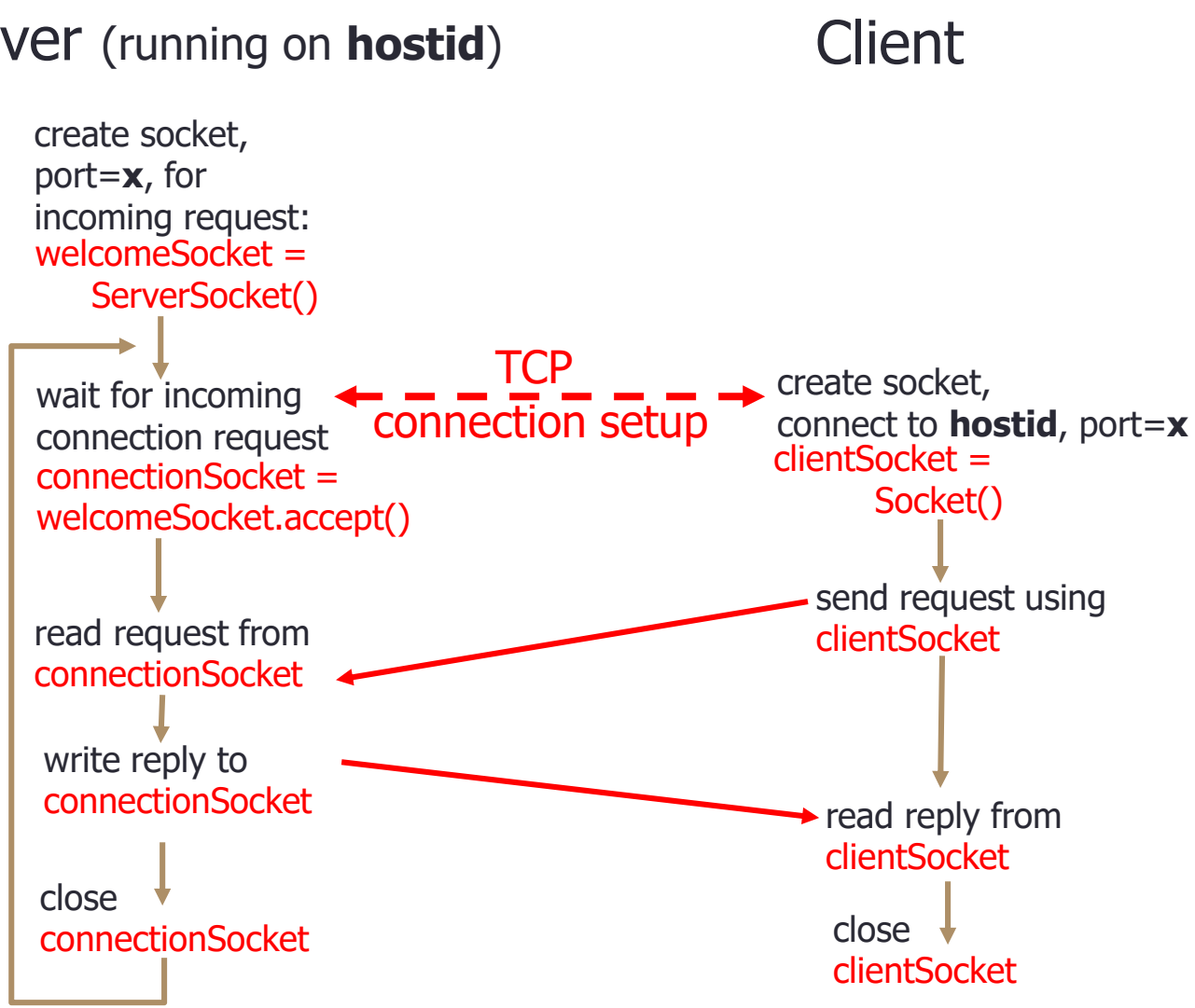
Stream Socket APIs (cont)

- accept() :
  - Server accept an incoming connection on a listening socket (request from a client)
  - There are basically three styles of using accept:
    - *Iterating server*: Only one socket is opened at a time.
    - *Forking server*: After an accept, a child process is forked off to handle the connection.
    - *Concurrent single server*: use select to simultaneously wait on all open socketIds, and waking up the process only when new data arrives

Stream Socket APIs (cont)

- listen()
  - Specifies the number of pending connections that can be queued for a server socket. (call waiting allowance)
- send()
  - Write to connection (speak)
  - Send a message
- recv()
  - read from connection (listen)
  - Receive data on a socket
- close()
  - close a socket (end the call)

Java Client/Server: TCP Socket API



JAVA Sockets

- In Package java.net
  - java.net.Socket
    - Implements client sockets (also called just "sockets").
    - An endpoint for communication between two machines.
    - Constructor and Methods
      - Socket(String host, int port): Creates a stream socket and connects it to the specified port number on the named host.
      - InputStream getInputStream()
      - OutputStream getOutputStream()
      - close()
  - java.net.ServerSocket
    - Implements server sockets.
    - Waits for requests to come in over the network.
    - Performs some operation based on the request.
    - Constructor and Methods
      - ServerSocket(int port)
      - Socket Accept(): Listens for a connection to be made to this socket and accepts it. This method blocks until a connection is made.

TCPServer.java

```
import java.io.*;
import java.net.*;
class TCPServer {
    public static void main(String argv[]) throws Exception {
   String clientSentence;
   String capitalizedSentence;

   ServerSocket welcomeSocket = new ServerSocket(6789);

   while(true) {
        Socket connectionSocket = welcomeSocket.accept();

        BufferedReader inFromClient = new BufferedReader(new
                          InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream  outToClient =
              new DataOutputStream(connectionSocket.getOutputStream());

        clientSentence = inFromClient.readLine();
        capitalizedSentence = clientSentence.toUpperCase() + '\n';

        outToClient.writeBytes(capitalizedSentence);
    }
   }
}
```

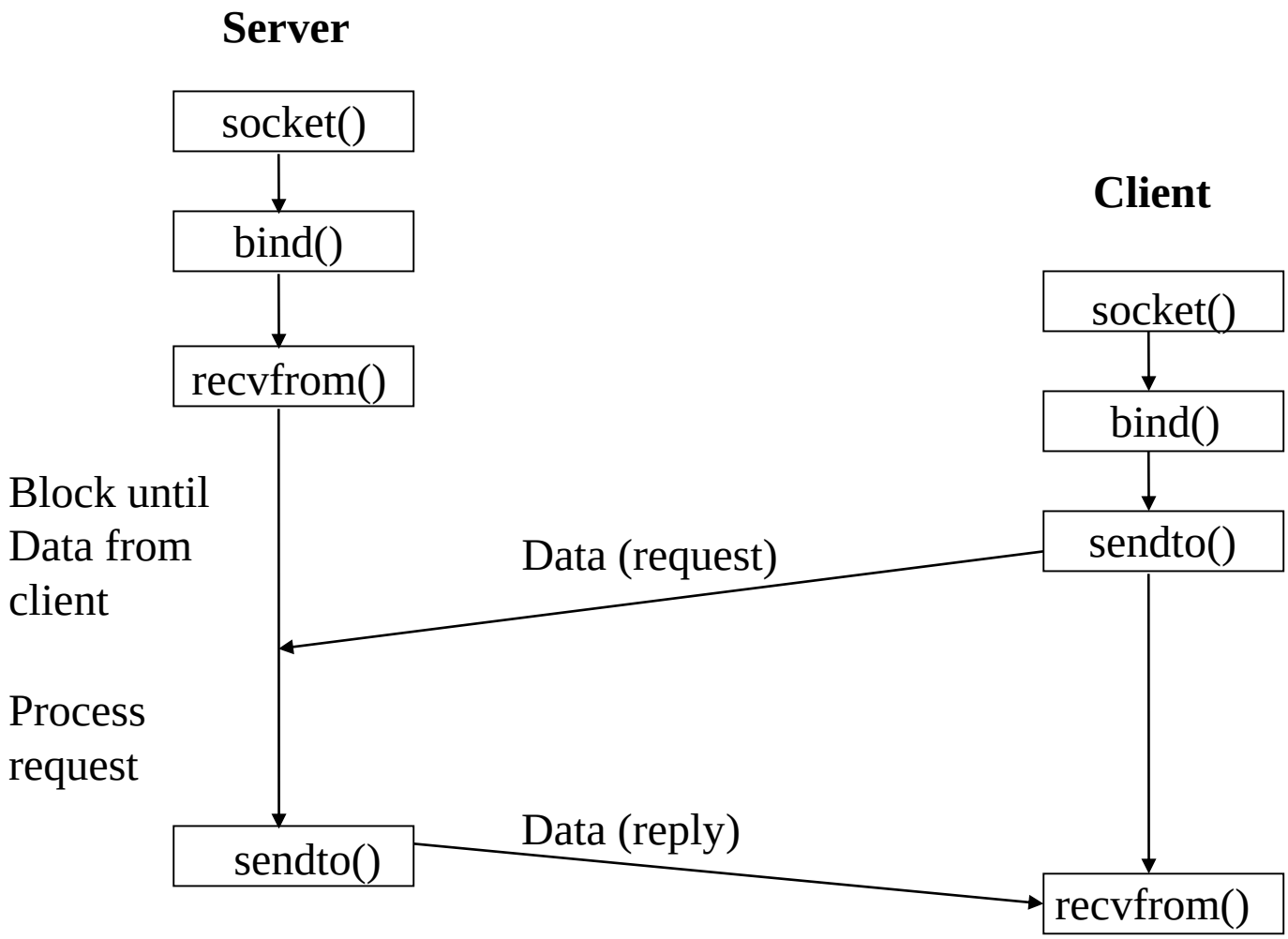
TCPClient.java

```
import java.io.*;
import java.net.*;

class TCPClient {
   public static void main(String argv[]) throws Exception {
          String sentence;
          String modifiedSentence;

              Socket clientSocket = new Socket("server IP address", 6789);

              DataOutputStream outToServer = new DataOutputStream(clientSocket.getOutputStream());

              BufferedReader inFromServer = new BufferedReader(
                            new InputStreamReader(clientSocket.getInputStream()));
                            BufferedReader inFromUser = new BufferedReader(
                            new InputStreamReader(System.in));
                            sentence = inFromUser.readLine();

              outToServer.writeBytes(sentence + '\n');
              modifiedSentence = inFromServer.readLine();
              System.out.println("FROM SERVER: " + modifiedSentence);

              clientSocket.close();

   }
}
```

Datagram Socket (UDP)

- UDP is a simple transport-layer protocol
- If a datagram is errored or lost, it won't be automatically retransmitted (can process in application)
- UDP provides a *connectionless* service, as there need not be any long-term relationship between a UDP client and server

Datagram Socket (UDP)

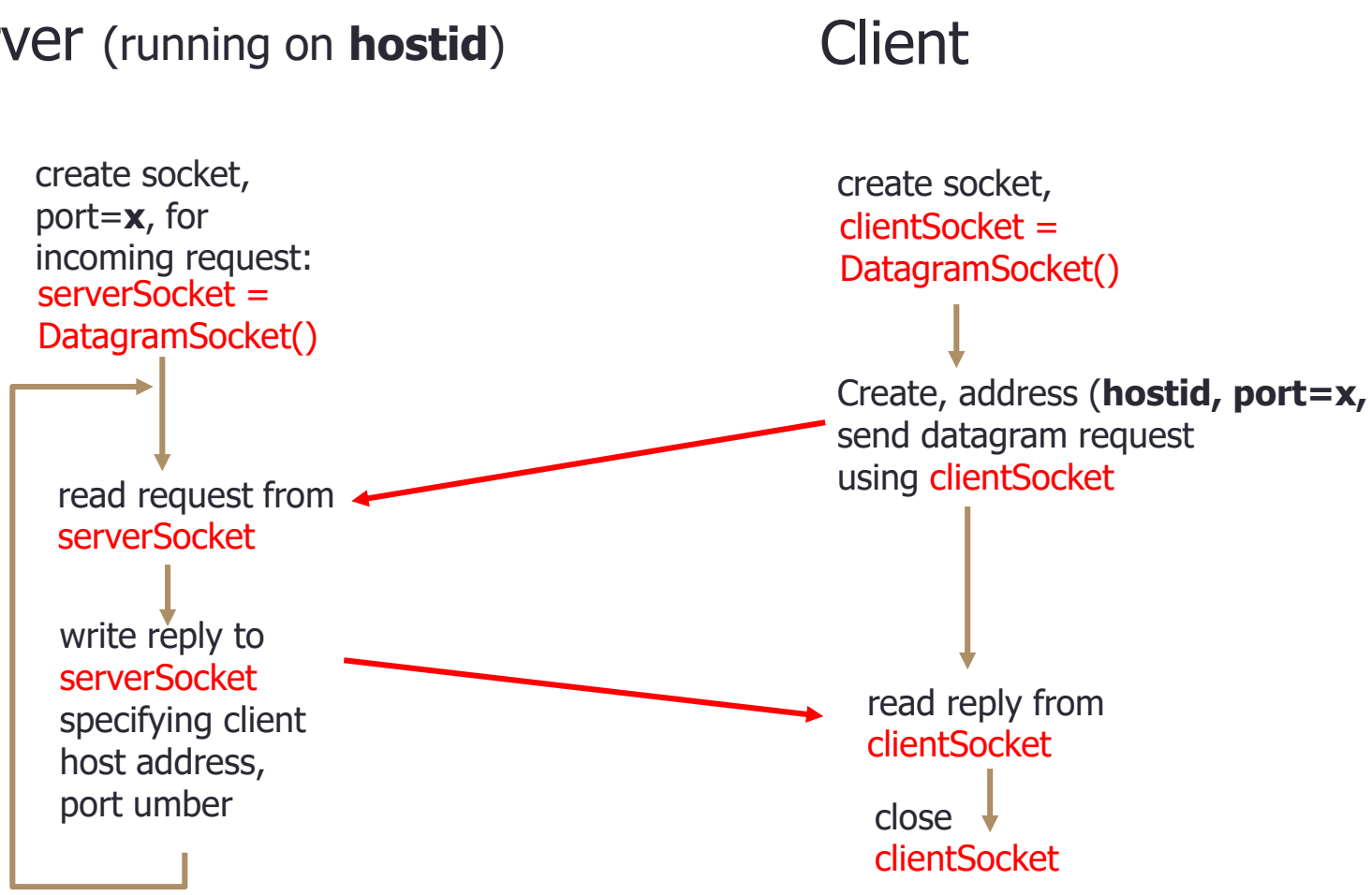


Socket Programming with UDP

- UDP
  - Connectionless and unreliable service.
  - There isn't an initial handshaking phase.
  - Doesn't have a pipe.
  - transmitted data may be received out of order, or lost

- Socket Programming with UDP
  - No need for a welcoming socket.
  - No streams are attached to the sockets.
  - the sending hosts creates "packets" by attaching the IP destination address and port number to each batch of bytes.
  - The receiving process must unravel to received packet to obtain the packet's information bytes.

Client/server socket interaction: UDP



JAVA UDP Sockets

- In Package java.net
  - java.net.DatagramSocket
    - A socket for sending and receiving datagram packets.
    - Constructor and Methods
      - DatagramSocket(int port): Constructs a datagram socket and binds it to the specified port on the local host machine.
      - void receive(DatagramPacket p)
      - void send(DatagramPacket p)
      - void close()

UDPServer.java

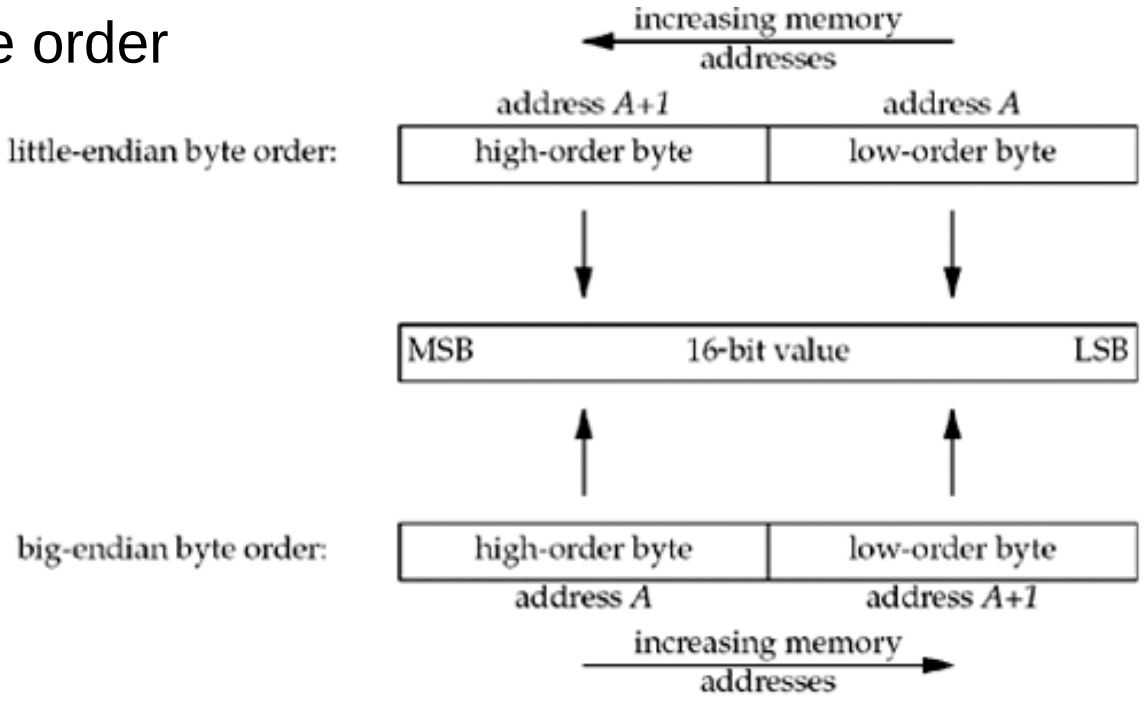
```
import java.io.*;
import java.net.*;

class UDPServer {
   public static void main(String args[]) throws Exception {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true) {

         DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
         serverSocket.receive(receivePacket);

         String sentence = new String(receivePacket.getData());
         InetAddress IPAddress = receivePacket.getAddress();

         int port = receivePacket.getPort();
         String capitalizedSentence = sentence.toUpperCase();
            sendData = capitalizedSentence.getBytes();

         DatagramPacket sendPacket =
                        new DatagramPacket(sendData, sendData.length, IPAddress, port);

         serverSocket.send(sendPacket);
         }
      }
}
```

UDPClient.java

```
import java.io.*;
import java.net.*;

class UDPClient {
   public static void main(String args[]) throws Exception {
      BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();
      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
      DatagramPacket sendPacket =
                        new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
      clientSocket.send(sendPacket);

      DatagramPacket receivePacket =
                        new DatagramPacket(receiveData, receiveData.length);

      clientSocket.receive(receivePacket);
      String modifiedSentence = new String(receivePacket.getData());
      System.out.println("FROM SERVER:" + modifiedSentence);

      clientSocket.close();

   }
}
```

Byte Ordering

- There are two ways to store the two bytes in memory
  - little-endian byte order
  - big-endian byte order



Byte Ordering (cont)

- There is no standard between these two byte orderings
- A variety of systems that can change between little-endian and big-endian byte ordering
- Problem : Converting between
  - *host byte order*
  - *network byte order* (The Internet protocols use big-endian byte ordering)
- Four functions to convert between these two byte orders.

htons(), htonl(), ntohs(), ntohl()

- Convert multi-byte integer types from host byte order to network byte order

```
#include <netinet/in.h>

uint32_t htonl(uint32_t hostlong);  // host to network long
uint16_t htons(uint16_t hostshort);// host to network short
uint32_t ntohl(uint32_t netlong);   // network to host long
uint16_t ntohs(uint16_t netshort); // network to host short
```

- Java Class ByteOrder: https://docs.oracle.com/javase/8/docs/api/java/nio/ByteOrder.html

Mini Project: Building a Simple Web Server

- Handles only one HTTP request
- Accepts and parses the HTTP request
- Gets the required file from the server's file system.
- Creates an HTTP response message consisting of the requested file preceded by header lines
- Sends the response directly to the client

WebServer.java

```
import java.io.*;
import java.net.*;
import java.util.*;
class WebServer{
    public static void main(String argv[]) throws Exception  {
        String requestMessageLine;
      String fileName;
        ServerSocket listenSocket = new ServerSocket(6789);
      Socket connectionSocket = listenSocket.accept();

        BufferedReader inFromClient =
            new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));

     DataOutputStream outToClient =
              new DataOutputStream(connectionSocket.getOutputStream());
```

WebServer.java

```
requestMessageLine = inFromClient.readLine();

StringTokenizer tokenizedLine =
   new StringTokenizer(requestMessageLine);

if (tokenizedLine.nextToken().equals("GET")){
          fileName = tokenizedLine.nextToken();
          if (fileName.startsWith("/") == true )
                fileName  = fileName.substring(1);

        File file = new File(fileName);
         int numOfBytes = (int) file.length();
            FileInputStream inFile  = new FileInputStream (fileName);
            byte[] fileInBytes = new byte[numOfBytes];

        inFile.read(fileInBytes);
```

WebServer.java

```
    outToClient.writeBytes("HTTP/1.0 200 Document Follows\r\n");

  if (fileName.endsWith(".jpg"))
      outToClient.writeBytes("Content-Type: image/jpeg\r\n");

  if (fileName.endsWith(".gif"))
      outToClient.writeBytes("Content-Type: image/gif\r\n");

  outToClient.writeBytes("Content-Length: " + numOfBytes + "\r\n");

  outToClient.writeBytes("\r\n");
  outToClient.write(fileInBytes, 0, numOfBytes);
     connectionSocket.close();
   }
    else System.out.println("Bad Request Message");
  }
}
```

Concurrent Problem

- Servers need to handle a new connection request while processing previous requests.
  - Most TCP servers are designed to be concurrent.
- When a new connection request arrives at a server, the server accepts and invokes a new process to handle the new client.

How to handle the port numbers

cosmos% **netstat –a –n –f inet**

Active Internet connections (including servers)

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp	0	0	*.23	*.*	LISTEN

cosmos% **netstat –a –n –f inet**

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp	0	0	192.249.24.2.23	192.249.24.31.1029	ESTABLISHED
tcp	0	0	*.23	*.*	LISTEN

cosmos% **netstat –a –n –f inet**

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	(state)
tcp	0	0	192.249.24.2.23	192.249.24.31.1029	ESTABLISHED
tcp	0	0	192.249.24.2.23	192.249.24.31.1030	ESTABLISHED
tcp	0	0	*.23	*.*	LISTEN