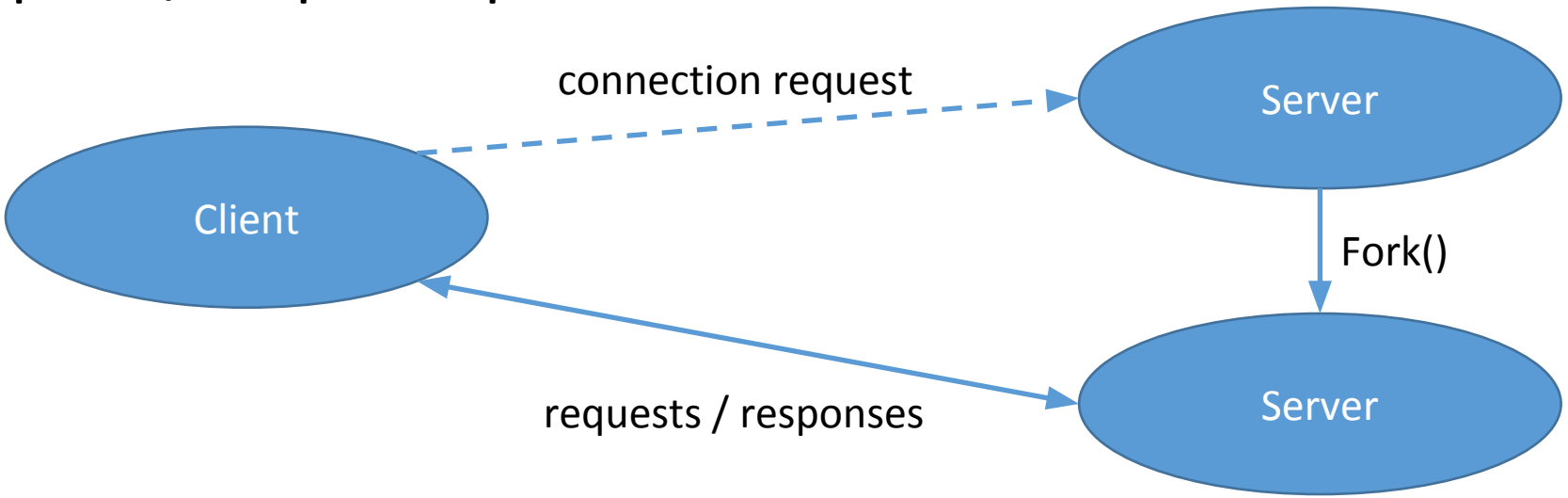
Networking with Sockets in Java World

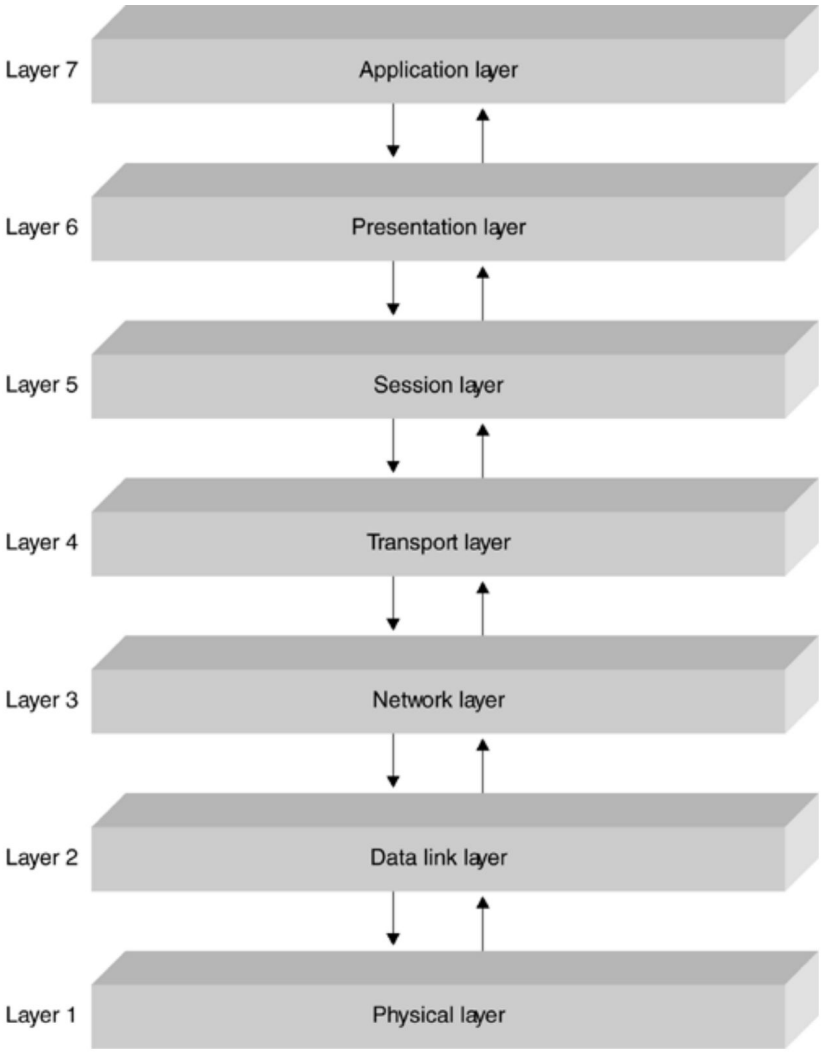
Alexander Vlasov

Client-Server Architecture

- The most commonly used model for distributed applications
  - Can be applied for a particular request-response interaction
- The client is the entity (process) accessing the remote resource and the server provides access to the resource
- Request / response protocols



Seven layers of the OSI Reference Model



Layer 7—Application Layer. The final OSI layer is the application layer, which is where the vast majority of programmers write code. Application layer protocols dictate the semantics of how requests for services are made, such as requesting a file or checking for e-mail. In Java, almost all network software written will be for the application layer, although the services of some lower layers may also be called upon.

Layer 6—Presentation Layer. The sixth layer deals with data representation and data conversion. Different machines use different types of data representation (an integer might be represented by 8 bits on one system and 16 bits on another). Some protocols may want to compress data, or encrypt it. Whenever data 6 types are being converted from one format to another, the presentation layer handles these types of tasks.

Layer 5—Session Layer. The purpose of the session layer is to facilitate application-to-application data exchange, and the establishment and termination of communication sessions. Session management involves a variety of tasks, including establishing a session, synchronizing a session, and reestablishing a session that has been abruptly terminated. Not every type of application will require this type of service, as the additional overhead of connection-oriented communication can increase network delays and bandwidth consumption. Some applications will instead choose to use a connectionless form of communication.

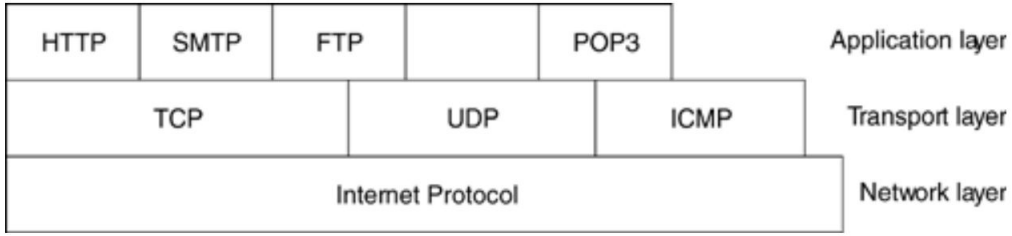
Layer 4—Transport Layer. The fourth layer, the transport layer, is concerned with controlling how data is transmitted. This layer deals with issues such as automatic error detection and correction, and flow control (limiting the amount of data sent to prevent overload).

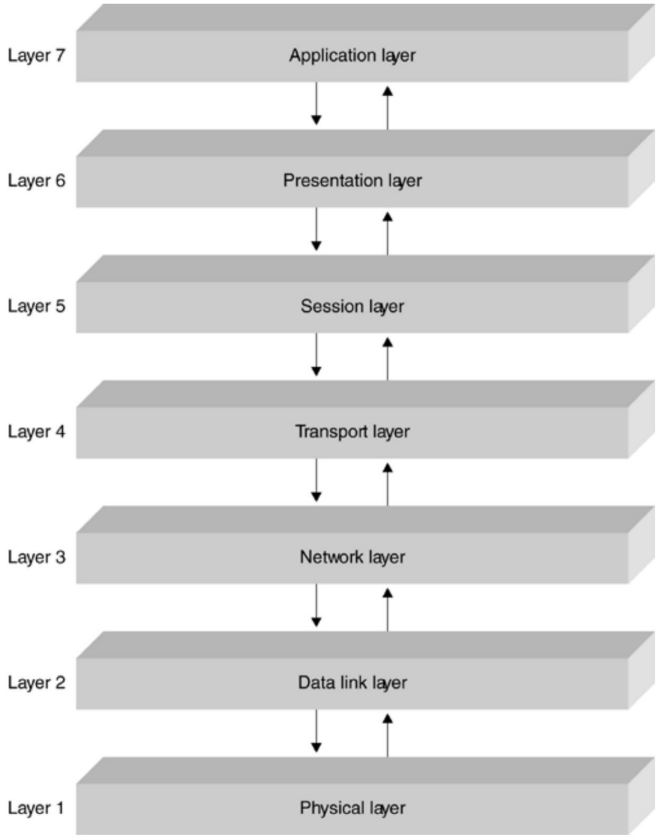
Layer 3—Network Layer. Moving up from the data link layer, which sends frames over a network, we reach the network layer. The network layer deals with data packets, rather than frames, and introduces several important concepts, such as the network address and routing. Packets are sent across the network, and in the case of the Internet, all around the world. Unless traveling to a node in an adjacent network where there is only one choice, these packets will often take alternative routes (the route is determined by routers). Communication at this level is still very low-level; network programmers are rarely required to write software services for this layer.

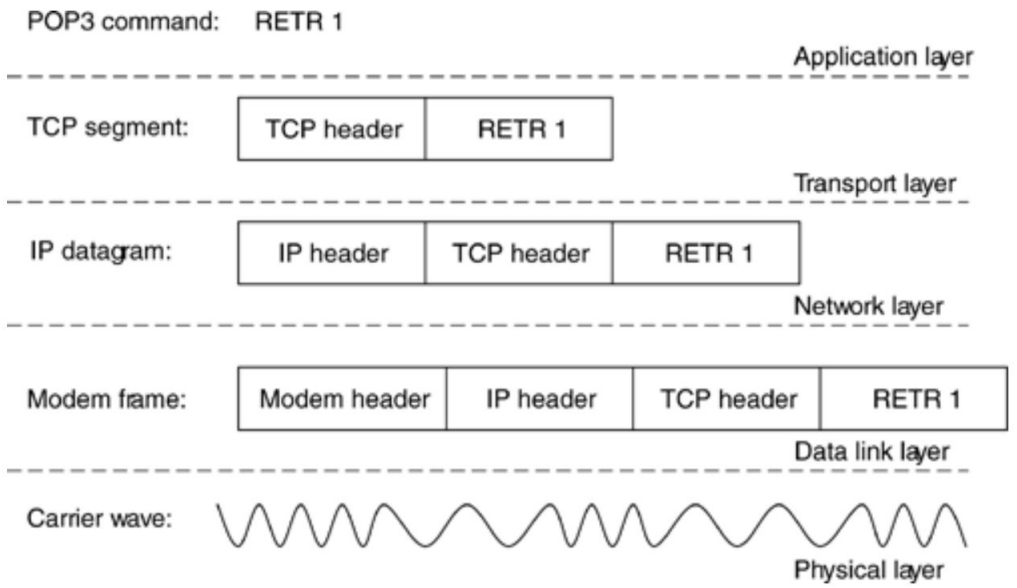
Layer 2—Data Link Layer. The data link layer is responsible for providing a more reliable transfer of data, and for grouping data together into frames. Frames are similar to data packets, but are blocks of data specific to a single type of hardware architecture (whereas data packets are used at a higher level and can move from one type of network to another). Frames have checksums to detect errors in transmission, and typically a "start" and "end" marker to alert hardware to the division between one frame and another. Sequences of frames are transmitted between network nodes, and if a frame is corrupted it will be discarded. The data link layer helps to ensure that garbled data frames will not be passed to higher layers, confusing applications. However, the data link layer does not normally guarantee retransmission of corrupted frames; higher layers normally handle this behavior.

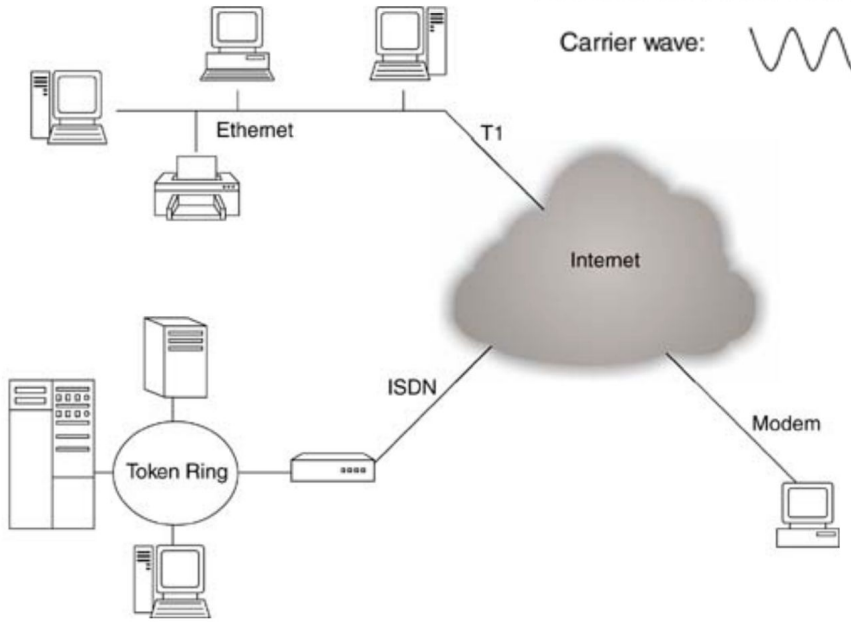
Layer 1—Physical Layer. The physical layer is networking communication at its most basic level. The physical layer governs the very lowest form of communication between net-work nodes. At this level, networking hardware, such as cards and cables, transmit a sequence of bits between two nodes. Java programmers do not work at this level—it is the domain of hardware driver developers and electrical engineers. At this layer, no real attempt is made to ensure error-free data transmission. Errors can occur for a variety

TCP/IP stack divided by lavers









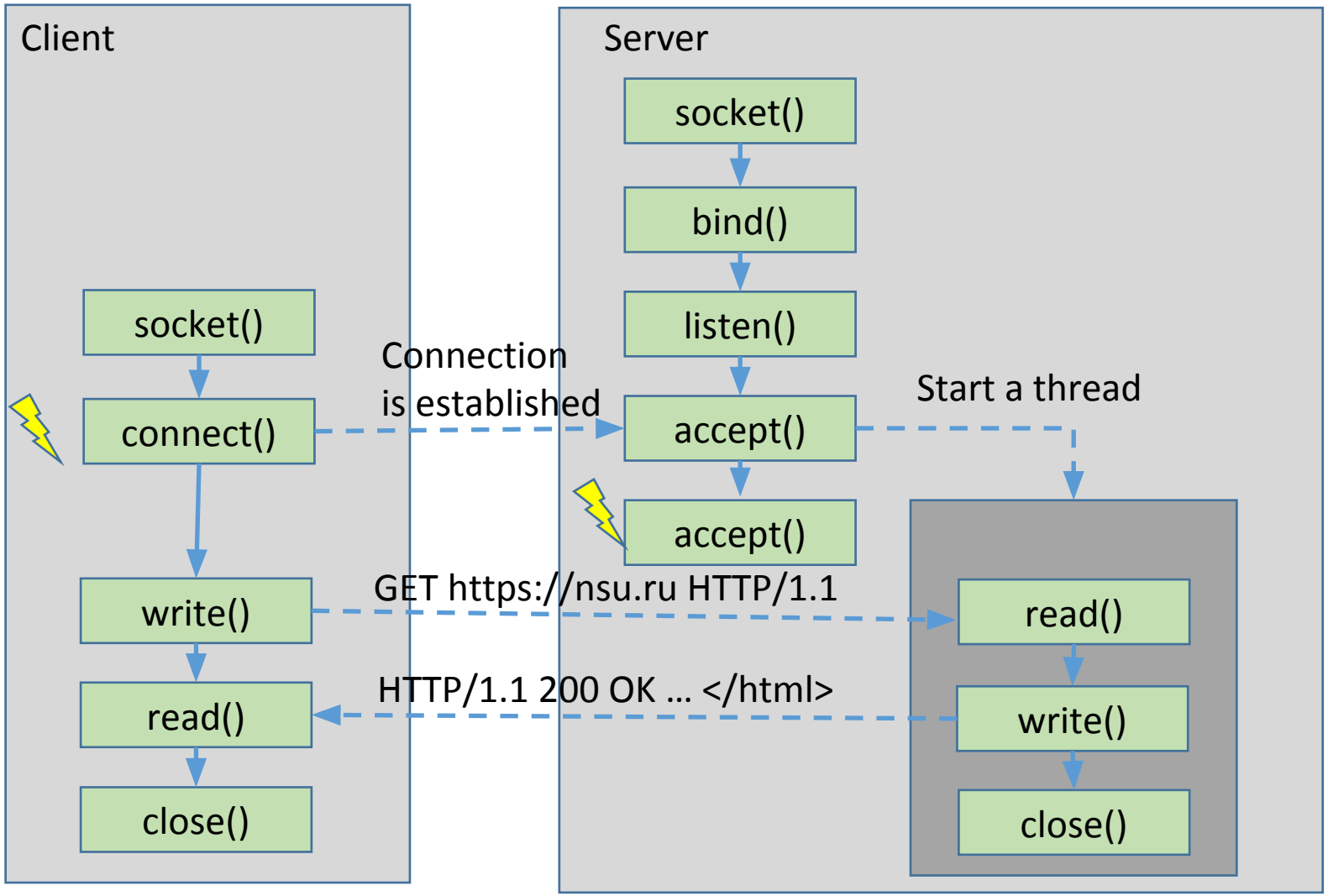
Sockets

- Socket is an end-point of a virtual network connection between processes – much like a full-duplex channel – A socket address: IP address and a port number – A transport protocol used for communication over a socket
- TCP socket – stream-based, connection-oriented
- UDP socket – datagram-based, connectionless
- Sockets, a.k.a. Berkeley sockets, were introduced in 1981 as the Unix BSD 4.2 generic API for inter-process communication
- Earlier, a part of the kernel (BSD Unix)
- Now, a library (Linux, Solaris, Windows, MacOS)

Ports

- Port is an entry point to a process that resides on a host.
- 65,535 logical ports with integer numbers 1 - 65,535
- A port can be allocated to a particular service:
  - A server listens the port for incoming requests
  - A client connects to the port and requests the service
  - The server replies via the port.
- Ports with numbers 1-1023 are reserved for well-known services.

The Berkeley Socket API for the Client-Server Architecture



Sockets in **java.net**

Two classes of **TCP sockets**:

- **Socket** – connecting socket, a.k.a. client socket
  - Used to connect to another (remove) TCP socket specified by an IP address and a port number.
  - A connected TCP socket provides two sequenced byte streams, input and output streams, used to communicate with the remote process by reads and writes.
- **ServerSocket** – listening socket, a.k.a. server socket
  - Used to listen for connection requests, to accept connection and create a **Socket** object connected to the requester.

Sockets in **java.net**

Two classes of **UDP sockets**:

- **DatagramSocket**
  - used for sending and/or receiving datagrams represented by objects of the **DatagramPacket** class.
- **MulticastSocket**
  - is a subclass of **DatagramSocket** with capabilities for joining multicast groups on the Internet.
  - A multicast group is identified by an IP address of class D (multicast address (224-239.x.x.x)

java.net.InetAddress

- Represents an IP address of a node on the Internet.
- Does not have public constructors.
- Getting an IP address:

```
import java.util.*;
import java.net.*;

public class MainInetAddress {
    public static void main(String[] args) {
        InetAddress ip1 = null;
        InetAddress ip2 = null;
        InetAddress localIP = null;
        InetAddress[] ips = null;
        try {
            localIP = InetAddress.getLocalHost();
            ip1 = InetAddress.getByName("nsu.ru");
            ip2 = InetAddress.getByName("84.237.50.25");
            ips = InetAddress.getAllByName("nsu.ru");
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

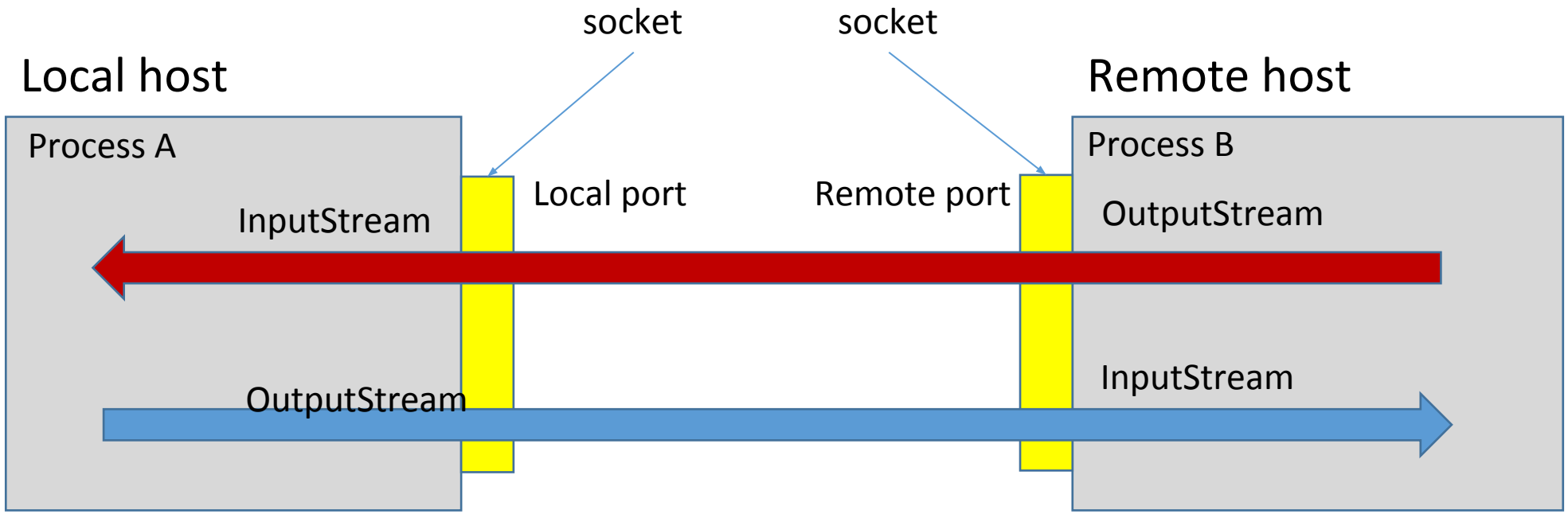
Link documentation

Link source

java.net.Socket

- Implements a connecting TCP socket that provides connection to a specified host on a specified port.
- When connected, provides input and output byte streams

Link documentation



Communicating via a TCP Socket

- Steps:
  - Establish a socket connection to the specified host on the specified port by create a connected Socket object.
  - Set socket's attributes.
  - Get an input stream of the socket connection for reading data.
  - Get an output stream of the connection for writing data.
  - Communicate via the input and output streams by reads and writes according to an application specific communication protocol.
  - Close the socket connection.

TCPServer

```
class TCPServerSingleThreaded implements Runnable {
   private ServerSocket serverSocket;
   public TCPServerSingleThreaded(int port) throws IOException {
      this.serverSocket = new ServerSocket(port);
   }
   public void run() {
      while(true) {
         try {
            Socket socket = serverSocket.accept();
            new ConnectionHandler(socket).run();
         } catch (IOException e) { /*…*/ }
      }
   }
}
```

```
class TCPServerMultiThreaded implements Runnable {
   ServerSocket serverSocket;
   public TCPServerMultiThreaded(int port) throws IOException {
      this.serverSocket = new ServerSocket(port);
   }
   public void run() {
      for (;;) {
         try {
            Socket socket = serverSocket.accept();
            new Thread(new ConnectionHandler(socket)).start();
         } catch (IOException e) { /*…*/ }
      }
   }
}
```

ConnectionHandler

```
class ConnectionHandler implements Runnable {
   private Socket socket;
   public ConnectionHandler(Socket socket) {
      this.socket = socket;
   }
   public void run() {
      handleConversation(socket);
   }
   public void handleConversation(Socket socket) {
      try(socket) {
         InputStream in = socket.getInputStream();
         OutputStream out = socket.getOutputStream();
         // write reply to the output
         out.flush();
            ch (IOException e) {
```

```
class EchoConnectionHandler extends ConnectionHandler {
   EchoConnectionHandler(Socket socket) { super(socket); }
   public void handleConversation(Socket socket) {
      try(socket) {
         InputStream in = socket.getInputStream();
         OutputStream out = socket.getOutputStream();
         byte[] buffer = new byte[8192];
         int count;
         while ((count = in.read(buffer)) >= 0) {
            out.write(buffer,0,count);
         }
         out.flush();
      } catch (IOException e) {
      }
   }
}
```

TCPClient

```
class TCPClient implements Runnable {
    Socket socket;
    public void run() {
        var host="localhost";
        var port=9899;
        try(var socket = new Socket(host,port)) {
            OutputStream out = socket.getOutputStream();
            out.flush();
            InputStream in = socket.getInputStream();
        } catch (IOException e) { /*…*/ }
    }
}
```

Broadcast to current physical network

```
try {
    int port = 8888;
    byte[] data = {0, 1, 2, 3, 4, 5, 6, 7};
    // send to all nodes on the current physical network
    // via the limited broadcast address
    InetAddress address = InetAddress.getByName("255.255.255.255");
    DatagramPacket packet = new DatagramPacket(data, data.length, address, port);
    DatagramSocket socket = new DatagramSocket();
    socket.send(packet);
}catch(UnknownHostException  e){
}
catch(IOException err){
}
```

```
// send to all nodes in 192.168.1.*
// via a directed broadcast address
InetAddress address = InetAddress.getByName("192.168.1.255");
```

Receive broadcast datagrams

```
try {
    int port = 8888;
    DatagramSocket socket = new DatagramSocket(port);
    byte[] data = new byte[8192+1];
    DatagramPacket packet = new DatagramPacket(data, data.length);
    socket.receive(packet);
}catch(UnknownHostException  e){
}
catch(IOException err){
}
```

```
DatagramSocket socket = new DatagramSocket(null);
socket.bind(new InetSocketAddress(8888));
```