Lecture 3 : Introduction to Socket Programming

A socket is an endpoint for communication between two machines over a network. It provides an interface for applications to send and receive data using network protocols like TCP or UDP.

What is a socket?

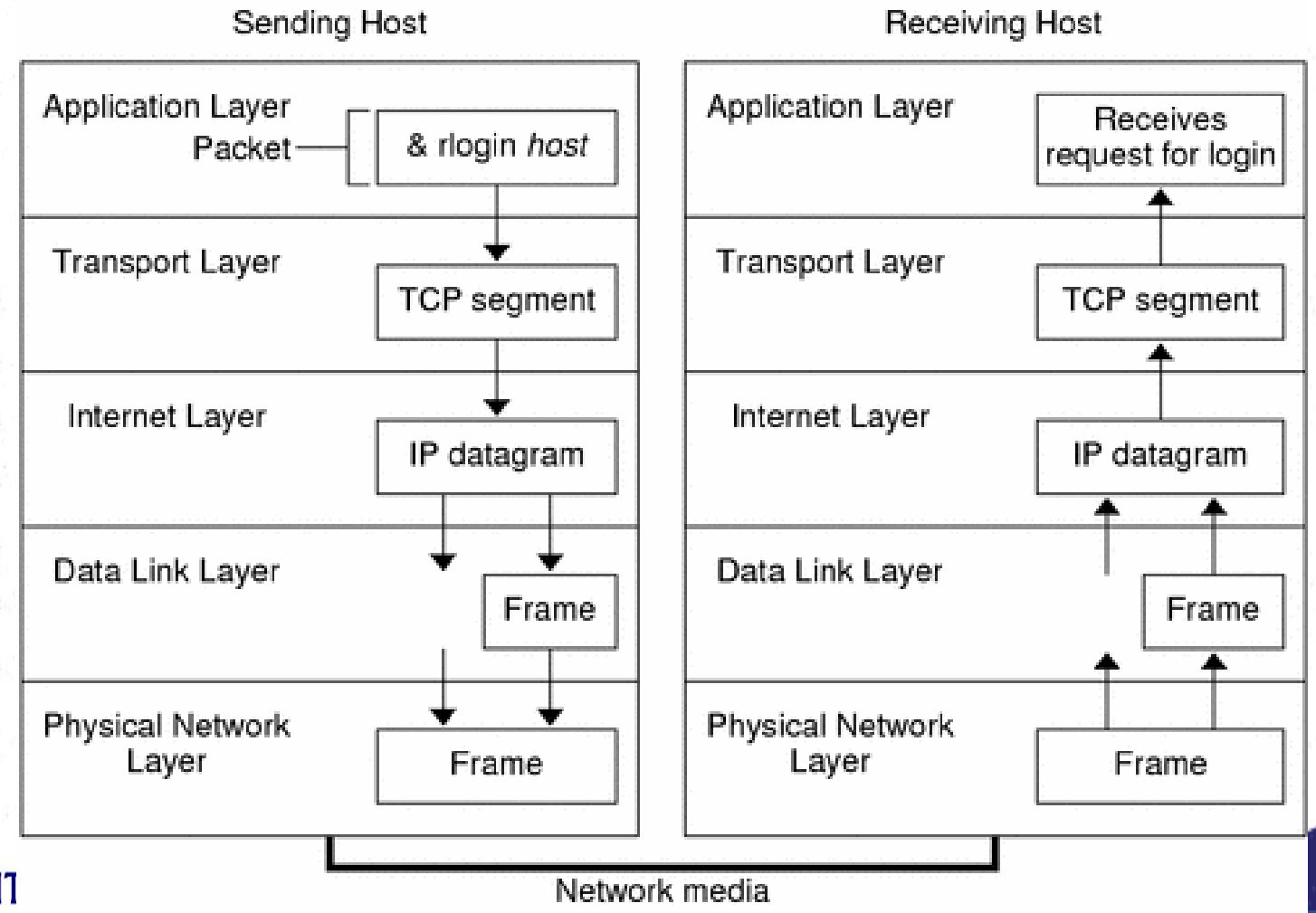
- Port vs. Socket

Aspect	Port	Socket
Definition	A numerical identifier for a specific service or process on a device.	A combination of an **IP address** and a **port number** that uniquely identifies a communication endpoint.
Purpose	Helps differentiate services running on the same device (e.g., HTTP on port 80, HTTPS on port 443).	Establishes a connection between two devices for data exchange over a network.
Scope	Exists within a system (e.g., a server has multiple ports for different applications).	Exists between two devices, uniquely identifying a communication session.
Dependency	A port exists independently and can be used by multiple sockets.	A socket requires a port to function.
Format	A single number (e.g., 80, 443, 3000).	IP address + Port (e.g., `192.168.1.10:8080`).
Example	Port 22 (SSH), Port 80 (HTTP), Port 443 (HTTPS).	A client connecting to `192.168.1.5:5000` creates a socket for data transfer.

- An interface between application and network
  - The application creates a socket
  - The socket *type* dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless

- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

TCP/IP Stack


Sending Host
Receiving Host
Application Layer
Packet
& rlogin *host*
Application Layer
Receives request for login
Transport Layer
TCP segment
Transport Layer
TCP segment
Internet Layer
IP datagram
Internet Layer
IP datagram
Data Link Layer
Frame
Data Link Layer
Frame
Physical Network Layer
Frame
Physical Network Layer
Frame
Network media

Two essential types of sockets

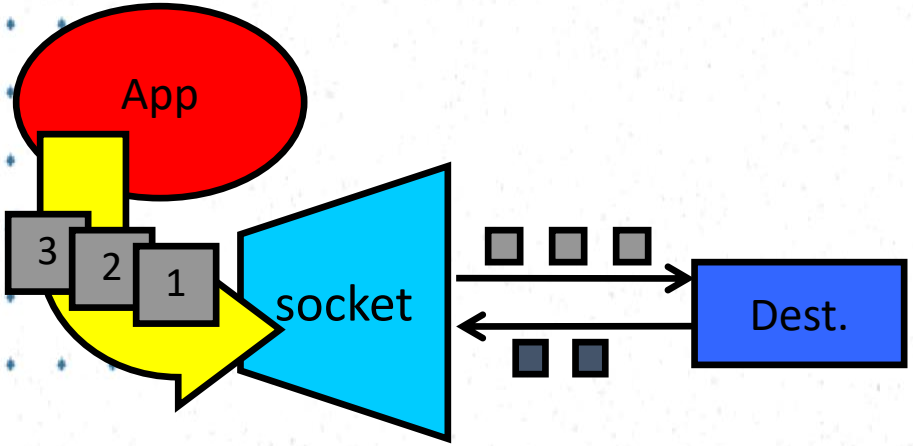
- TCP Socket
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional

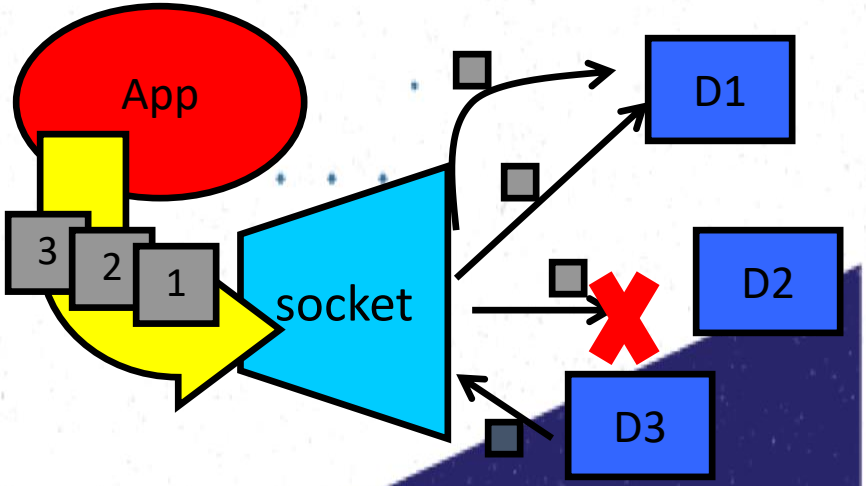
ex - SMTP

- UDP Socket
  - unreliable delivery

can be packet loss

  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive





Applications

- TCP (Transmission control protocol)
  - Point to point chat applications, File transfer (FTP), Email (SMTP)
  - Used when there's a requirement for guaranteed delivery

- UDP (User datagram protocol)
  - Streaming, Multicast/Broadcast
  - Useful when the speed of more important than the assurance of delivery

A Socket-eye view of the Internet

medellin.cs.columbia.edu

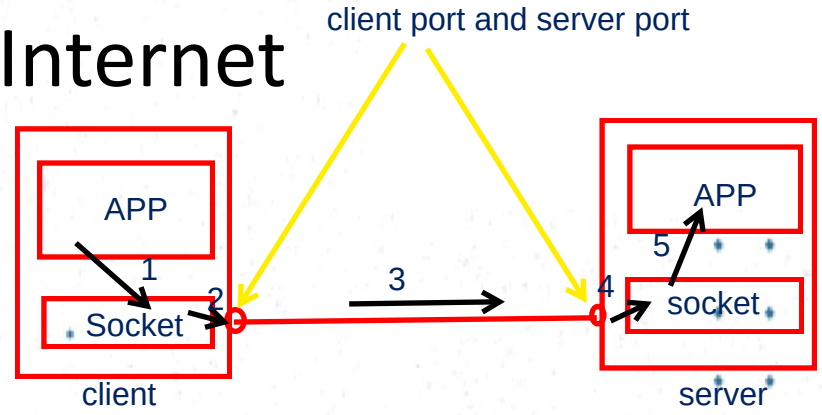
(128.59.21.14)

newworld.cs.umass.edu

(128.119.245.93)

cluster.cs.columbia.edu

(128.59.21.14, 128.59.16.7, 128.59.16.5, 128.59.16.4)

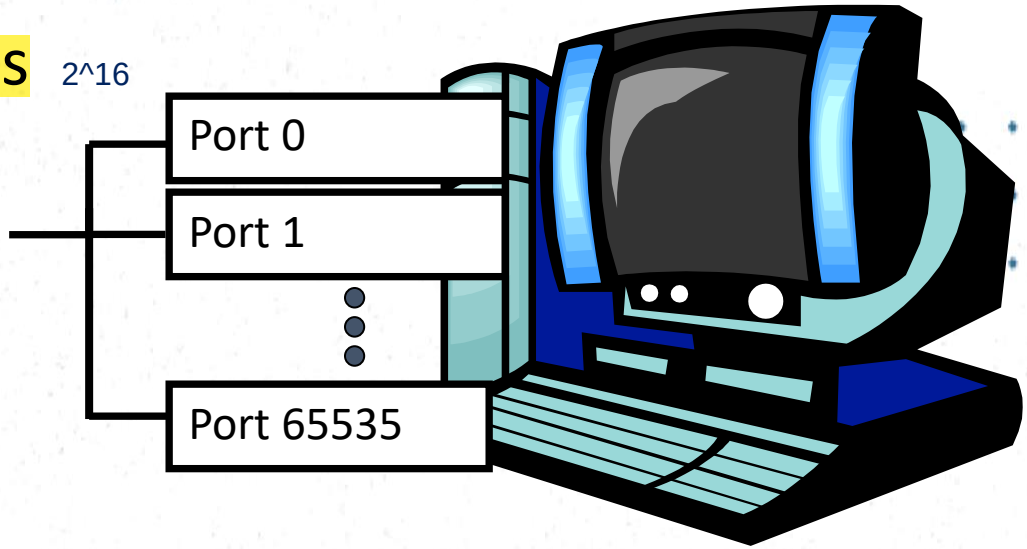


- Each host machine has an IP address
- When a packet arrives at a host

Ports

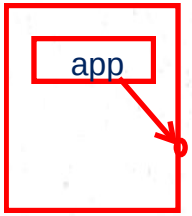
ports use for communicate between rest of the network

- Each host has 65,536 ports 2^16
- Some ports are *reserved for specific apps*
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)
    for well known applications



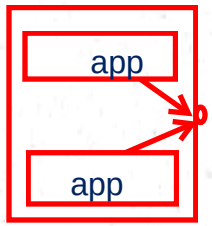
A socket provides an interface to send data to/from the network through a port

Addresses, Ports and Sockets



- In TCP, only one application (process) can listen to a port
- In UDP Multiple applications (processes) may listen to incoming messages on a single port

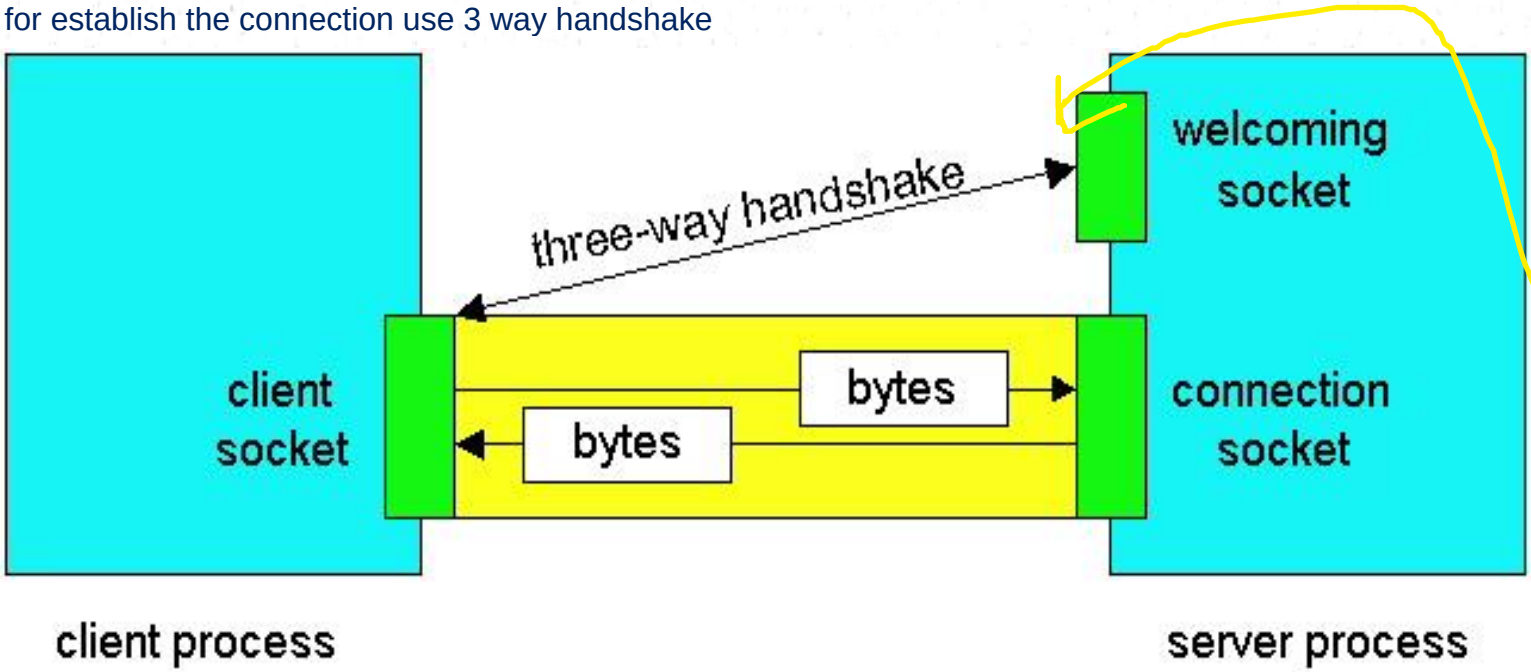
broadcast based



- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - Each family (process) of the apartment complex (computer) communicates with some same mailbox (port)

TCP Sockets

for establish the connection use 3 way handshake

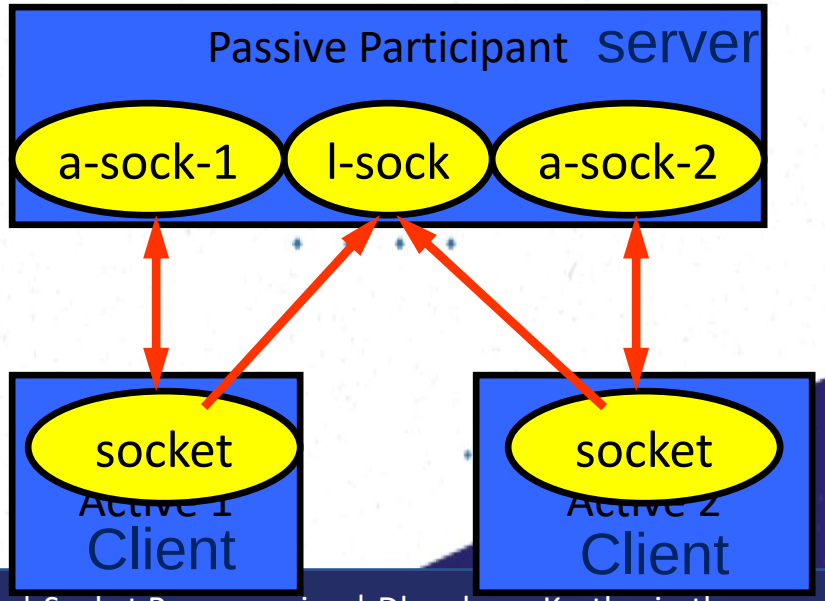


new client connect in the tcp protocol, initial part is 3 way handshake use for establish connection

Client socket, welcoming socket (passive) and connection socket (active)

Connection setup

- Passive participant Server
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- Active participant client
  - step 2: request & establish connection
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants



Dealing with blocking

- Calls to sockets can be blocking (no other client may be able to connect to the server)
- Can be resolved using multi-threaded programming
- Start a new thread for every incoming connection

Dealing with Blocking in Socket Programming

Problem with Blocking Calls

- **Blocking Sockets**: When a server accepts a connection, it may block (wait) until the operation is completed.
- This prevents other clients from connecting while the server is handling a request.

Solution: Multi-Threading

To allow multiple clients to connect simultaneously, **multi-threading** is used:

Approach	Description
Blocking (Single-threaded)	The server handles only one client at a time, blocking other connections.
Multi-threading	A new thread is created for each incoming connection, allowing multiple clients to be served simultaneously.
Non-blocking (Asynchronous I/O)	Uses non-blocking sockets or event-driven programming (e.g., Node.js, Python's `asyncio`).

Java Sockets Programming

- The package java.net provides support for sockets programming (and more).

- Typically you import everything defined in this package with:

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

Classes

`InetAddress`	Identify URI. handle DNS
`Socket`	
`ServerSocket`	
`DatagramSocket`	handle UDP communication
`DatagramPacket`	UDP packet

an IP address (both IPv4 and IPv6). It provides static methods to retrieve information about hosts and network addresses.

InetAddress class

- Static methods you can use to create new InetAddress objects.
  - getByName(String host) Returns an InetAddress object for the specified hostname or IP address.
  - getAllByName(String host) Returns an array of InetAddress objects for all IP addresses associated with a hostname.
  - getLocalHost() Returns the local machine's InetAddress.

```
InetAddress x = InetAddress.getByName(
                                "cse.unr.edu");
```

❖Throws **UnknownHostException**

```
try {

  InetAddress a = InetAddress.getByName(hostname);

  System.out.println(hostname + ":" +
      a.getHostAddress());

} catch (UnknownHostException e) {

  System.out.println("No address found for " +
        hostname);

}
```

host's IP address

Socket class

The Socket class in Java is used for active TCP connections (client-side or accepted connections from a server).

- Corresponds to active TCP sockets only!
  - client sockets
  - socket returned by accept();
- Passive sockets are supported by a different class:
  - ServerSocket
- UDP sockets are supported by
  - DatagramSocket

Key Classes for Socket Communication

Class	Description
`Socket`	Represents an **active** TCP socket (client-side or accepted server connection).
`ServerSocket`	Used for **passive** listening on a port to accept incoming TCP connections.
`DatagramSocket`	Supports **UDP** communication (connectionless, best-effort delivery).

JAVA TCP Sockets

- java.net.Socket
  - Implements client sockets (also called just "sockets").
  - An endpoint for communication between two machines.
  - Uses input/output streams to pass messages
    -

- java.net.ServerSocket
  - Implements server sockets.
  - Waits for requests to come in over the network.
  - Accepts the client connection requests
  - Performs some operation based on each request

Socket Constructors

- Constructor creates a TCP connection to a named TCP server.
  - There are a number of constructors:

```
Socket(InetAddress server, int port);
```

Connects to a remote server using an InetAddress and port number.

```
Socket(InetAddress server, int port,
       InetAddress local, int localport);
```

Connects to a remote server while binding the socket to a specific local IP and port.

```
Socket(String hostname, int port);
```

Connects to a remote server using a hostname (e.g., "example.com") and port number.

Socket Methods

Method	Description
`void close()`	Closes the socket connection.
`InetAddress getInetAddress()`	Returns the remote IP address connected to this socket.
`InetAddress getLocalAddress()`	Returns the local IP address of this socket.
`InputStream getInputStream()`	Returns an `InputStream` for reading data from the socket.
`OutputStream getOutputStream()`	Returns an `OutputStream` for sending data through the socket.
`boolean isClosed()`	Returns `true` if the socket is closed.
`boolean isConnected()`	Returns `true` if the socket is successfully connected.
`void setSoTimeout(int timeout)`	Sets a timeout (in milliseconds) for socket read operations.
`int getPort()`	Returns the remote port number to which the socket is connected.
`int getLocalPort()`	Returns the local port number this socket is bound to.

Socket I/O

- Socket I/O is based on the Java I/O support
  - in the package **`java.io`**
- InputStream and OutputStream are abstract classes
  - common operations defined for all kinds of InputStreams, OutputStreams…

InputStream Basics

```
// reads some number of bytes and
// puts in buffer array b
int read(byte[] b);

// reads up to len bytes
int read(byte[] b, int off, int len);
```

Both methods can throw **`IOException`**.

Both return –1 on EOF.

Returns the number of bytes read or -1 at EOF.

OutputStream Basics

```
// writes b.length bytes
void write(byte[] b);

// writes len bytes starting
// at offset off
void write(byte[] b, int off, int len);
```

Both methods can throw **IOException**.

ServerSocket Class (TCP Passive Socket)

- Constructors:

Constructor	Description
`ServerSocket(int port)`	Creates a server socket that listens on the specified `port`.
`ServerSocket(int port, int backlog)`	Creates a server socket with a maximum queue length (`backlog`) for incoming connections.
`ServerSocket(int port, int backlog, InetAddress bindAddr)`	Binds the server to a specific `InetAddress` and port.

✅ The `backlog` parameter specifies how many pending connections can wait before being refused.

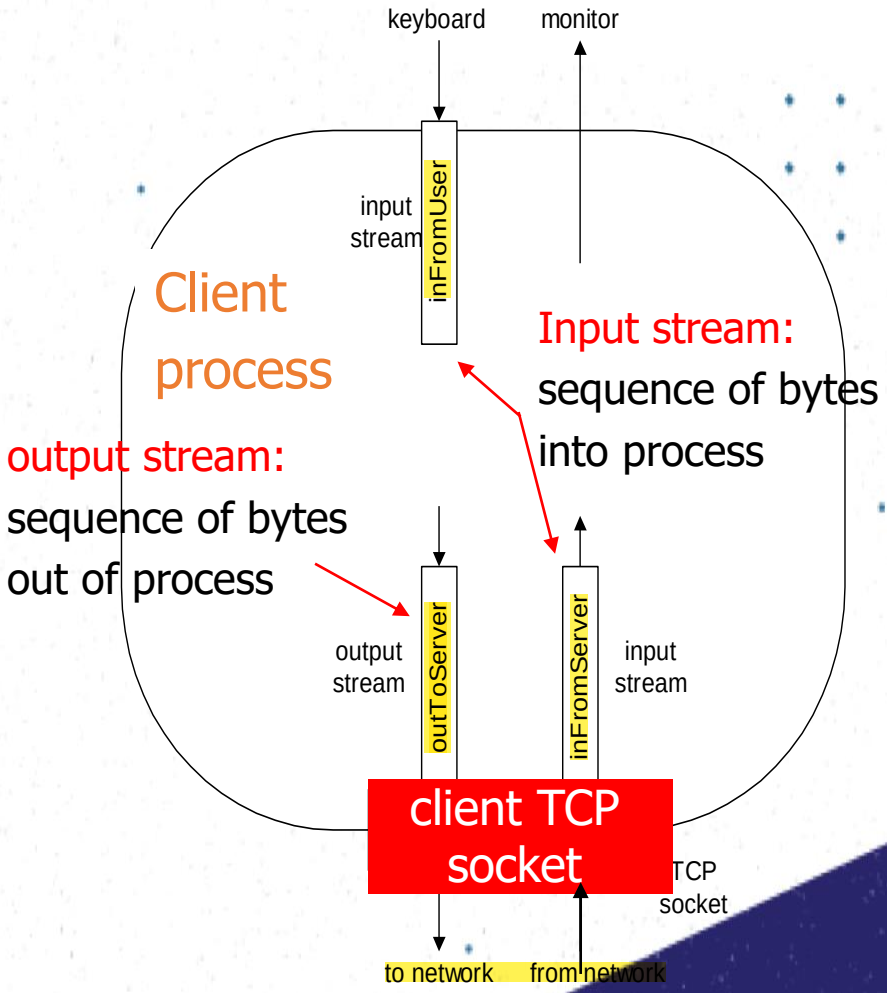
ServerSocket Methods

Method	Description	Exceptions Thrown
`Socket accept()`	Listens for an incoming connection and returns a new `Socket` object when a client connects. **(Blocking call)**	`IOException`
`void close()`	Closes the `ServerSocket`, stopping it from accepting new connections.	`IOException`
`InetAddress getInetAddress()`	Returns the IP address the server is bound to.	None
`int getLocalPort()`	Returns the port number the server is listening on.	None

Socket programming with TCP

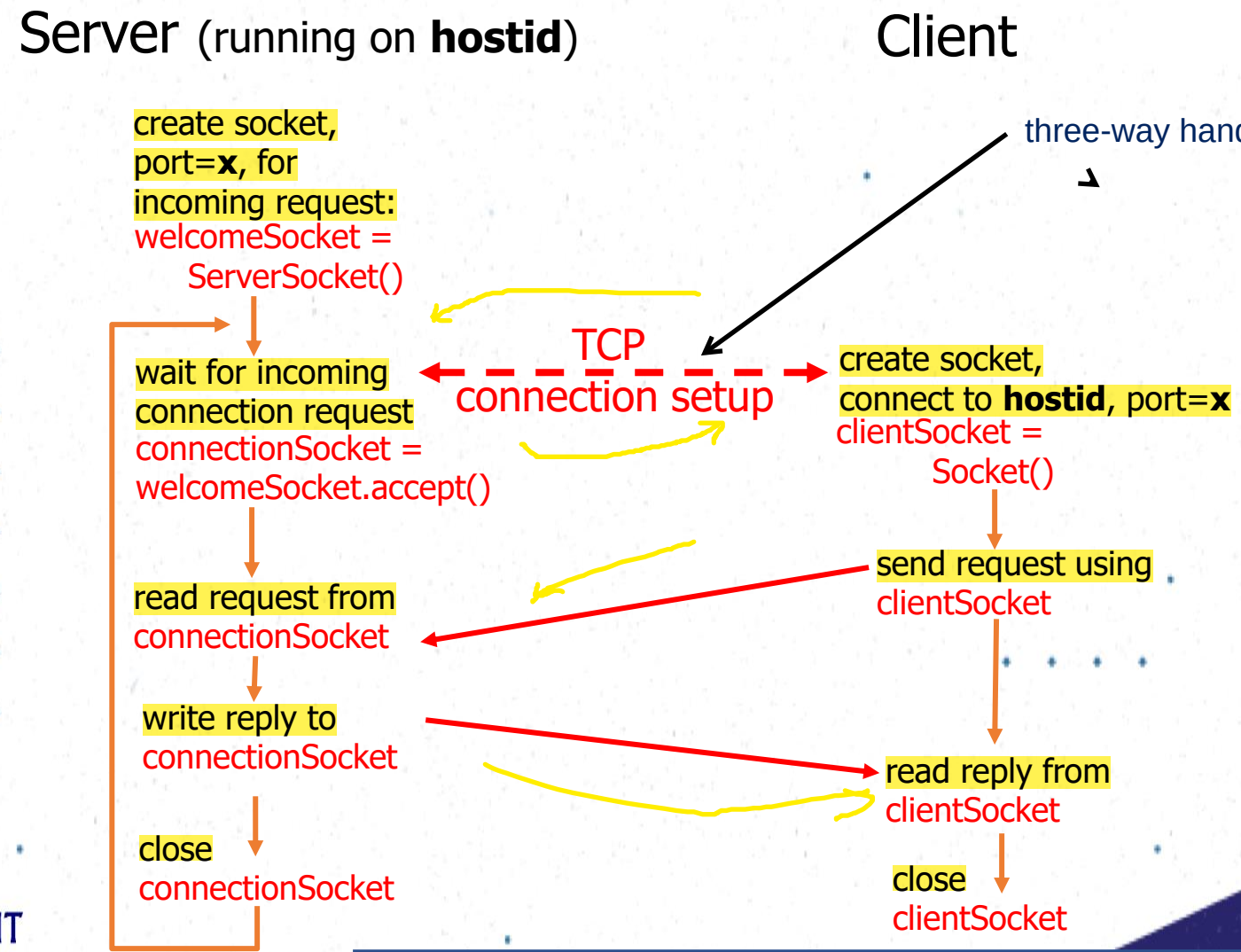
Example client-server app:

- client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (**inFromServer** stream)



By default, TCP socket operations (e.g., accept(), read()) block execution until completed.
Disadvantage: A single-threaded server might become unresponsive while waiting for data.
Solution: Use multi-threading or non-blocking I/O (NIO).

Client/server socket interaction: TCP



TCPClient.java

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

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

      BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

      Socket clientSocket = new Socket("hostname", 6789);

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

TCPClient.java

```
        BufferedReader inFromServer =
          new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();

outToServer.writeBytes(sentence + '\n');

modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

  }
}
```

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) {
                              waiting for incoming connections
           Socket connectionSocket = welcomeSocket.accept();

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

TCPServer.java

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

    clientSentence = inFromClient.readLine();

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

    outToClient.writeBytes(capitalizedSentence);

        }
    }
}
```

UDP Sockets

- DatagramSocket class

- DatagramPacket class needed to specify the payload
  - incoming or outgoing

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.

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()

DatagramSocket Constructors

Constructor	Description	Exceptions Thrown
`DatagramSocket()`	Creates a UDP socket bound to any available port.	`SocketException`, `SecurityException`
`DatagramSocket(int port)`	Creates a UDP socket bound to the specified **port**.	`SocketException`, `SecurityException`
`DatagramSocket(int port, InetAddress a)`	Binds the socket to a specific **port** and **IP address**.	`SocketException`, `SecurityException`

Datagram Methods

Method	Description
`void connect(InetAddress address, int port);`	Connects the socket to a specific **IP address** and **port** (optional for UDP).
`void close();`	Closes the socket and releases resources.
`void receive(DatagramPacket p);`	Receives an incoming UDP packet and stores it in a `DatagramPacket`.
`void send(DatagramPacket p);`	Sends a `DatagramPacket` over the network.

Lots more!

Datagram Packet

- Contain the payload
  - a byte array

- Can also be used to specify the destination address
  - when not using connected mode UDP

DatagramPacket Constructors

For receiving:

```
DatagramPacket( byte[] buf, int len);
```

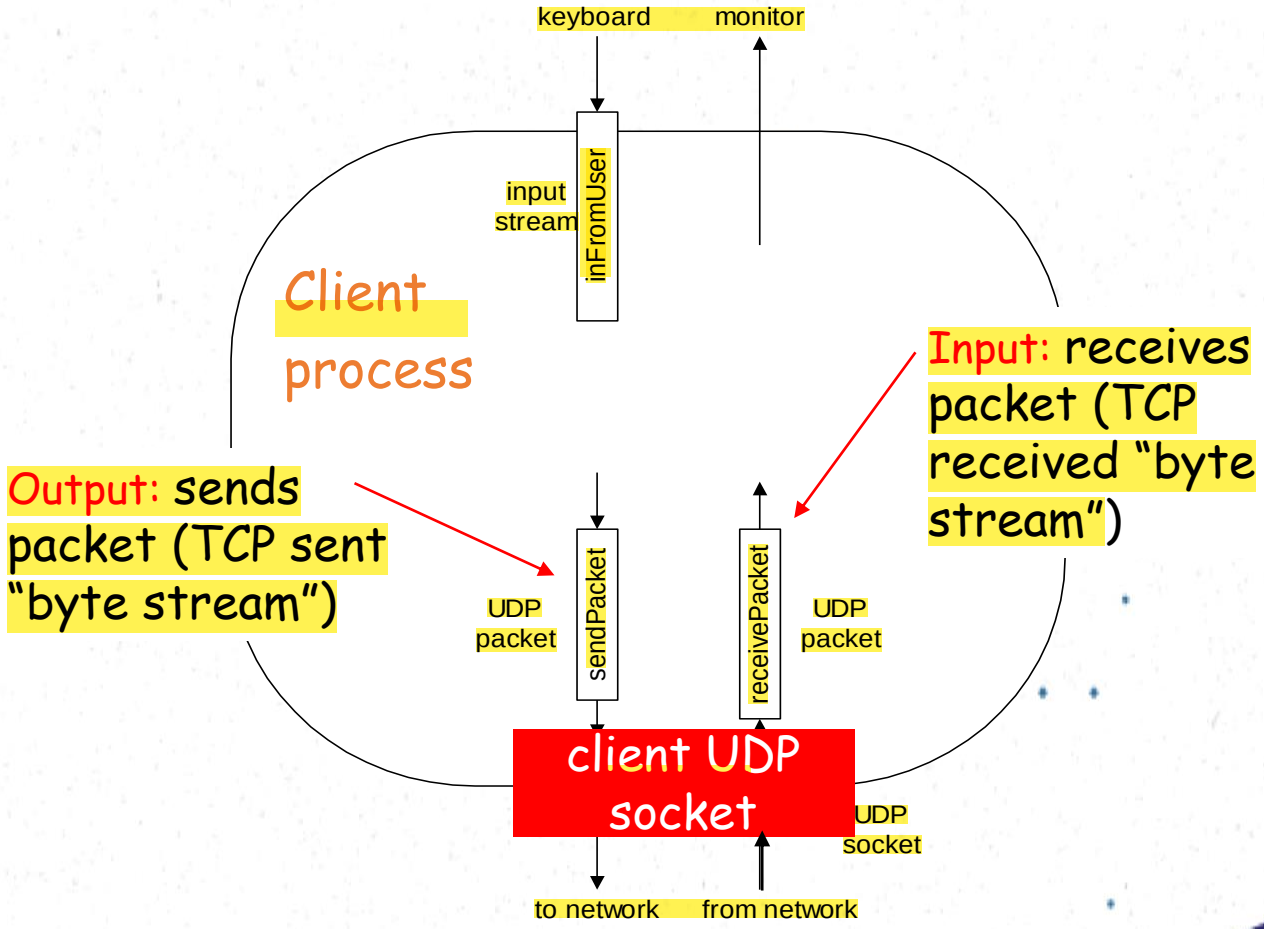
For sending:

```
DatagramPacket( byte[] buf, int len
                InetAddress a, int port);
```

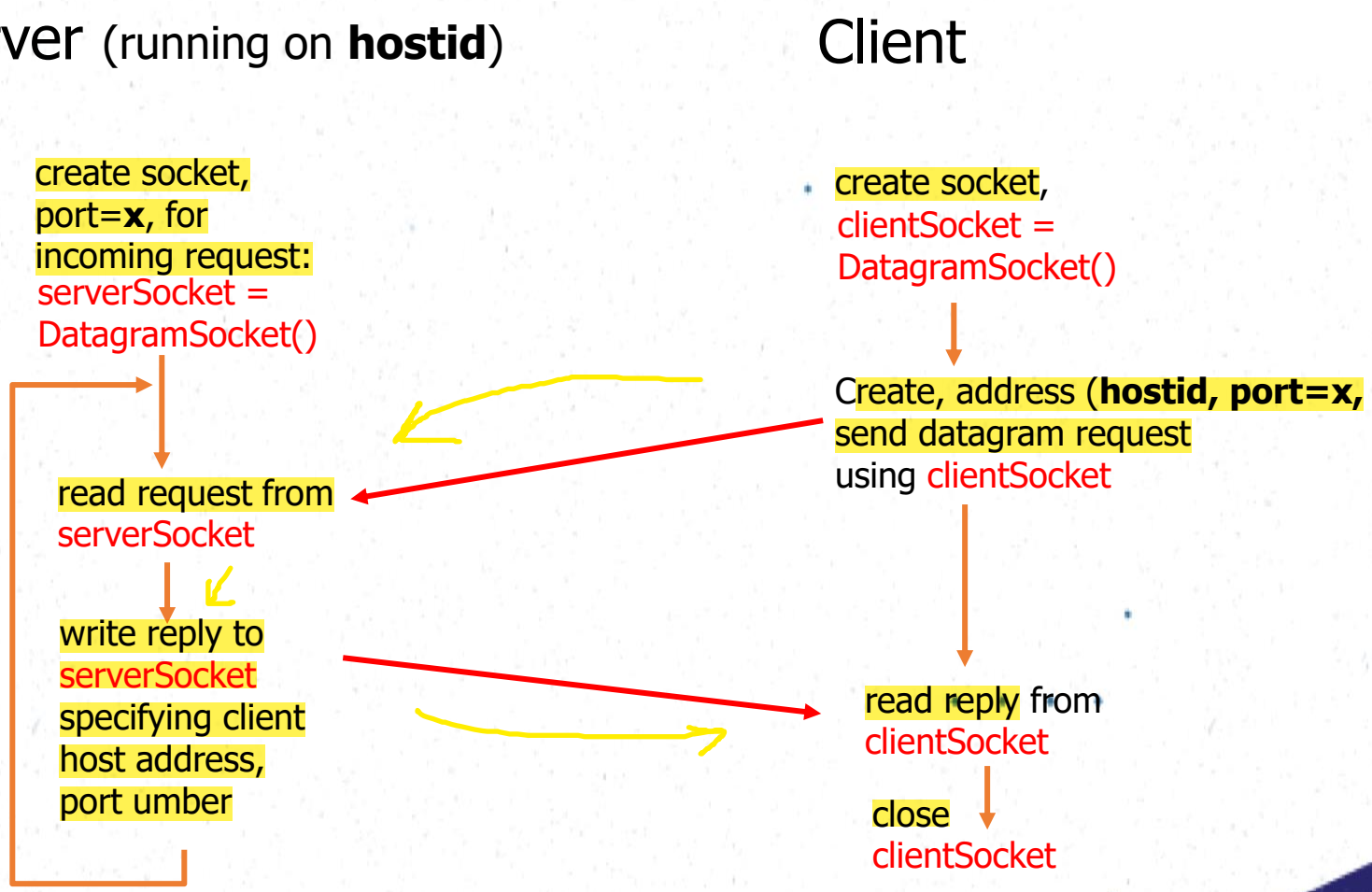
DatagramPacket methods

```
byte[] getData();
void setData(byte[] buf);

void setAddress(InetAddress a);
void setPort(int port);

InetAddress getAddress();
int getPort();
```

Example: Java client (UDP)



Client/server socket interaction: UDP



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();
```

UDPClient.java

```
    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();

    }
}
```

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());
```

UDPServer.java

```
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);
    }
  }
}
```

TCP vs HTTP

- https://netwo
  difference/

PARAMETER	TCP	HTTP
Acronym for	Transmission Control Protocol	Hypertext Transfer Protocol
OSI Layer	Transport Layer (Layer 4)	Application Layer (Layer 7)
Philosophy	TCP protocol is used for session establishment between two machine.	HTTP protocol is used for content access from web server.
TCP ports	No Port number	HTTP uses TCP's port number 80.
Authentication	TCP-AO (TCP Authentication Option)	HTTP does not perform authentication.
Usage	TCP is used extensively by many internet applications.	HTTP is useful in transferring smaller files like web pages.
State	Connection-Oriented Protocol	Stateless but not session less
Type of Transfer	Establishes Connection between Client and Server.	Transfers records between the Web client and Web server.
URL	No URL	When you are managing HTTP, HTTP will appear in URL.
Communication	3-Way Handshake (SYN, SYN-ACK, ACK)	One-way communication system.
Use	HTTP, HTTPs, FTP, SMTP, Telnet	Most widely used for web based applications
Download speed	The speed for TCP is slower.	HTTP is faster than TCP.

WebSockets

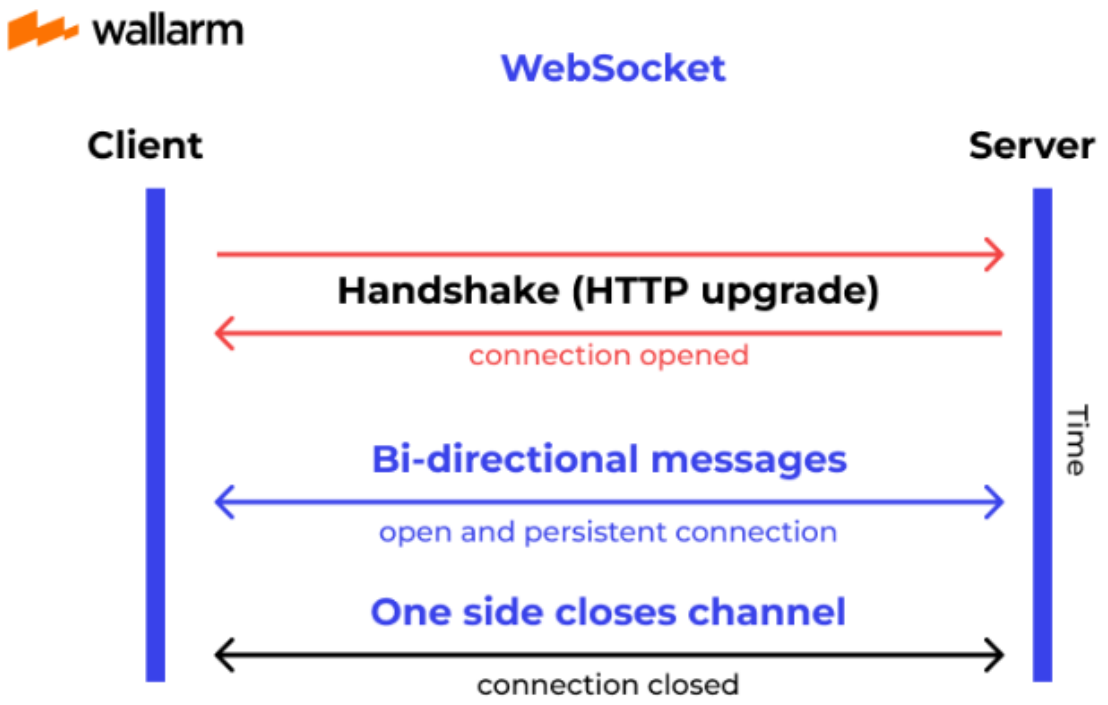
- https://www.wa
  explanation-of-

How do WebSockets work ?

As per the conventional definition, WebSocket is a duplex protocol used mainly in the client-server communication channel. It's bidirectional in nature which means communication happens to and fro between client-server.

The connection, developed using the WebSocket, lasts as long as any of the participating parties lays it off. Once one party breaks the connection, the second party won't be able to communicate as the connection breaks automatically at its front.

WebSocket need support from HTTP to initiate the connection. Speaking of its utility, it's the spine for modern web application development when seamless streaming of data and assorted unsynchronized traffic is concerned.



Summary

- Socket programming is the most fundamental form of Client-Server distributed computing available for app. developers
- Can be used to develop client-server distributed applications (e.g. Messaging applications)
- However, most real-world distributed systems use more high level distributed computing technologies (E.g. Web services, EJBs)
- Yet the underlying communication mechanism of these high level Dist. Computing frameworks is socket communication