CISC 435
Computer Networks
Network Programming

Amir Mohamad

Shadi Khalifa

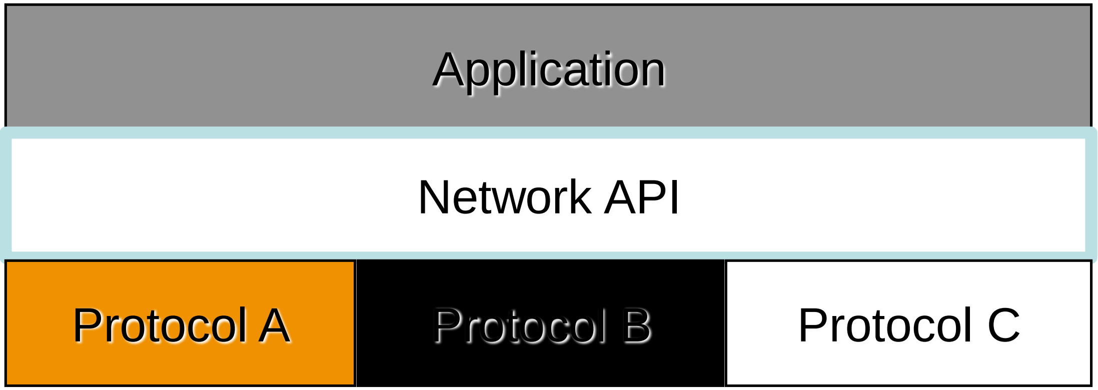
Acknowledgement: *This presentation contains some figures and text from: Computer networks a systems approach.* by Larry L. Peterson & Bruce S. Davie

Introduction to Network programming

- Implementing an application program on top of a network.
- Implementing the protocols running within the network.
- Network API and network protocols differences.

Implementing an Application

- Most network protocols are implemented in software
- Nearly all computer systems implement their network protocols as part of the OS.
- There is an interface called application programming interface (API) offers services provided by the OS to its network applications.



Socket Interface

- A socket is an abstract representation of a communication endpoint.

- The point where a local application process attaches to the network.

- The ***socket interface*** originally provided by the ***Berkeley distribution of Unix*** was widely used and ported to OS's other than its native system in 80's.
  - MacTCP was implemented for Mac OS in 1988.
  - Windows WinSock was provided in the 90's (1992).

- Uses existing I/O programming interface as much as possible.
  - Socket is considered as a file in reading from and writing to.

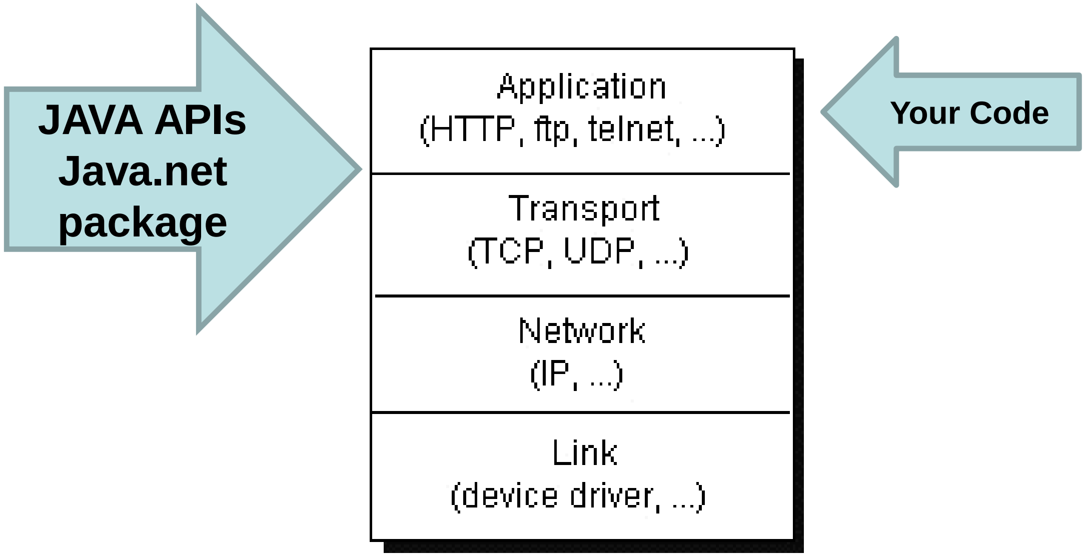
Socket Interface

- The interface defines operations for:
  - Creating a socket.
  - Attaching the socket to the network
  - Sending/receiving messages through the socket
  - Closing the socket.

- An example application will be given in the second part (client - server).

Networking basics

- Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP), as this diagram illustrates:



JAVA APIs
Java.net
package

Application
(HTTP, ftp, telnet, ...)

Transport
(TCP, UDP, ...)

Network
(IP, ...)

Link
(device driver, ...)

Your Code

TCP

- **TCP (Transmission Control Protocol)** is a
  - connection-based protocol
  - that provides a reliable flow of data between two computers.

- TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

- TCP provides a point-to-point channel for applications that require reliable communications.

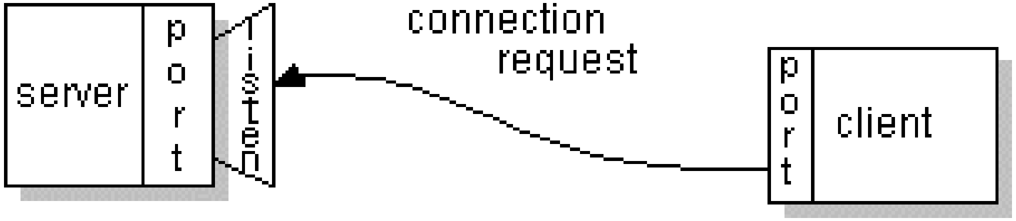
UDP

- **UDP (User Datagram Protocol)** is a protocol that
  - sends independent packets of data, called datagrams, from one computer to another
  - with no guarantees about arrival.
  - UDP is not connection-based like TCP.

- Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

Sockets

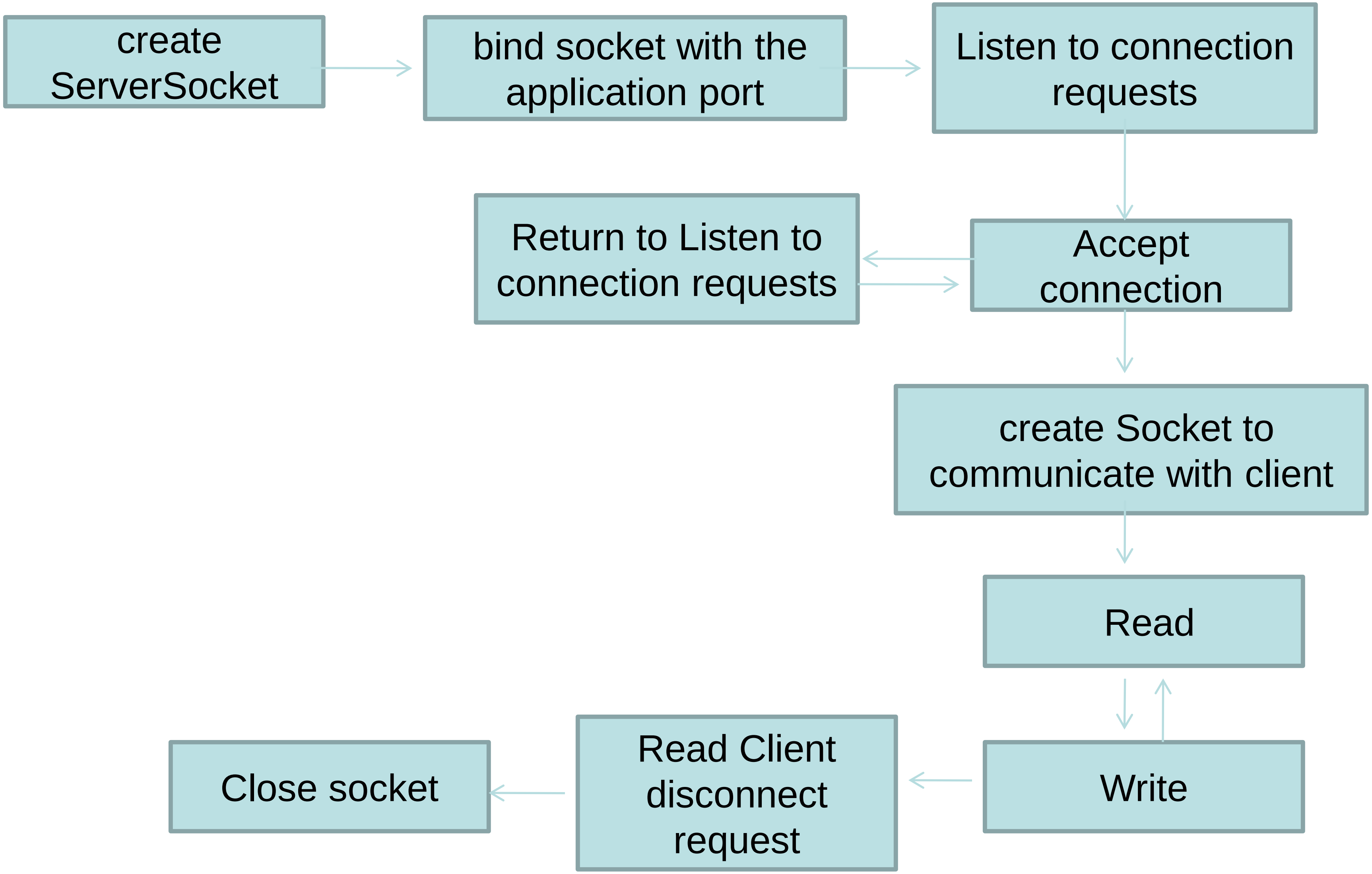
- A **socket** is one end-point of a two-way communication link between two programs running on the network.
- Socket classes are used to represent the connection between a client program and a server program. The *java.net* package provides two classes:
  - *Socket* : implement the client side of the connection.
  - *ServerSocket* : implement the server side of the connection.



TCP Connection

- Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the port for a client to make a connection request.
- To make a connection request, the client tries to rendezvous with the server using the server's IP and port.
- The client also needs to identify its’ application to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.
- If everything goes well, the server accepts the connection.
- Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client.
- It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.
- On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
- The client and server can now communicate by writing to or reading from their sockets.

Server Operation



create ServerSocket

bind socket with the application port

Listen to connection requests

Accept connection

Return to Listen to connection requests

create Socket to communicate with client

Read

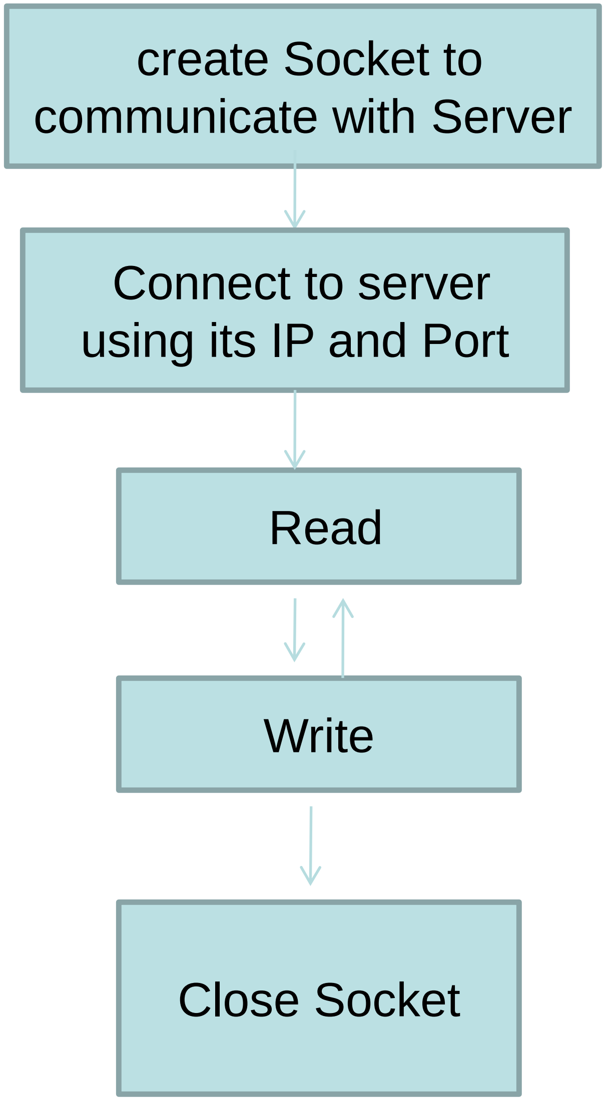
Write

Read Client disconnect request

Close socket

Client Operation

Client Operation

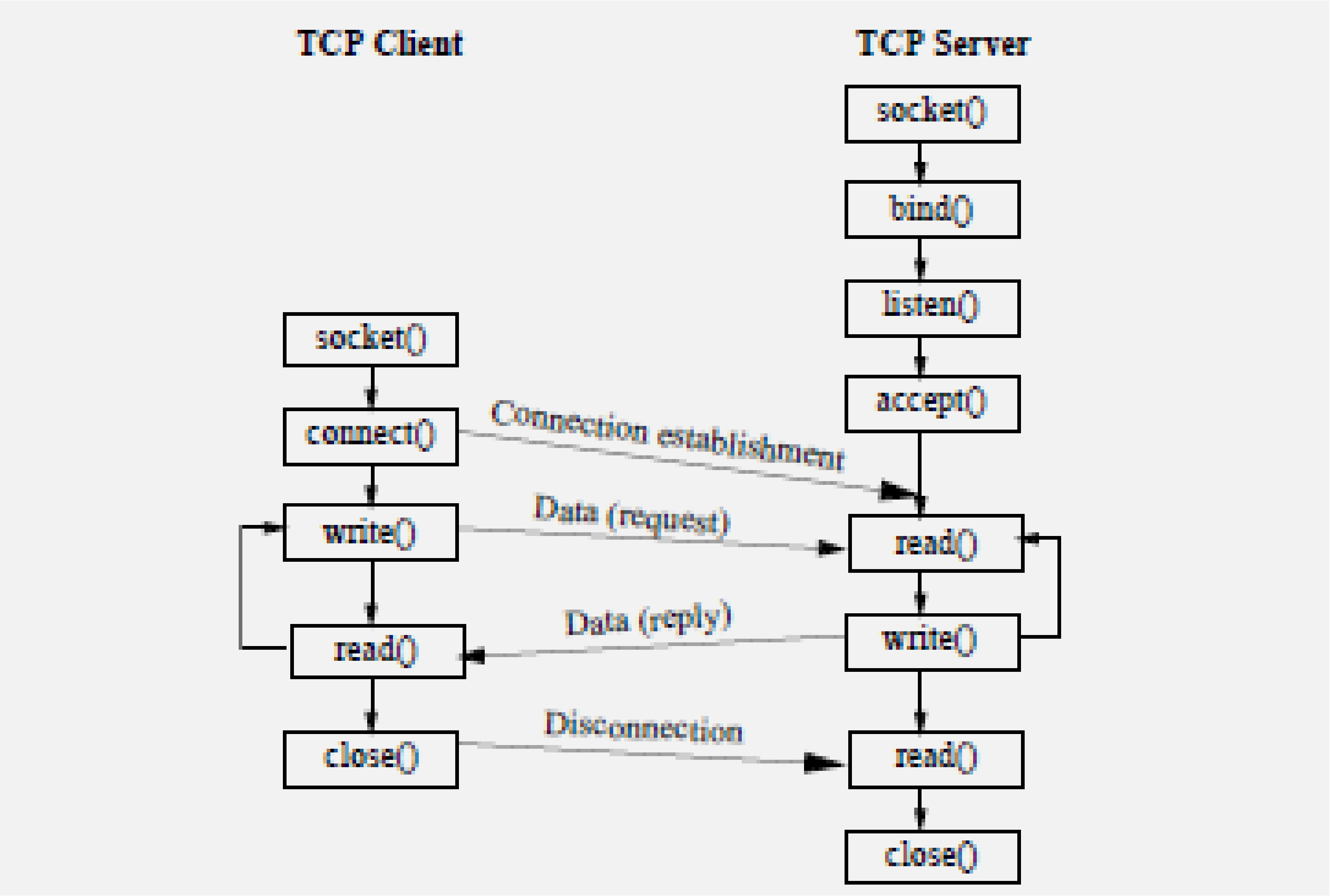

create Socket to communicate with Server

Connect to server using its IP and Port

Read

Write

Close Socket


TCP Client
TCP Server
socket()
bind()
listen()
accept()
socket()
connect()
Connection establishment
write()
Data (request)
read()
read()
Data (reply)
write()
close()
Disconnection
read()
close()

Java Server

- The server program is implemented by one classes:
  - XServer : which contains the *main* method for the server program and performs the work of listening to the port, establishing connections, and reading from and writing to the socket.

XServer Class

Any code/pseudo code listed below is a trial to help you get started. You should figure out what is appropriate to use in your project.

- Import
  - java.io.*;
  - java.net.*;

- ServerSocket:
  - ServerSocket serverSocket = new ServerSocket(port number the server is going to listen on);
  - Port number must not be already used by any other application

- Client Socket:
  - Socket clientSocket = serverSocket.accept();
  - The accept method waits until a client starts up and requests a connection on the host and port of this server.
  - When a connection is requested and successfully established, the accept method returns a new Socket object which is bound to the same local port and has its remote address and remote port set to that of the client.
  - The server can communicate with the client over this new Socket and continue to listen for client connection requests on the original ServerSocket

XServer Class

- PrintWriter to write to the client:
  - PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);

- BufferedReader to read from the client :
  - BufferedReader in = new BufferedReader(new InputStreamReader (clientSocket.getInputStream()));

XServer Class

- Read from Client and process the input
  - String data = protocol.processInput(in.readLine());
- write to Client
  - out.println(data);
  - out.flush();
- Closing the socket
  - Close PrintWriter
    - out.close();
  - Close BufferReader:
    - in.close();
  - Close Client Socket
    - clientSocket.close();
  - Close ServerSocket
    - serverSocket.close();

To summarize XServer Class

1. Open a ServerSocket.
2. Open one or more client Socket.
3. Bind client socket(s) to clients by accepeting their connection requests.
4. Open an input stream and output stream to the client socket.
5. Read from and write to the stream according to the server's protocol.
6. Close the streams.
7. Close the client socket(s).
8. Close the ServerSocket.

Java Client

- Import
  - java.io.*;
  - java.net.*;

- Socket:
  - Socket socket = new Socket(server IP or hostname, port number);

- PrintWriter to write to the server:
  - PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

- BufferedReader to read from the server :
  - BufferedReader in = new BufferedReader(new InputStreamReader (socket.getInputStream()));

Java Client

- write to server
  - out.println(“text sent to server”);
  - out.flush();
- Read from server
  - String data = in.readLine();
- Closing the socket
  - Close PrintWriter
    - out.close();
  - Close BufferReader:
    - in.close();
  - Close Socket
    - socket.close();

To Summarize JAVA Client

1. Open a socket.
2. Open an input stream and output stream to the socket.
3. Read from and write to the stream according to the server's protocol.
4. Close the streams.
5. Close the socket.

- **Note : Only step 3 differs from client to client, depending on the server. The other steps remain largely the same.**

Datagrams

- The UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another.

- A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

- In contrast of TCP, applications that communicate via datagrams send and receive completely independent packets of information. These clients and servers do not have and do not need a dedicated point-to-point channel. The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

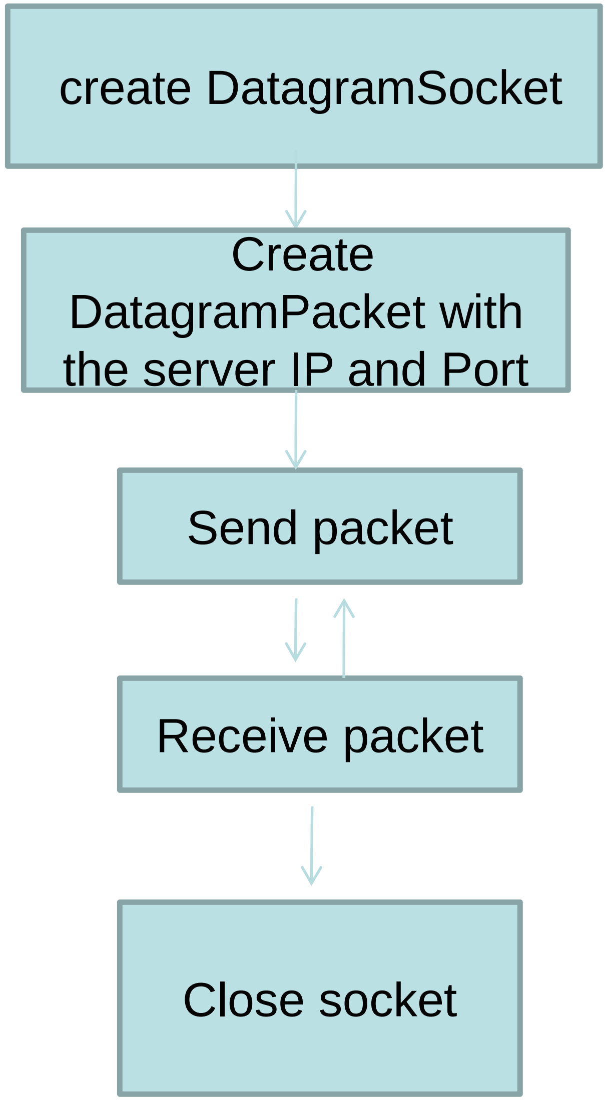
Java Datagrams

- The java.net package contains three classes to help you write Java programs that use datagrams to send and receive packets over the network:
  - ***DatagramSocket :***
    - A datagram socket is the sending or receiving point for a packet delivery service.
    - UDP broadcasts sends are always enabled on a DatagramSocket.
  - ***DatagramPacket*** :
    - Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another based solely on information contained within that packet.
  - ***MulticastSocket :***
    - A MulticastSocket is a (UDP) DatagramSocket used for sending and receiving IP multicast packets. .
    - A multicast group is specified by a **class D** IP address and by a standard UDP port number. Class D IP addresses are in the range **224.0.0.0 to 239.255.255.255**, inclusive. The address **224.0.0.0 is reserved** and should not be used.

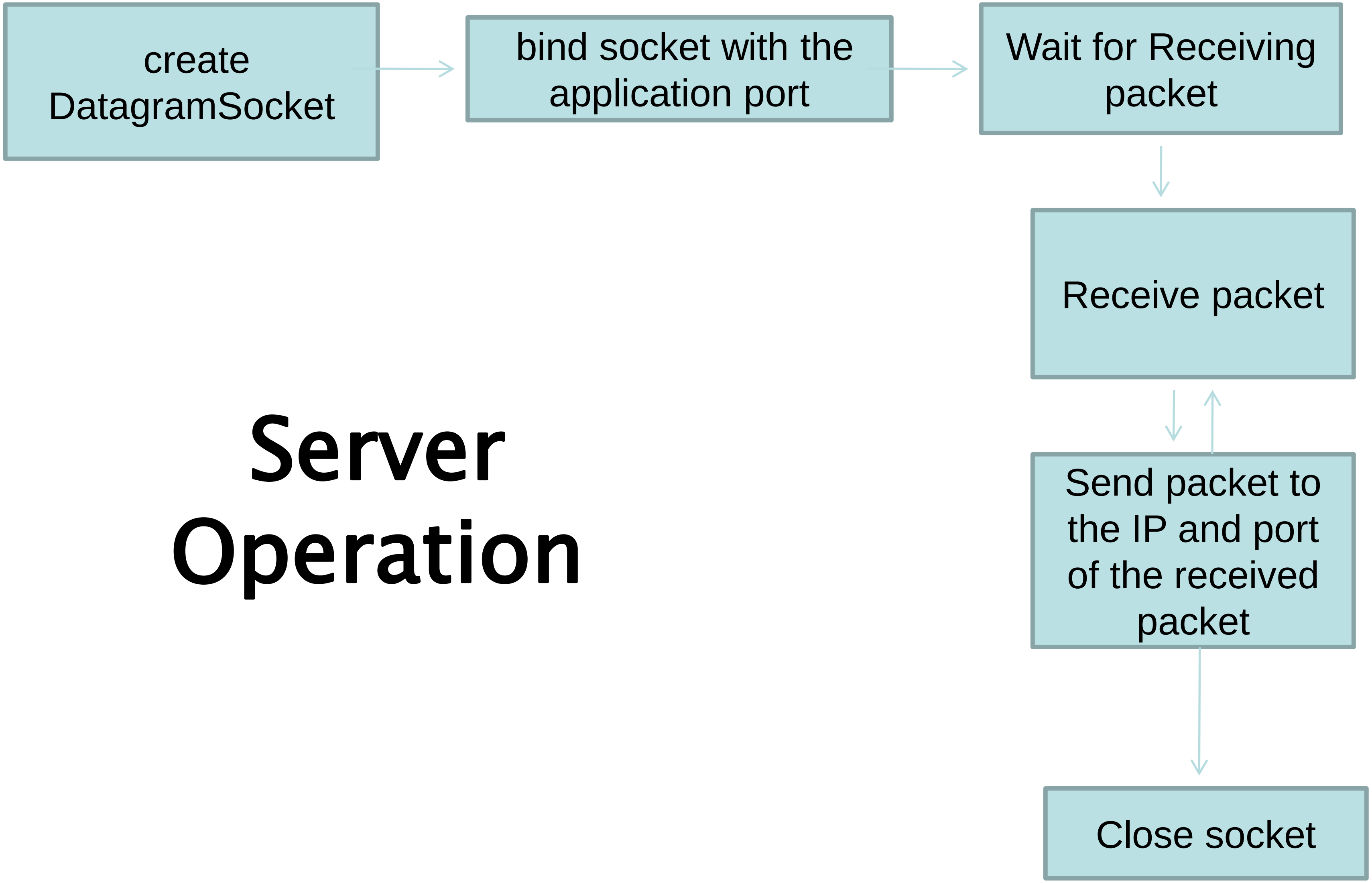
  - An application can send and receive *DatagramPackets* through a *DatagramSocket*.
  - In addition, *DatagramPackets* can be broadcast to multiple recipients all listening to a *MulticastSocket*.

Client Operation

Client Operation



Server Operation



create DatagramSocket

bind socket with the application port

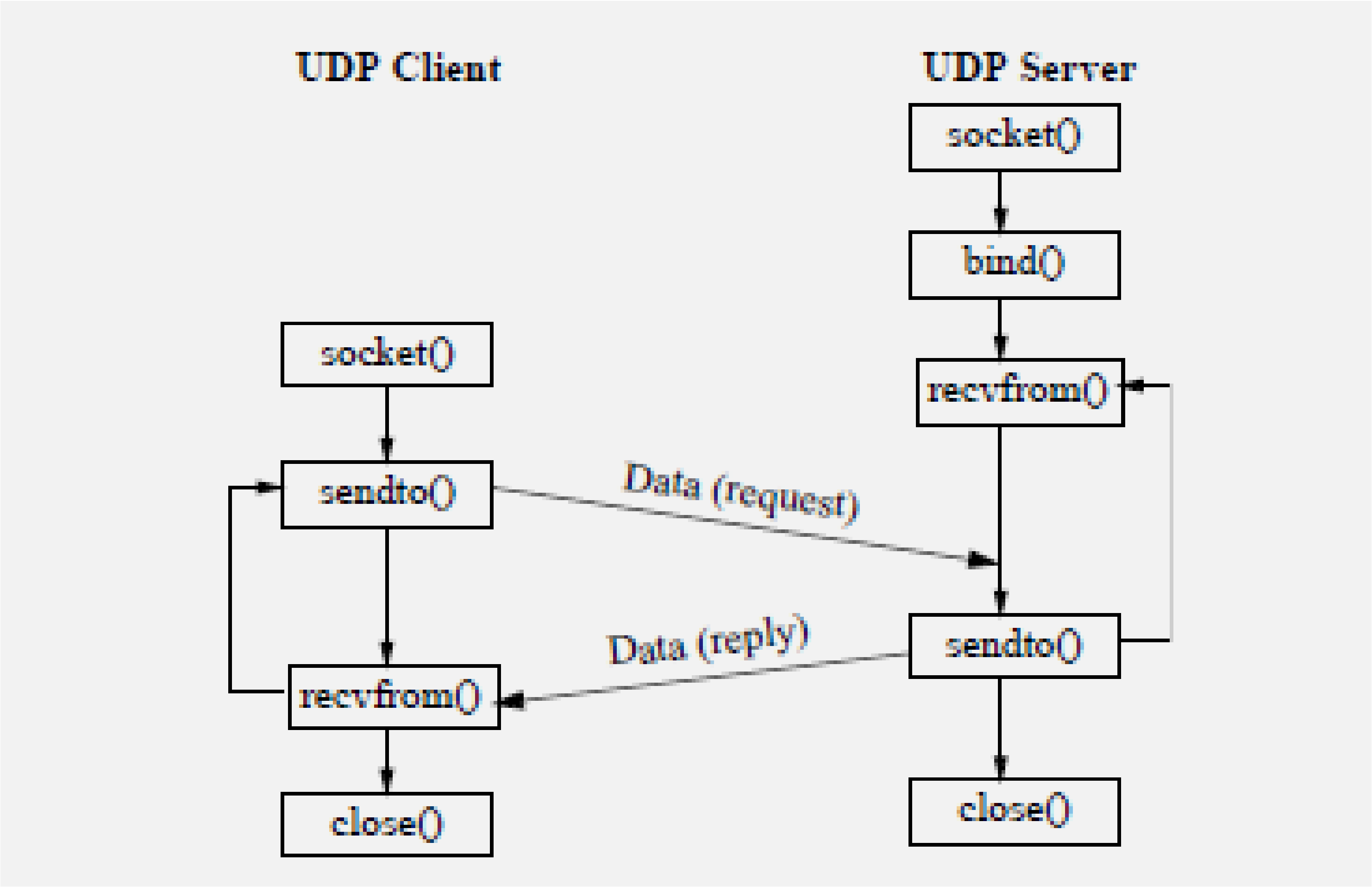
Wait for Receiving packet

Receive packet

Send packet to the IP and port of the received packet

Close socket

Server Operation


UDP Client
UDP Server
socket()
bind()
recvfrom()
socket()
sendto()
Data (request)
Data (reply)
sendto()
recvfrom()
close()
close()

Java Server

- The server program is implemented by one classes:
  - XServer : which contains the *main* method for the server program and performs the work of listening to the port, receiving, and sending packets.

XServer Class

- Import
  - java.io.*;
  - java.net.*;

- DatagramSocket:
  - DatagramSocket socket = new DatagramSocket (port number the server is going to listen on);
  - Port number must not be already used by any other application

- Listening Thread:
  - Create and run a thread to wait for packets to arrive

    ```
    Thread t = new Thread(){
        public void run() {
            while (true) {
             //receive request packets
             //do processing using the XProtocol class
             //send response using the received packet IP and port
            }
       }
    };
    t.start();
    ```

XServer Class

- Receive request packet
  - byte[] buf = new byte[256];
  - DatagramPacket packet = new DatagramPacket(buf, buf.length);
  - socket.receive(packet);
  - String received = new String(packet.getData(), 0, packet.getLength());

- Declare and Initialize the protocol:
  - XProtocol xProtocol = new XProtocol();
  - buf = xProtocol.processInput(received);
  - XProtocol is just a simple class with one function :
    - *public String processInput(String theInput)*
    - This function takes an input from the client -> do some processing -> return the output

- Send Response:
  - InetAddress address = packet.getAddress();
  - int port = packet.getPort();
  - packet = new DatagramPacket(buf, buf.length, address, port);
  - socket.send(packet);

- Close Socket:
  - socket.close();

Java Client

- Import
  - java.io.*;
  - java.net.*;

- Create DatagramSocket:
  - DatagramSocket socket = new DatagramSocket();

- Send Request
  - byte[] buf = new byte[256];
  - InetAddress address = InetAddress.getByName(String server IP);
  - DatagramPacket packet = new DatagramPacket(buf, buf.length, address, server application port);
  - socket.send(packet);

- Get Response:
  - packet = new DatagramPacket(buf, buf.length);
  - socket.receive(packet); // wait till a packet is received
  - String received = new String(packet.getData(), 0, packet.getLength());

- Close Socket:
  - socket.close();

UDP Datagram Packet Constraint

- The theoretical maximum amount of data for an IPv4 UDP datagram is 65,507 bytes.
- In practice, On many platforms, the actual limit is more likely to be 8,192 bytes (8K).
- In fact, many operating systems don't support UDP datagrams with more than 8K of data and either split, or discard larger datagrams.
- If a large datagram is too big and as a result the network drops it, your Java program won't be notified of the problem. (UDP is an unreliable protocol)
- Consequently, you shouldn't create DatagramPacket objects with more than 8,192 bytes of data.
- This is a problem for TCP datagrams too, but the stream-based API provided by Socket and ServerSocket completely shields programmers from these details.