# Socket programming

## Goals:

- Present the basics of socket programming
- Show concretely how it works using Java

## Agenda:

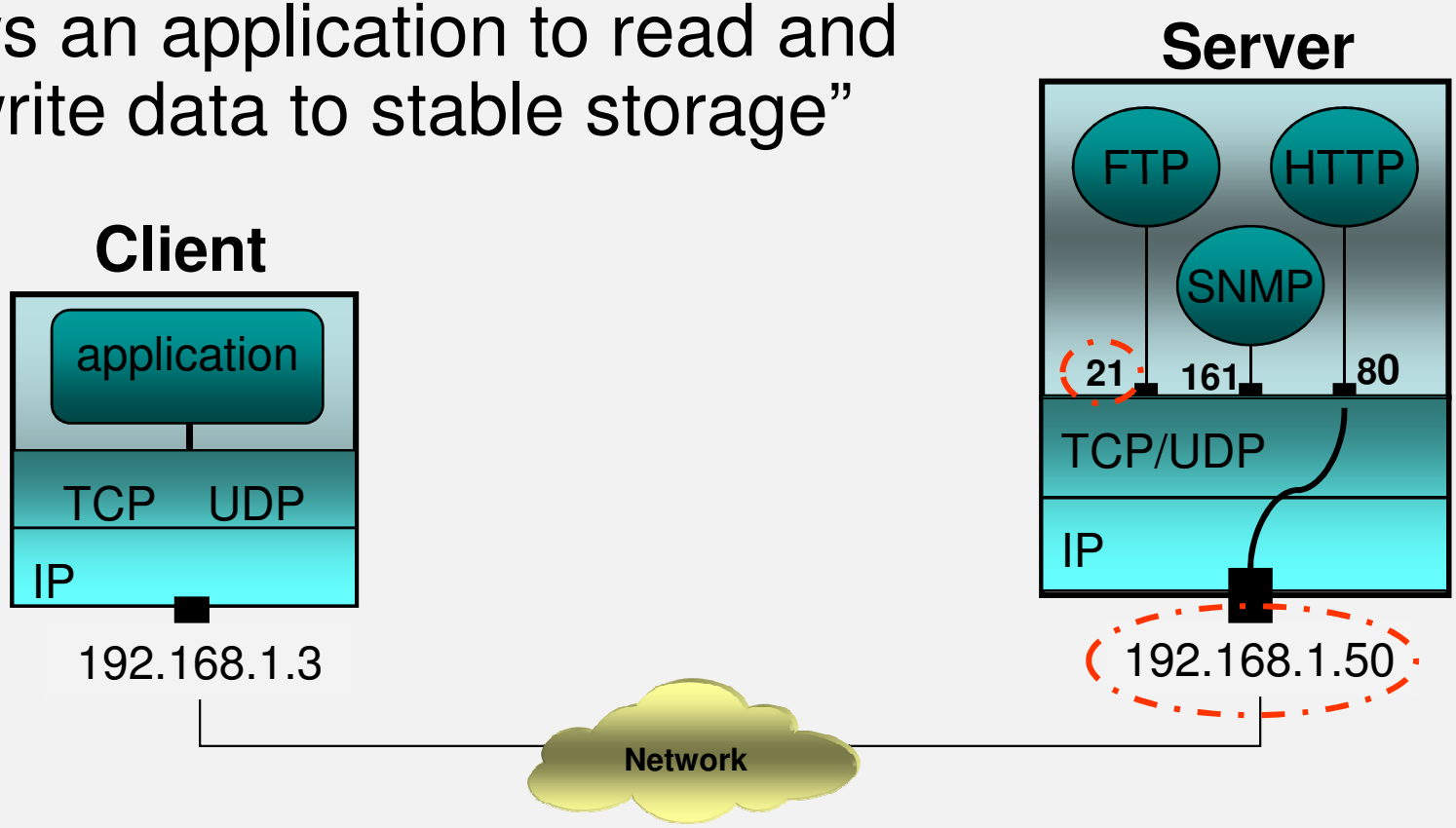
- Basics
- Client and server implementation
- Sending and receiving data

# Basics

- What is a socket?
- Socket communication

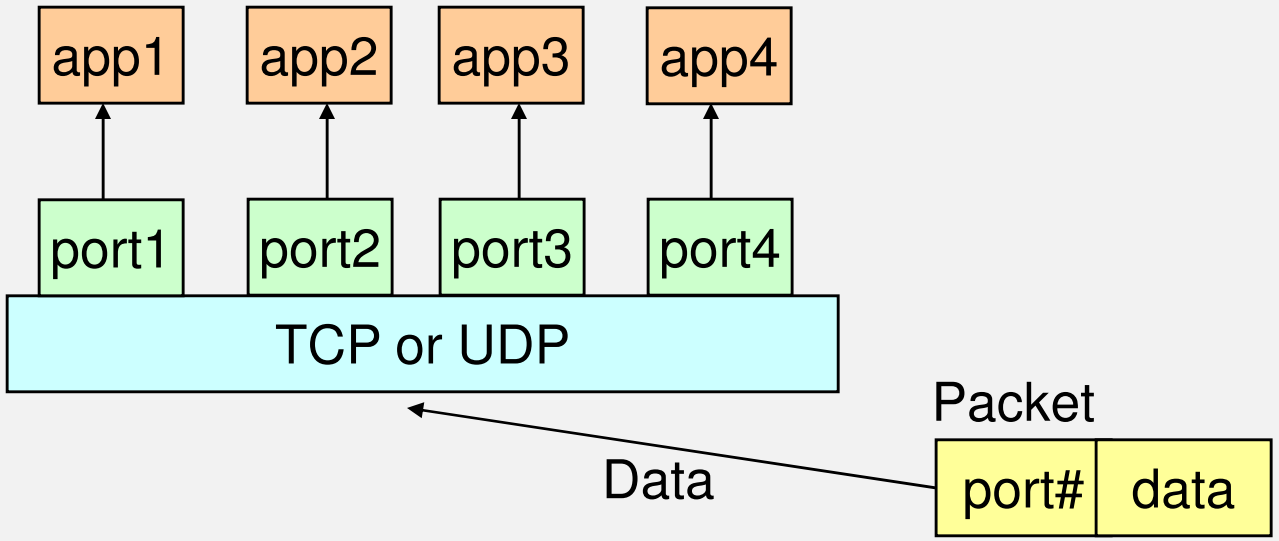
# What is a socket

"A *socket* is an abstraction through which an application may send and receive data, in much the same way as an open file handle allows an application to read and write data to stable storage"



# What is a socket

- The *ports* are used by TCP and UDP protocols to identify the destination program (application) of an incoming data



# What is a socket

- Port numbers between 0 and 1,023 are reserved (used by common/well known services)
  - FTP: 21/tcp, 21/udp
  - HTTP: 80/tcp, 80/udp
  - HTTPS: 443/tcp, 443/udp
  - SNMP: 161/tcp, 161/udp

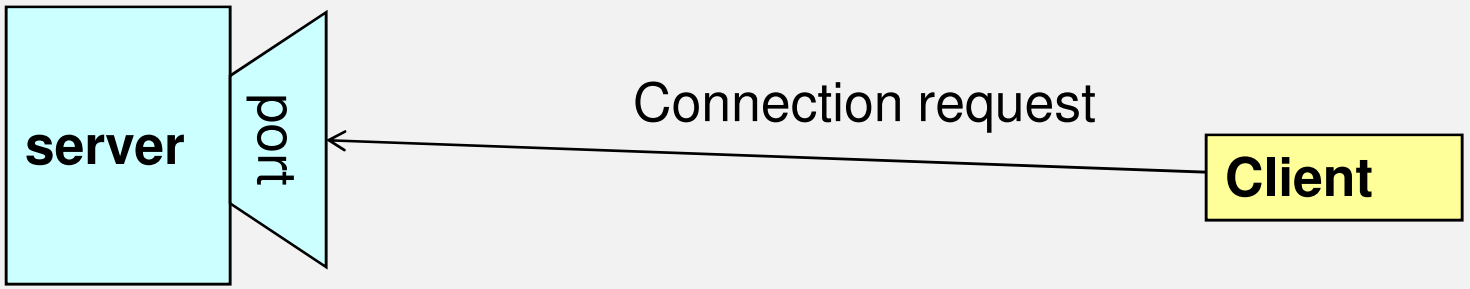
  * managed by the Internet Assigned Numbers Authority (IANA)

- When selecting a port number for your server, select one that is greater than 1,023

# Socket communication

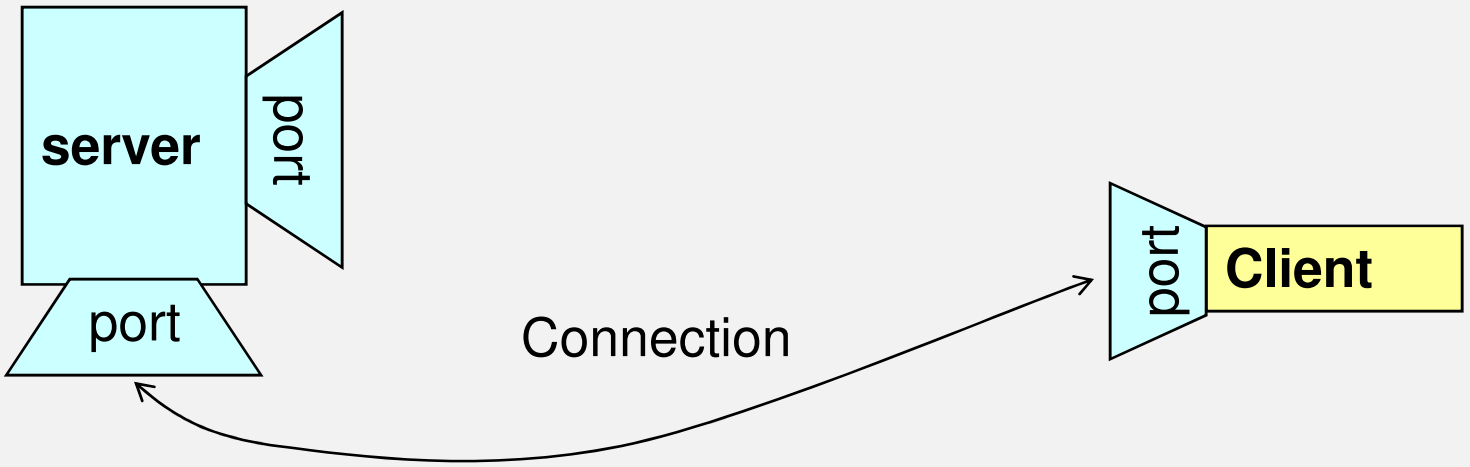
- A server (application) runs on a specific computer and has a socket that is bound to a specific port.

- The server listens to the socket  and waits for a client to make a connection request.



# Socket communication

- The server accepts the incoming connection request.
- The server gets a new socket bounds to a different port.



# Socket communication

- Two main communication protocols can be used for socket programming
  - Datagram communication
    - datagram sockets (UDP)
  - Stream communication
    - stream sockets *(*TCP)

# Socket communication

- Datagram communication
  - UDP is a connectionless protocol
    - For each datagram, we need to send the local socket descriptor and the receiving socket's address
    - There is a size limit of 65,500 bytes on each datagram
    - No guarantee that the sent datagrams will be received in the same order

    ==> UDP is often used in implementing client/server applications built over local area networks

# Socket communication

- Stream communication
  - TCP is a connection-oriented protocol
    - A connection must first be established between the client and the server.
    - No limit on the data to send
    - The sent packets are received in the order in which they were sent.

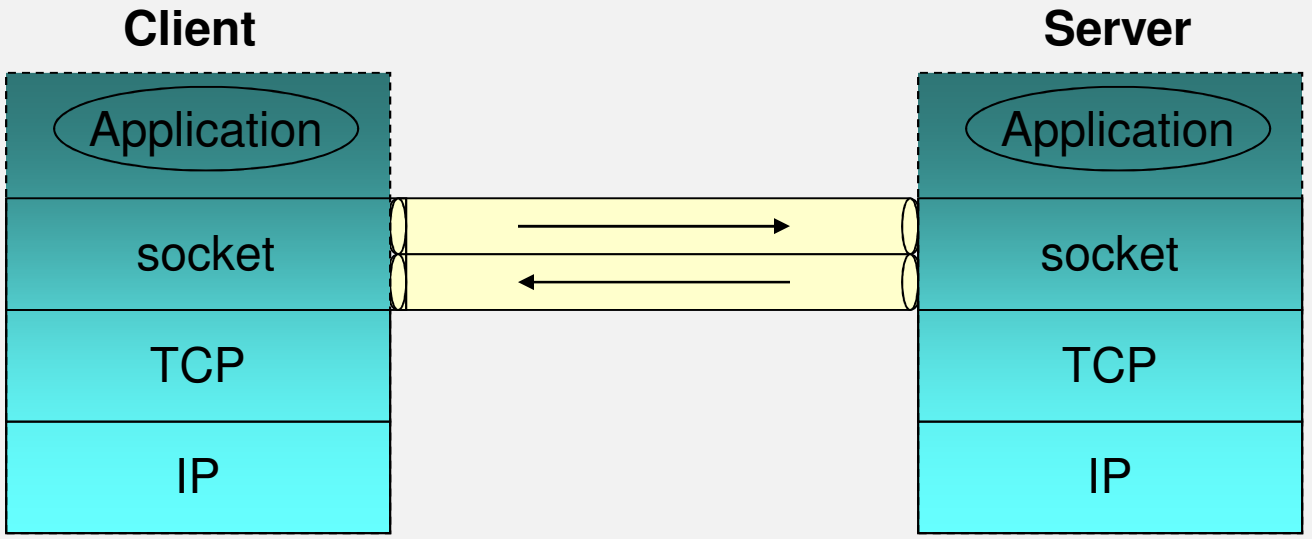
    ==> TCP is useful for implementing network services such as remote login (rlogin, telnet) and file transfer (FTP)

# Client and server implementation

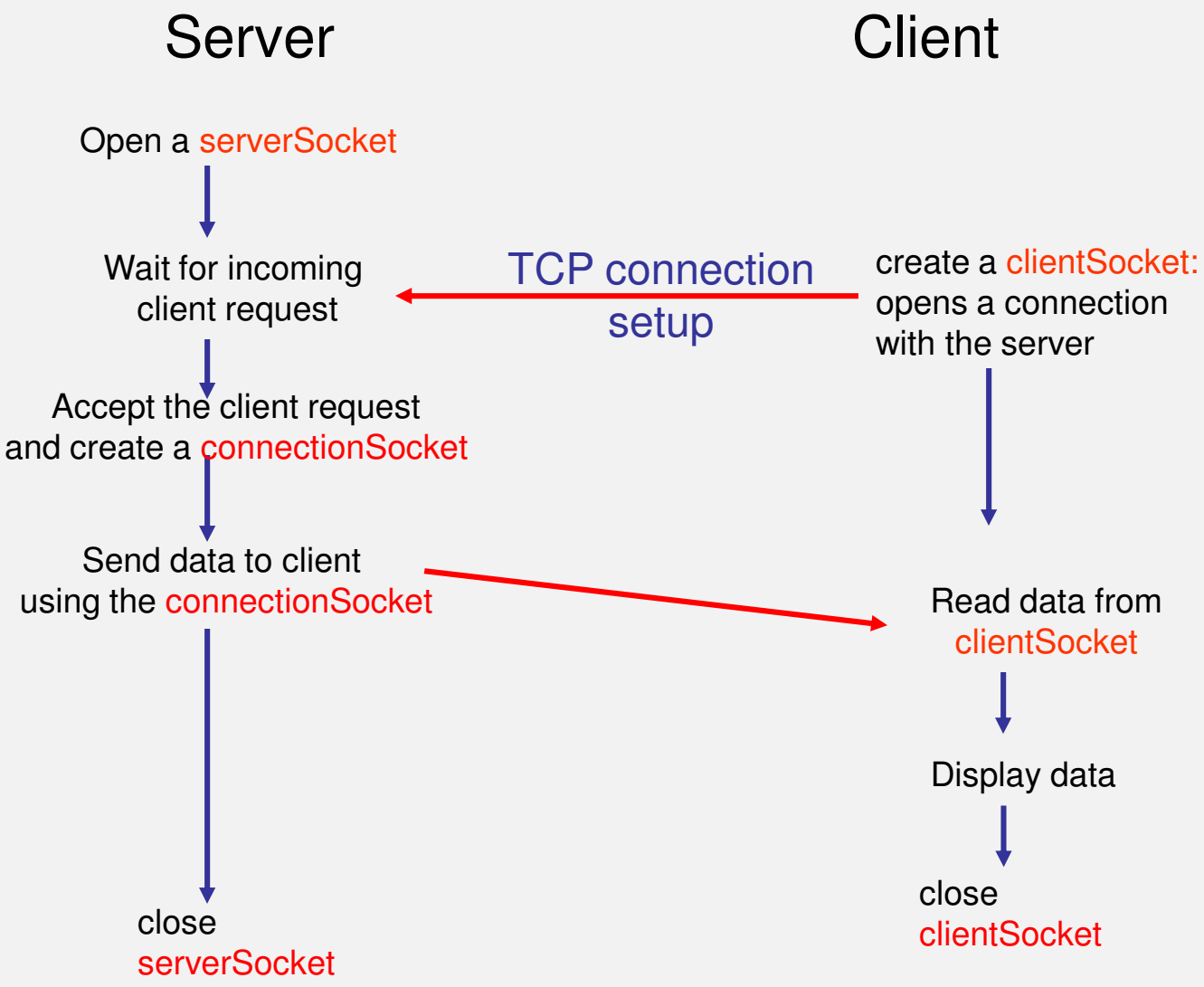
- Main classes
- TCP client/server implementation
- UDP client/server implementation

# Main classes

- java.net package provides three main classes:
  - Socket – for implementing a TCP client
  - ServerSocket – for implementing a TCP server
  - DatagramSocket – for implementing both a UDP client and server
- Data exchange
  - TCP: InputStream and OutputStream
  - UDP: DatagramPacket



# TCP client/server implementation



# TCP server implementation

Open a server socket

```
ServerSocket server;
    try {
        server = new ServerSocket(portNumber);
    } catch (IOException e) { System.out.println(e); }
```

Create a connection socket object

```
try {
    Socket connectSocket = server.accept();
} catch (IOException e) { .. }
```

Write data to the connection socket

```
String data = "Hello from server";
try {
    OutputStream out = connectSocket.getOutputStream();
    out.write(data.getBytes()  );
  } catch (IOException e) {…}
```

# TCP server implementation

Close the sockets

```
try {
        connectSocket.close(); // Close the socket. We are
                               // done with this client!
        server.close();   // close the server socket

} catch (IOException e) { System.out.println(e); }
```

# TCP client implementation

Open a client socket

```
try {
    Socket  clientSocket = new Socket(serverIP/Name, serverPort);
} catch (IOException e) { System.out.println(e); }
```

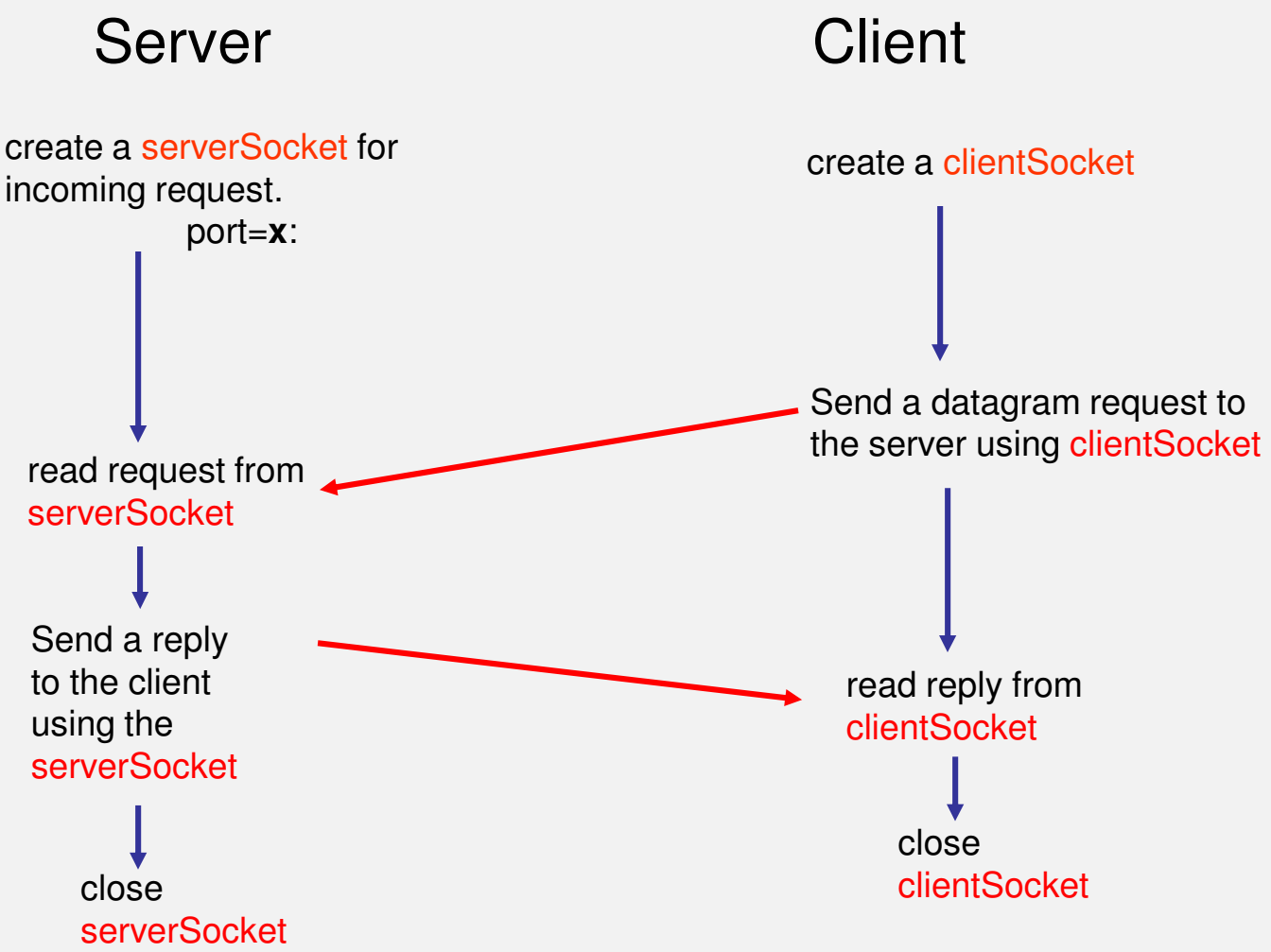
Read data from the socket

```
int  MAXLENGTH= 256;
byte[ ] buff = new byte[MAXLENGTH];
try {
        InputStream in = clientSocket.getInputStream();
        in.read(buff);
    } catch (IOException e) {System.out.println(e); }
```

Close socket

```
try {
 clientSocket.close(); // Close the socket and its streams
 } catch (IOException e) {…}
```

# UDP client/server implementation



# UDP client implementation

Create the clientSocket

```
try {
   DatagramSocket clientSocket = new DatagramSocket();
} catch (IOException e) { System.out.println(e); }
```

Send a datagram to the server

```
int  PACKETLENGTH= 256;
byte[ ] data = new byte[PACKETLENGTH];
try {
      DatagramPacket packet = new DatagramPacket(data,
         data.length, serverIP, serverPort);
      clientSocket.send(packet);
} catch (IOException e) {System.out.println(e); }
```

# UDP client implementation

Read the server reply

```
byte[ ] rcvData = new byte[PACKETLENGTH];
try {
     DatagramPacket receivePacket =
        new DatagramPacket (rcvData, rcvData.length);
    clientSocket.receive(rcvPacket);

    String rcvString = new String(rcvPacket.getData());
    System.out.println("The received packet is: "+rcvString);
} catch (IOException e) {System.out.println(e); }
```

Close socket

```
    try {
       clientSocket.close(); // Close the socket
     } catch (IOException e) {…}
```

# UDP server implementation

Open a socket

```
DatagramSocket server;
    try {
        serverSocket = new DatagramSocket (portNumber);
    } catch (IOException e) { System.out.println(e); }
```

Receive a datagram from client

```
byte[ ] buff = new byte[PACKETLENGTH];
try {
        DatagramPacket rcvPacket = new DatagramPacket
                (buff, buff.length);
        server.receive(rcvPacket );
} catch (IOException e) { .. }
```

# UDP server implementation

Get the client IP and Port

```
InetAddress clientIP = rcvPacket.getAddress();
int clientPort = rcvPacket.getPort();
```

Send a datagram to the client

```
String data = "Hello from server";
try {
        DatagramPacket sendPacket = new DatagramPacket
                (sendData, sendData.length,  clientIP, clientPort );
        serverSocket.send(sendPacket);
  } catch (IOException e) {…}
```

# Sending and receiving data

- Communication protocol
  - Message encoding and decoding
  - Framing
- Example

# Sending and receiving data

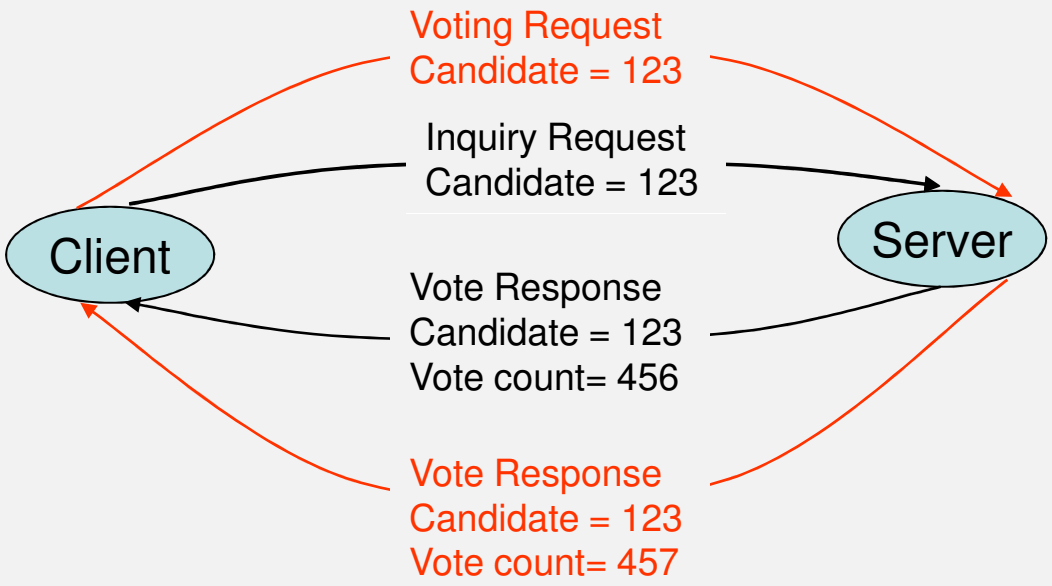
- Sender and receiver must agree on the communication protocol
  - How the exchanged information will be *encoded* (represented as a sequence of bits)
    - How the sequence of bits is arranged by the sender and interpreted, or *parsed*, by the receiver?
    - Framing: refers to the problem of enabling the receiver to locate the beginning and end of a message.
  - Which program sends what information and when
  - How the received information affects the behavior of the program.

# Sending and receiving data

- Framing:
  - Possible problems:
    - *Deadlock*
    - Protocol errors
  - Two general techniques enable a receiver to unambiguously find the end of the message:
    - *Delimiter-based*
      - E.g. end-of-stream indication, a particular character
    - *Explicit length*
- The same considerations apply to finding the boundaries of the individual *fields* of a given message

# Sending and receiving data

- **Example**
  - **Consider the following voting protocol**



# Sending and receiving data

1. Message representation:

```
public class VoteMsg {
        private boolean isInquiry; // true if inquiry; false if vote
        private boolean isResponse;// true if response from server
        private int candidateID; // in [0,1000]
        private long voteCount; // nonzero only in response
        …..
}
```

2. Message encoding and decoding

| Magic-string | Type [‘v’,’i’] | RespFlag [‘R’] | Candidate ID | Vote Count |
|---|---|---|---|---|

# Sending and receiving data

```
public class VoteMsgEncoder implements VoteMsgCoder {
        public static final String MAGIC = "Voting";
        public static final String VOTESTR = "v";
        public static final String INQSTR = "i";
        public static final String RESPONSESTR = "R";

        public static final String FIELDELIMSTR = " ";
        public static final int MAX_MSG_LENGTH = 2000;

        public byte[ ] encode(VoteMsg msg) throws IOException {
       String msgString = MAGIC + FIELDELIMSTR + (msg.isInquiry() ? INQSTR : VOTESTR)
              + FIELDELIMSTR + (msg.isResponse() ? RESPONSESTR + FIELDELIMSTR : "")
              + Integer.toString(msg.getCandidateID()) + FIELDELIMSTR
               + Long.toString(msg.getVoteCount());
      byte data[ ] = msgString.getBytes();
      return data;
        }
……….
```

# Sending and receiving data

```
public class VoteMsgDecoder implements VoteMsgCoder {
        boolean isInquiry;
                boolean isResponse;
                int candidateID;
                long voteCount;
        public VoteMsg decode(byte[ ] message) throws IOException
        {
          ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
           Scanner s = new Scanner(new InputStreamReader(msgStream));
           String token;
                                try {
                                 token = s.next();
                                 if (!token.equals(MAGIC)) {
                                    throw new IOException("Bad magic string: " + token); }
                                       token = s.next();
                                        if (token.equals(VOTESTR)) {
                                                    isInquiry = false;
                                             } else if (!token.equals(INQSTR)) {
                                                    throw new IOException("Bad vote/inq indicator:
                                          " + token);
                                             } else {
```

Check if the message stars with the magic word

Check if it is a vote message

| Magic-string | Type [‘v’,’i’] | RespFlag [‘R’] | Candidate ID | Vote Count |
|---|---|---|---|---|

# Sending and receiving data

Check if it is a response

Get information from the message

Create and return a vote message

```
            token = s.next();
            if (token.equals(RESPONSESTR)) {
                    isResponse = true;
                    token = s.next();
            } else {
                    isResponse = false;
            }
            candidateID = Integer.parseInt(token);
            if (isResponse) {
                    token = s.next();
                    voteCount = Long.parseLong(token);
            } else {
                    voteCount = 0;
            }
          } catch (IOException ioe) {
            throw new IOException("Parse error...");
          }
          return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
        }

    }
```

| Magic-string | Type [‘v’,’i’] | RespFlag [‘R’] | Candidate ID | Vote Count |
|---|---|---|---|---|

# Sending and receiving data

## 3. Framing

Define a message framing class, which implements a delimiter-based framing using the “newline” character ("\n").

```
public class DelimFramer {
        private InputStream in; // data source
        private static final byte DELIMITER = "\n";
                                        // message delimiter

        public void frameMsg(byte[] message, OutputStream out) throws IOException {

            for (byte b : message) {
                if (b == DELIMITER) {
                    throw new IOException("Message contains delimiter"); }
            }
            out.write(message);
            out.write(DELIMITER);
            out.flush();
    }
```

ensure that the message does not contain the delimiter

# Sending and receiving data

Define a message framing class, which implements delimiter-based framing using the “newline” character ("\n").

```
public byte[] readNextMsg() throws IOException {
   ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
    int nextByte;

    while ((nextByte = in.read()) != DELIMITER) {
        if (nextByte == -1) { // end of stream?
              if (messageBuffer.size() == 0) { // if no   byte read
                        return null;
              } else { // if bytes followed by end of stream:  framing error
             throw new EOFException("Non-empty message without
                                                    delimiter");}
        }
        messageBuffer.write(nextByte);
    }
    return messageBuffer.toByteArray();
     ..............
}
```

fetch bytes until the delimiter is found

Write the current byte to the buffer

Return the message

- References
  - TCP/IP Sockets in Java: Practical Guide for Programmers, Second Edition, Kenneth L. Calvert and Michael J. Donahoo, ISBN: 978-0-12-374255-1
  - "All About Sockets" http://java.sun.com/docs/books/tutorial/networking/sockets/