# Networking: IPv6, UDP and TCP

# Network Programming in Java – UDP and TCP

SCOMRED, November 2018

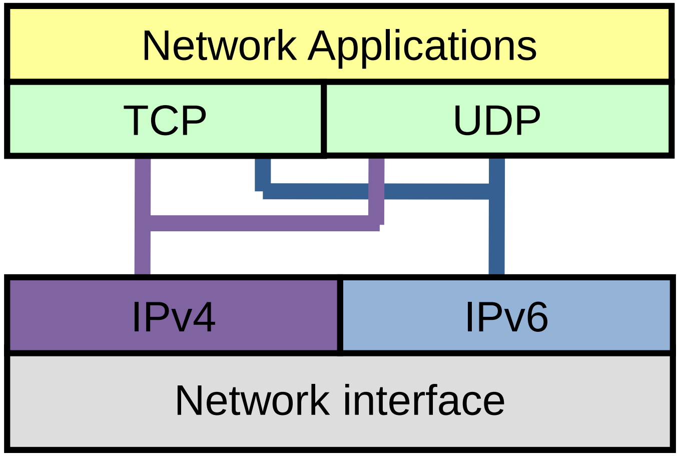
# IPv6 (Internet Protocol version 6)

In nowadays internet, two versions of the IP protocol are being used at the same time, version four and version six. IPv6 is meant to totally replace IPv4, but that won't be in the near future.

One major difference between IPv4 and IPv6 is the number of bits making a node address, 32 bits in IPv4, and 128 bits in IPv6.

Despite this difference, in IPv6 addresses are use the same way as with IPv4, the prefix length indicates how many leading bits of the address are representing the network, that part of the address is the network prefix.

In the present, most nodes are **dual-stack**, they have both an IPv4 layer and an IPv6 layer. By using this strategy nodes can communicate through the internet with IPv4 nodes and also with IPv6 nodes, and this will continue to happen while the internet changes to IPv6.



# IPv6 addresses representation

IPv6 node addresses are rather long (128 bits), to represent them in a human suitable way, they are split into 8 sets of 16 bits each. Each set of 16 bits is represented in hexadecimal and separated from other sets by a colon:

**xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx**

Some writing shortenings are possible: within each set, leading zeros may be removed; the biggest sequence of zeros may be replaced by a double colon.

For instance: **fd1e:0078:0a04:0005:0000:0000:0000:0034/64**

Can also be represented as: **fd1e:78:a04:5::34/64**

This node address belongs to network: **fd1e:78:a04:5::/64**

The first valid node on this network is **fd1e:78:a04:5::1/64**

The last valid node on this network is:

**fd1e:78:a04:5:ffff:ffff:ffff:ffff/64**

Unlike with IPv4, in IPv6 there’s no broadcast address, so the equivalent address can be used by a node.

# Layer 4

In the first lecture we have studied how networks operate up to layer three (the network layer).

Yet, what layer three provides (IPv4 or IPv6) is not suitable to be used by network applications. Two main issues would arise if applications were to use IP directly:

- IP is unreliable, meaning when sending data through the network there's no guarantee it will be delivered, even worst the sender will never known if data has arrived to the destination node or not.

- IP doesn't identify network applications, the IP protocol only handles node addresses. However, within each node, usually there are many network applications running, data from different network applications running it the same node can't get mixed. So, with nothing else but IP, in each node, only one network application could be running.

The layer four (transport layer) handles both these issues, for both IPv4 and IPv6, the layer four implementations that operate over them are TCP and UDP.

# Port numbers

Both UDP (**User Datagram Protocol**) and TCP (**Transmission Control Protocol**) handle the issue of identifying applications the very same way, they use port numbers. Port numbers are 16 bits numbers assigned by the local operating system to network applications as they request, they are unique within the node.

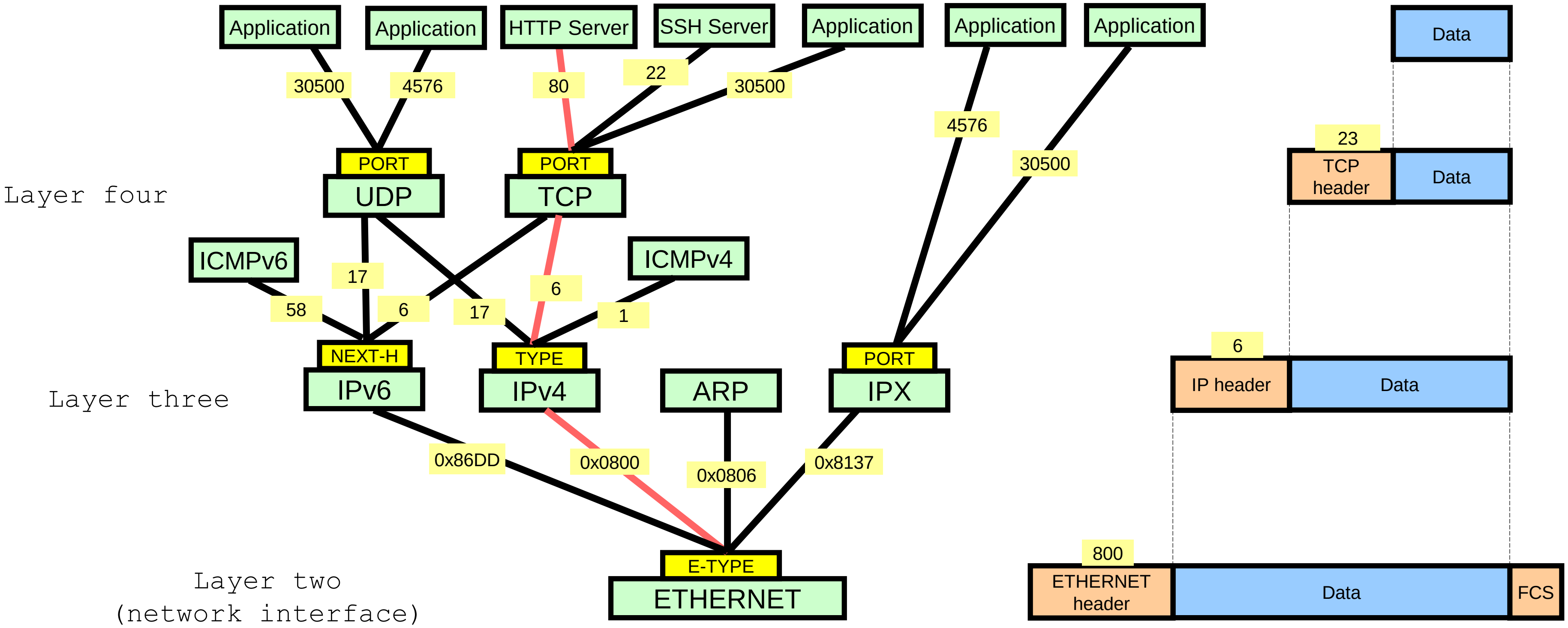
UDP and TCP packets have their own headers, while the header of an IP packet contains the source and destination IP node addresses, the header of UDP and TCP packets contains the source and destination port numbers.

The destination port number represents a running network application on the destination node address.

The source port number represents a running network application on the source node address.

If UDP or TCP data is sent to a port number that is not being used by any application on the destination node, that data is lost.

# Layers, type identifiers and port numbers

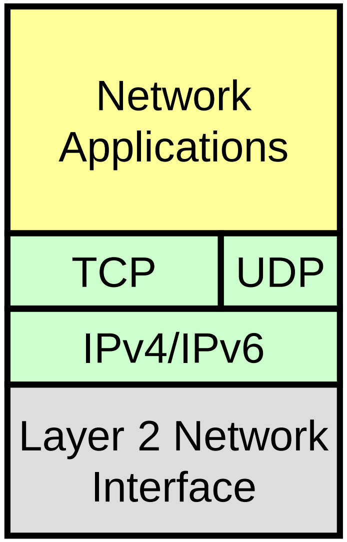


The image above represents how network layers interoperate **within a node.** They all share a same resource, the network interface, and yet data doesn’t get mixed.

We can also see that despite the existence of two IP layers, there’s only one UDP layer and one TCP layer.

# TCP - reliability

TCP/IP stack



Regarding reliability, UDP doesn’t add any significant feature to IP, so UDP is as unreliable as IP. Specifically, when UDP is used to send data, the sender never known it was ever delivered to the destination application.

TCP on the other hand is reliable, and it’s also connection oriented, while UDP is connectionless.

Being connection oriented means that before actually sending data, a connection must be established between two running applications. It’s called the **TCP connection.**

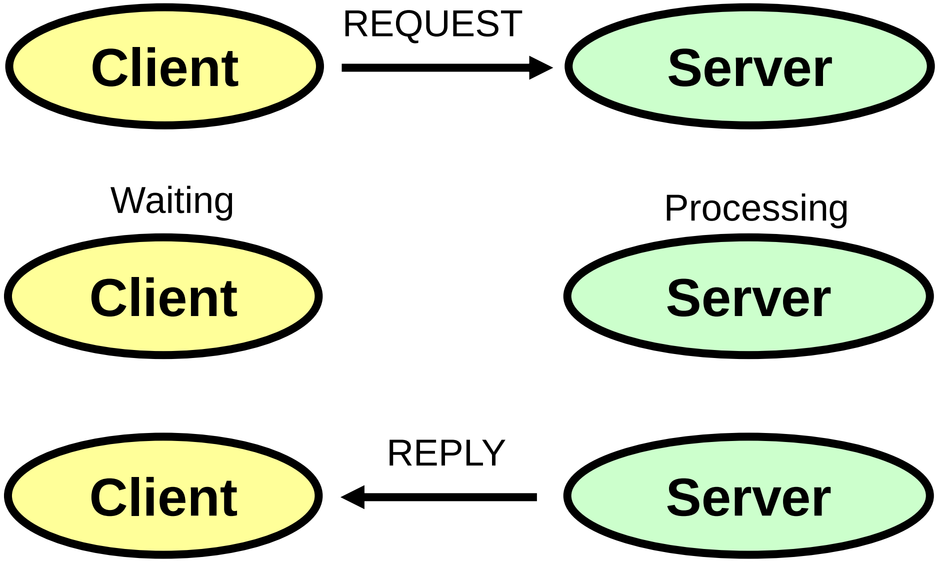
Once the TCP connection is established, then it behaves like a dedicated reliable channel for data transfer between the two applications. Bytes may be written in one side and they will be reliably available for reading on the other side in the same exact sequence.

# The client-server model

Almost every network application dialogues with other network applications by using a very simple model called client-server model. For this model to work, each application must take a different role (client or server) and be aware of it.

**1st** – It all starts by the client initiative of sending a request to the server. To do that it must know the server's IP address and port number.

**2nd** – The server has received the request and is processing it. Meanwhile, the client is waiting for a response.



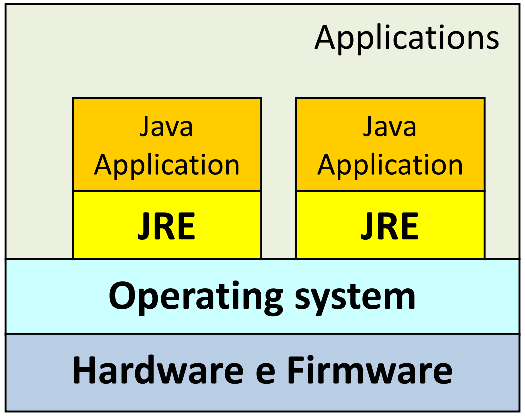
**3rd** – The server sends back a response to the client. The client address (IP and port number) are known to the server from the moment the request was first received.

Port numbers used by servers are fixed and standard for each type of service. Examples: TCP/22 for SSH and TCP/80 for HTTP.

# Java language programming basics

Unlike with other programming languages, when a program written in Java is compiled, the resulting product is not suitable to be run by the operating system.

Programs written in Java are intended to be run in a special environment called the **Java Virtual Machine** or **Java Runtime Environment** (JRE).



Java is object-oriented, an object is made of variables (data) and methods (functions). Objects are classes' instances, to create an object, a class must be declared first, and then objects can be created from that declared class.

Classes, methods and variables must be declared and they have names, objects are created in runtime by using the keyword **new** followed by the calling of the **constructor method**.

The constructor is a method with the same name of the class, it's often used to initialize stuff within the object upon creation.

# Java – variables and objects

When a new object is created, the constructor returns a reference to it, that reference has to be stored in a **variable**, otherwise you won't be able to use that object.

In fact, the Java virtual machine performs a periodic garbage collecting, and unreferenced objects are automatically removed.

When assigning an object reference to a variable, the variable must be of the same class as the referred object.

Methods and variables declared in a class can be declared as public or private. If **private**, they can be accessed only from methods of the class, if **public**, they can be accessed from methods of other objects and classes.

In object-oriented programming you should enforce variables **encapsulation**, in simple words, all variables are declared as private, yet they are indirectly publicly accessible through a set of public methods.

The only required knowledge to be able use the object is their public methods, these publics methods make the object’s interface, other objects interact by using the interface.

# Network sockets in Java

Under the API (Application Programming Interface) point of view, date is sent and receive from the network by using sockets.

In Java, sockets are objects (objects of the **Socket** class or a subclass), in other languages they may be different things, and yet they all represent the same: a mean through which the application can use the network.

In this course, sockets are going to be used to: send and receive UDP packets; establish TCP connections; transfer data through an established TCP connection. These operations are going to be based on the client-server model.

One thing both UDP and TCP clients must known in the first place is the server application address: the IP node’s address of the node where it’s running and the local port number it’s using. This is required for the client to be able to send the request to the server application.

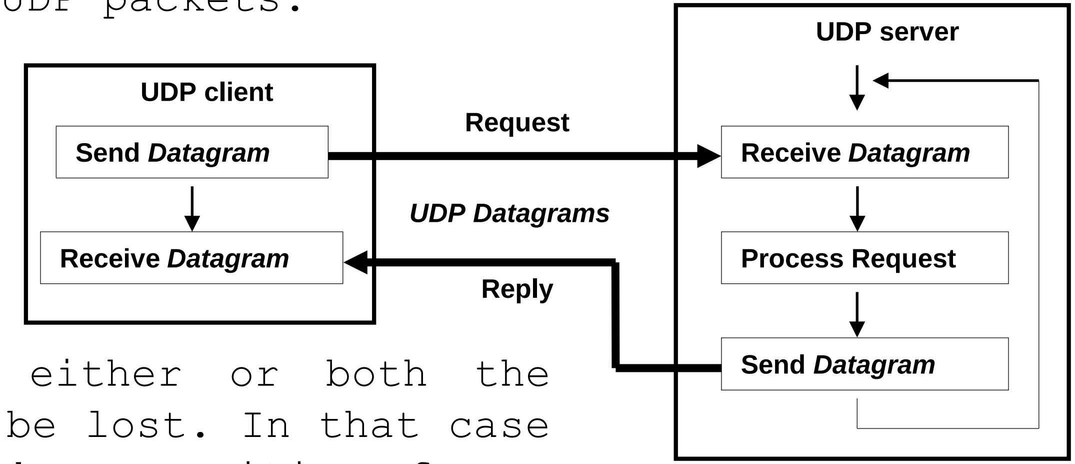
Usually the IP node address is provided by the user interacting with the client application, either an IPv4 address, an IPv6 address or the node’s DNS name. The server’s local port number is fixed and commonly hardcoded, both on the client application and the server application.

# UDP clients and servers

UDP has some notorious limitations, it’s unreliable and connectionless. One other limitation is about the content length of each UDP packet (datagram), it can hold only up to 512 bytes.

Because UDP is connectionless, UDP client-server interactions are in most cases limited to single UDP packets:

The UDP client sends a request (a single UDP packet) and the server replies back with a response (a single UDP packet).



Because UDP is unreliable, either or both the request and the response may be lost. In that case the UDP client application hangs waiting for a response that will never arrive.

If data length of either the request or the response exceeds 512 bytes, they could be split into several UDP datagrams, but this turns the implementation rather more complex. In this case using TCP instead of UDP might be an easier alternative.

In Java there’s a specific subclass of the Socket class to handle UDP, it’s the **DatagramSocket** class.

# DatagramSocket class instantiation

If we intend to implement a UDP server application, the socket must have a fixed UDP local port number:

```
DatagramSocket mySocket = new DatagramSocket(9999);
```

The UDP client application should then send its request to UDP port number 9999 on the server node. Notice port numbers bellow 1024 are available only to the system and administrators, ordinary users are not able to use them.

Regarding the UDP client application local port number, it can be any available port number on its node, to get that the client calls the constructor with no arguments:

```
DatagramSocket mySocket = new DatagramSocket();
```

There's no need for the client port number to be fixed because when the server receives the requests it takes knowledge of both the client's IP node address and the client's local port number, thus when replying back it uses these elements to settle the response's destination address.

# The InetAddress class

To store and handle IP addresses (both IPv4 or IPv6 addresses) Java provides the **InetAddress** class.

The **getByName()** static method receives as argument a string, it can be either an IPv4 address representation, an IPv6 address representation or a node's DNS name. This method creates and returns a new instance of the class (object) with the given IP address stored in it.

The **getHostAddress()** method returns a string containing the human readable representation of the stored IP address.

The **getByName()** method will be typically used on the client application to create a **InetAddress** class object with a user provided IP address stored in it. It will be latter used when sending the request to the server application.

The **getHostAddress()** method may be used to present the source address of a received UDP datagram, or in case of TCP the source address of an incoming TCP connection request.

# The DatagramPacket class

Represents a UDP packet. Before sending a UDP datagram, a **DatagramPacket** object must be created, it will also be required when receiving a UDP datagram to store it. Has several public methods to interact with it's internal data:

**Data** – Data transported by the packet, either to be sent or that has been received:

```
void setData(byte[] buf, int offset, int length);

byte[] getData();
```

**Length** – the number of bytes to be sent or that have been received:

```
void setLength(int length);

int getLength();
```

**Address – the IP remote address** (where to send the packet, or from where it was received):

```
void setAddress(InetAddress addr);

InetAddress getAddress();
```

# The DatagramPacket class

**Port – the remote UDP port number** (where to send the packet or from where the packet was received):

```
void setPort(int port);

int getPort();
```

The **DatagramPacket** class has several constructors, the two most often used are:

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

DatagramPacket(byte[] buf, int length, InetAddress address, int port);
```

The first one is suitable to create an object to store an incoming packet. It only settles the place to store its transported data (content/payload) and the maximum allowed packet size. The remote IP address and the remote port number are available once the packet is received.

# Sending an UDP packet

To send a UDP packet you must:

1st Create a UDP socket (DatagramSocket).

2nd Create a DatagramPacket, settle it's **data** and the **destination address.**

3rd Call the **send()** method of the DatagramSocket and provide it the DatagramPacket as argument. Example:

```
String phrase="Testing";
DatagramSocket mySocket = new DatagramSocket();
DatagramPacket udpPacket = new DatagramPacket(phrase.getBytes(), phrase.length,
         InetAddress.getByName("192.168.10.1"), 9999);
mySocket.send(udpPacket);
mySocket.close();
```

This example sends a UDP datagram with the content "Testing" to the UDP port number **9999** of the node with IPv4 address **192.168.10.1.** When an application no longer needs a socket, it should close it by calling the **close()** method.

# Receiving a UDP packet

To receive a UDP packet you must:

1st Create a DatagramSocket and define a fixed local UDP port number.

2nd Create a DatagramPacket and settle only the place where to store its data and the maximum data size.

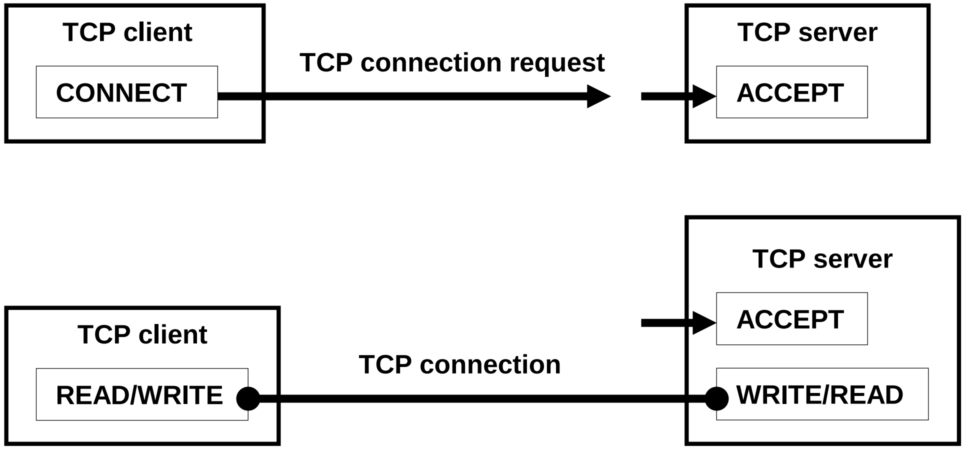
3rd Call the **receive()** method of the DatagramSocket providing in the DatagramPacket. Notice the receive() method will **block the application until a UDP packet is received.** Example:

```
byte[] data = new byte[30];
DatagramSocket mySocket = new DatagramSocket(9999);
DatagramPacket udpPacket = new DatagramPacket(data, 30);
mySocket.receive(udpPacket);
String phrase = new String( udpPacket.getData(), 0, udpPacket.getLength());
System.out.println(phrase + “ received from IP: " +
         udpPacket.getAddress().getHostAddress() + " port: “ +
         udpPacket.getPort());
mySocket.close();
```

# TCP (Transmission Control Protocol)

Unlike UDP, TCP is reliable and connection-oriented. Because it's connection-oriented, before taking advantage of its reliability features, a connection must be established, for that the two enrolled applications must take different roles. In the client-server model:

The TCP client application requests a TCP connection establishment (**connect**) with the server application. To be successful, on the other side there must be a TCP server application than **accepts** that request. Then the TCP connection is established.



To request the TCP connection establishment the TCP client must know the IP address of the node where the server is running and the local port number it is using.

Once the TCP connection has been successfully established, then bytes can be sent through it (write) and received from it (read). The delivery of bytes written on one side is guaranteed on the other side on the same sequence they where written.

# TCP benefits and challenges - synchronization

A TCP connection is a dedicated and reliable communication channel between two applications, through it, bytes are written on one side and will be available for reading on the other side.

This is a significant improvement over UDP, and yet it presents its own challenges.

**Byte synchronization** – receiving operations must match sending operations on the counterpart application. If not so, one application will get blocked on a reading operation, this happens because read operations are blocking, when the reception of a UDP datagram is requested or when the reading of some bytes on a TCP connection is requested, the operations blocks until data is available.

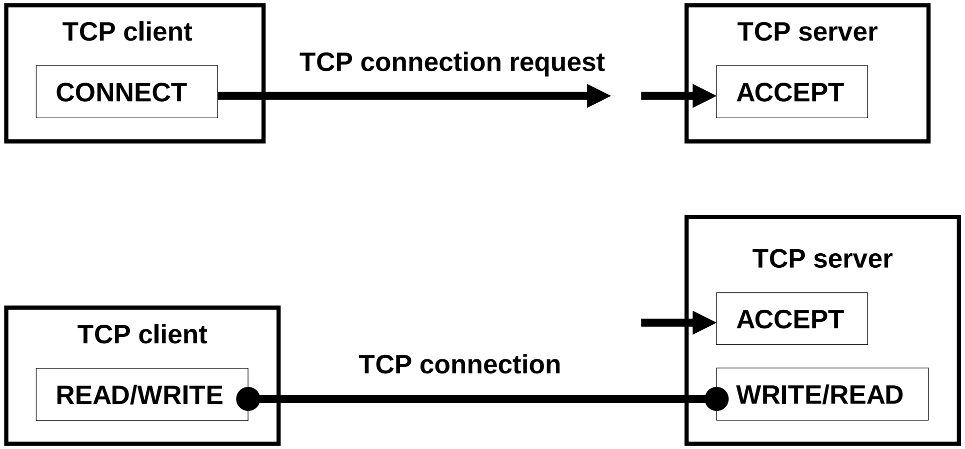
Whereas in UDP synchronization is packet oriented, in TCP synchronization is byte oriented. In UDP when you request the reception of a packet you don’t specify the packet size, in TCP when you request the reading of bytes you must specify how many bytes.

The big issue is, often, the TCP application doesn´t know how many bytes the counterpart is sending. There are mainly to approaches to solve this. One is, the sender previously announcing how many bytes it’s going to send next. One other approach is establishing one byte value to represent the end of sending.

# TCP benefits and challenges

When using TCP there’s another tricky situation on the server side.

The TCP server is blocked on the **accept** operation until a connection establishment request arrives from a client. When that happens a new socket is created, the new socket represents the connection with that specific client and is totally dedicated to that client only.



As clients connect to the server, the server ends up with several sockets to manage: the initial socket where new client connection establishment requests may arrive at any time, and sockets from already established connections with current clients.

This is a problem, at any time the server may have to:

- Accept a new connection from a new client.
- Read an incoming request from a client through one of the already established connections.

And it can’t tell what is going to happen first, so it can’t be blocked waiting for one specific event of those to happen.

# Asynchronous reception

The issue we have just been talking about is generally known as asynchronous reception. It happens whenever an application has several possible input sources and isn't able to tell in which data will be available first.

One possible, but not very efficient, solution is **polling.** Even though by default socket's input operations (e.g. receive, read and accept) block indefinitely until the input event occurs, that can be changed. In Java the **setSoTimeout()** method of the Socket class settles the maximum blocking time for an input operation over it, in milliseconds. If the operation takes longer it will be aborted and an error (exception) produced.

The polling strategy consists on establishing a short input timeout for every socket, and then going through each of them a trying the input operation.

One more efficient approach to this problem is creating multiple parallel running tasks, one for each socket. Then each task itself will be blocked waiting for the input event on its socket, but the server application in the overall is ready for any possible input even.

Parallel tasks may be implemented as processes or threads, in the case of Java we are going to use threads.

# The Socket class

In Java, a TCP client can request a TCP connection establishment by creating a Socket class object using the following constructor:

```
public Socket(InetAddress address, int port)  throws IOException
```

The provided IP address must match the node where the server is running, and the port number, the local TCP port number the server application is using.

In case of failure, for instance because the server application is not running, this will generate an exception, otherwise the TCP connection was successfully established and is represented by the socket.

To implement the server side there's a specific class named **ServerSocket**, its constructor receives as argument the local TCP port number where it will accept client's connection requests.

```
public ServerSocket(int port)  throws IOException
```

Of course, this will generate an exception if the requested port number is already being used. Otherwise, the application will be then ready to receive TCP connection requests, though incoming requests will be on hold until explicitly accepted.

# The accept() method of the ServerSocket class

After creating the ServerSocket, clients' connection requests are not declined, but they are put on hold in a queue until their turn arrives and the accept method is called:

```
public Socket accept()    throws IOException
```

If there are no pending requests for acceptance, this method blocks until one arrives.

On success, accept() returns a new Socket class object representing the TCP connection with that specific client.

Once the connection is established, on each side there’s a Socket object representing the TCP connection.

Now, to be able to send and receive data, Java applications must get the socket’s streams first:

```
public OutputStream getOutputStream()    throws IOException
public InputStream getInputStream()      throws IOException
```

Afterwards they can write bytes into the output stream and read bytes from the input stream.

# Reading and writing bytes

Among others, the **InputStream** class has the following public methods:

**int read() throws IOException**

**int read(byte[] b, int off, int len) throws IOException**

The first one read one single byte and returns its value as integer, it will have a value between 0 and 255. The second method reads **len** bytes and stores them in buffer **b**, starting in position **off**.

Likewise, among others, the **OutputStream** class has the following public methods:

**void write(int b) throws IOException**

**void write(byte[] b, int off, int len) throws IOException**

The first writes a single byte. The second writes **len** bytes, that are stored in buffer **b** starting from position **off**.

# Multi-thread TCP servers in Java

We already know a TCP server has the asynchronous events issue to handle. One effective way to handle that in Java is by creating several threads, each dedicated to, and waiting for one event. Because threads run in parallel, they won’t block each other.

In Java a thread is defined by a declaring class that implements the **Runnable interface**, the class must define the **run()** method. This is the method that will be called by the **start()** method of the **Thread class** to actually start running the thread in parallel.

The class may be declared in either two ways:

- By declaring a subclass of the Thread class (**extends** the Thread class). The Thread class implements the Runnable interface, so it defines the **run()** method, thus when declaring it we must override the inherited declaration (@Override annotation). Because the class is a Thread’s subclass, instances have the **start()** method to turn them into running threads.

- By declaring a new class that implements the **Runnable interface.** With this solution, instances can’t be turned into running threads by themselves. An additional object is required, an instance of the Thread class, it’s used as an intermediary, it creates the running thread and executes our **run()** method.

# TCP server example – the thread

Next we have an example of a class implementing the Runnable interface, it will be used by a TCP server when creating threads to attend clients.

```
public class AttendClient  implements Runnable {
         private Socket cliSock;
         public AttendClient(Socket s) {
                  cliSock=s;
                  }
         public void run() {                    // thread execution starts
                  // TO DO: get cliSock input and output streams
                  // TO DO: read requests and write replies
                  cliSock.close();
         }                                      // thread execution ends
}
```

The AttendClient() class’s constructor (used in instantiation) stores the Socket (connected to the client) in a private variable for latter use when the **run()** method is called (by the **start()** method of the Thread class).

# TCP server example – the main loop

Server applications are most often designed to run for ever, this is one of the few cases where infinite loops are ok:

```
public class TcpServer {
    public static void main(String args[]) {
        static ServerSocket sock = new ServerSocket(9999);
        static Socket nSock;
        while(true) {
            nSock=sock.accept();            // wait for a new connection
            Thread cliConn = new Thread(new AttendClient(nSock));
            cliConn.start();       // start running the thread in background
        }
    }
}
```

Once the accept() method unblocks, the returned Socket is passed to the AttendClient() on instantiation. The new instance of AttendClient is then passed to the constructor of the Thread class.

The thread is now ready to start running by calling its **start()** method, that in turn will call the **run()** method of AttendClient.

# Concurrency and locking

All running threads in an application have access to same data (public variables and methods), this of course raises concurrency issues. The programmer must ensure two threads will never be accessing the same data at the same time, specially if one or both are changing that data, if that happens, results become unpredictable.

Programming languages offer resources to control this, one is the lock. A lock is also known as mutex (mutual exclusion), it can be either on the released or acquired state.

When several threads try acquiring the same lock it’s guaranteed only one thread is going to be successful, all other threads are putted on hold until the lucky thread that acquired the lock releases it.

Bear in mind the lock doesn’t control access to anything beyond the lock itself. It’s up to the programmer ensuring that every thread acquires the appropriate lock before accessing some specific data.

**Intrinsic locks** – in Java, each and every object and class has its own dedicated lock, called **intrinsic lock.** Intrinsic locks are indirectly acquired when the **synchronized** keyword is deployed.

# Static and non-static

You might have noticed something odd, it was told classes have intrinsic locks. This only makes sense if classes exist in runtime and aren’t simple declarations of objects to be.

Variables and methods of a class are by default **non-static**, this means they exist only in objects created from the class.

And yet, some variables and methods of a class can be declared to be **static**, them they exist in the class and not in objects created from the class. The static part of a class exists in runtime, it’s unique and identified by the class name. The non-static part of the class exists only on objects created from in, and they don’t have the static part.

This clarification was required to appropriately talk about the synchronized keyword usage. The most straightforward way to use intrinsic locks is by declaring methods to be synchronized.

**If a method is declared to be synchronized, when called, the intrinsic lock is acquired by the caller, when the method returns, the intrinsic lock is released. If the method is static then the acquired lock belongs to the class, otherwise it belongs to the object.**

# Example – synchronized static methods

In the following totally static class, all methods are declared to be synchronized. So, when called, they all acquired the class’s intrinsic lock. This ensures it will never happen two threads running any of these methods at the same time.

```
public class MyCounter {
    private static Integer value;
    public static synchronized int get() { return value; }
    public static synchronized void inc() {value++; }
    public static synchronized void dec() {value--; }
    public static synchronized void set(int v) {value=v; }
    public static synchronized void reset() {value=0; }
}
```

Blocking the entire class or object for a method’s execution can be excessive or too coarse in many cases.

A more accurate use of intrinsic locking can be achieved by declaring a block to be synchronized with the intrinsic lock of a given object or class.

# Example – synchronized static methods

Now, imagine we want to add a new method to read a value from the user’s terminal. If it was declared as synchronized, then the class would be blocked while the user is typing, and that’s pointless.

Instead, we can lock the class (CLASSNAME.**class**) intrinsic lock only when that’s in fact required.

```
public class MyCounter {
    private static Integer value;
    public static synchronized void inc() { value++; }
    public static synchronized void dec() { value--; }
    public static void readVal() throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print(“Enter a new value please: ");
        int v = Integer.parseInt(in.readLine());
         synchronized(MyCounter.class) { value=v; }
    } }
```

A synchronized static method works the same ways as a block synchronized to CLASSNAME.**class**. A synchronized non-static method works the same ways as a block synchronized to **self**.

**Don't forget methods declared to be synchronized act on different intrinsic locks depending on being static or non-static.**

A careless programmer might forget that and create a **buggy** class like the following:

```
public class MyCounter {
    private static Integer value;
    public static synchronized int get() { return value; }
    public synchronized void inc() { value++; }
    public synchronized void dec() { value--; }
    public static synchronized void set(int v) { value=v; }
    public static synchronized void reset() { value=0; }
}
```

Problems are on methods inc() and dec(), they are non-static, so by declaring them synchronized, they lock the object and not the class, however, they access the static variable named value that belongs to the class and not to the object.