Socket Programming

Created	@February 22, 2022 10:09 AM
Reviewed	☐

Table of contents

Networking revisited
URL
Socket
TCP and UDP
Input and Output
Datagram
Multicast using java socket
Client to Client without server
Sharing objects using socket
How whatsapp works?
Netty
Web socket

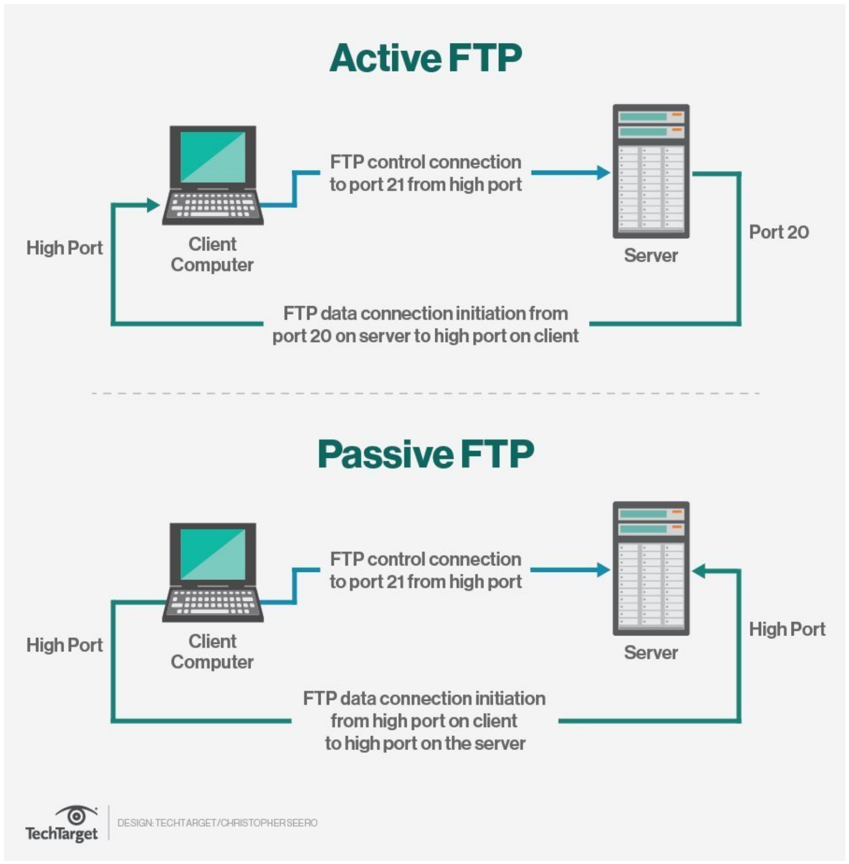
$$Socket\ Programming$$

▼ Networking revisited

▼ Ports

For example, a user request for a file transfer from a client, or local host, to a remote server on the internet uses File Transfer Protocol (FTP) for the transaction. Both devices must be configured to transfer files via FTP. To transfer the file, the Transmission Control Protocol (TCP) software layer in local host identifies the port number of 21, which, by convention, associates with an FTP request -- in the 16-bit port number integer that is appended to the request.

At the server, the TCP layer will read port number 21 and forward the request to the FTP program at the server.


Active FTP
FTP control connection to port 21 from high port
High Port
Client Computer
Server
Port 20
FTP data connection initiation from port 20 on server to high port on client
Passive FTP
FTP control connection to port 21 from high port
High Port
Client Computer
Server
High Port
FTP data connection initiation from high port on client to high port on the server
TechTarget
DESIGN: TECHTARGET/CHRISTOPHER SEERO

- Every Linux service **doesn't have Port Number.**

  Networking services such as browsers, bit-torrent client, etc. **will only have port Numbers. (**https://www.quora.com/Does-every-service-have-a-port-number-How-do-I-identify-the-port-number-of-a-particular-service-in-Linux**)**

- You can have two applications listening on the same IP address, and port number, so long one of the port is a UDP port, while other is a TCP port.

  Explanation:

  The concept of port is relevant on the transport layer of the TCP/IP stack, thus as long as you are using different transport layer protocols of the stack, you can have multiple processes listening on the same `<ip-address>:<port>` combination.

  One doubt that people have is if two applications are running on the same `<ip-address>:<port>` combination, how will a client running on a remote machine distinguish between the two? If you look at the IP layer packet header (https://en.wikipedia.org/wiki/IPv4#Header), you will see that bits 72 to 79 are used for defining protocol, this is how the distinction can be made.

  If however you want to have two applications on same TCP `<ip-address>:<port>` combination, then the answer is no (An interesting exercise will be launch two VMs, give them same IP address, but different MAC addresses, and see what happens - you will notice that some times VM1 will get packets, and other times VM2 will get packets - depending on ARP cache refresh).

I feel that by making two applications run on the same `<op-address>:<port>` you want to achieve some kind of load balancing. For this you can run the applications on different ports, and write IP table rules to bifurcate the traffic between them.

Ports list:

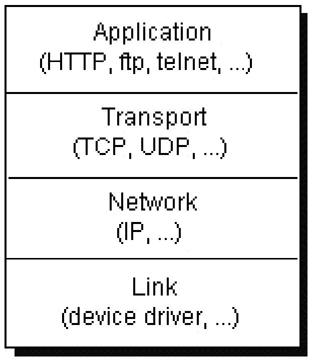
Port	Service name	Transport protocol
20, 21	File Transfer Protocol (FTP)	TCP
22	Secure Shell (SSH)	TCP and UDP
23	Telnet	TCP
25	Simple Mail Transfer Protocol (SMTP)	TCP
50, 51	IPSec	
53	Domain Name System (DNS)	TCP and UDP
67, 68	Dynamic Host Configuration Protocol (DHCP)	UDP
69	Trivial File Transfer Protocol (TFTP)	UDP
80	HyperText Transfer Protocol (HTTP)	TCP
110	Post Office Protocol (POP3)	TCP
119	Network News Transport Protocol (NNTP)	TCP
123	Network Time Protocol (NTP)	UDP
135-139	NetBIOS	TCP and UDP
143	Internet Message Access Protocol (IMAP4)	TCP and UDP
161, 162	Simple Network Management Protocol (SNMP)	TCP and UDP
389	Lightweight Directory Access Protocol	TCP and UDP
443	HTTP with Secure Sockets Layer (SSL)	TCP and UDP
989, 990	FTP over SSL/TLS (implicit mode)	TCP
3389	Remote Desktop Protocol	TCP and UDP

▼ Why FTP has 2 ports?

FTP uses two TCP connections for communication. One to pass control information, and is not used to send files on port 21, only control information. And the other, a data connection on port **20 to send the data files** between the client and the server. The connection has to be established before the files can actually be sent across. I think the user authentication takes place over the control connection on port 21. After which, the data starts transferring over the data connection on port 20.

▼ 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:



When you write Java programs that communicate over the network, you are programming at the application layer. Typically, you don't need to concern yourself with the TCP and UDP layers. Instead, you can use the classes in the `java.net` package. These classes provide system-independent network communication. However, to decide which Java classes your programs should use, you do need to understand how TCP and UDP differ.

TCP

When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call. If you want to speak to Aunt Beatrice in Kentucky, a connection is established when you dial her phone number and she answers. You send data back and forth over the connection by speaking to one another over the phone lines. Like the phone company, 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. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.

Definition:

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

UDP

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. 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.

Definition:

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.

For many applications, the guarantee of reliability is critical to the success of the transfer of information from one end of the connection to the other. However, other forms of communication don't require such strict standards. In fact, they may be slowed down by the extra overhead or the reliable connection may invalidate the service altogether.

Consider, for example, a clock server that sends the current time to its client when requested to do so. If the client misses a packet, it doesn't really make sense to resend it because the time will be incorrect when the client receives it on the second try. If the client makes two requests and receives packets from the server out of order, it doesn't really matter because the client can figure out that the packets are out of order and make another request. The reliability of TCP is unnecessary in this instance because it causes performance degradation and may hinder the usefulness of the service.

Another example of a service that doesn't need the guarantee of a reliable channel is the ping command. The purpose of the ping command is to test the communication between two programs over the network. In fact, ping needs to know about dropped or out-of-order packets to determine how good or bad the connection is. A reliable channel would invalidate this service altogether.

The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data from one application to another. Sending datagrams is much like sending a letter through the mail service: The order of delivery is not important and is not guaranteed, and each message is independent of any others.

Note:

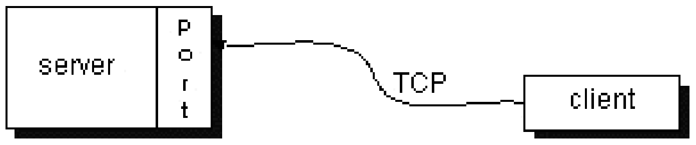
Many firewalls and routers have been configured not to allow UDP packets. If you're having trouble connecting to a service outside your firewall, or if clients are having trouble connecting to your service, ask your system administrator if UDP is permitted.

Understanding Ports

Generally speaking, a computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the computer. So how does the computer know to which application to forward the data? Through the use of *ports*.

Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP and UDP use to deliver the data to the right application.

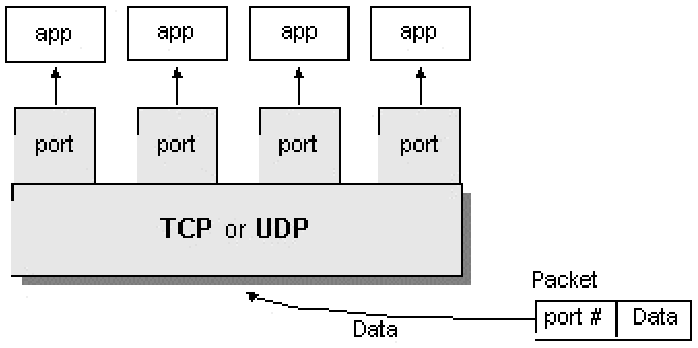
In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then rendezvous with the server at the server's port, as illustrated here:



Definition:

The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.

In datagram-based communication such as UDP, the datagram packet contains the port number of its destination and UDP routes the packet to the appropriate application, as illustrated in this figure:



Port numbers range from 0 to 65,535 because ports are represented by 16-bit numbers. The port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called *well-known ports*. Your applications should not attempt to bind to them.

Networking Classes in the JDK

Through the classes in `java.net`, Java programs can use TCP or UDP to communicate over the Internet. The `URL`, `URLConnection`, `Socket`, and `ServerSocket` classes all use TCP to communicate over the network. The `DatagramPacket`, `DatagramSocket`, and `MulticastSocket` classes are for use with UDP.

▼ URL

▼ What is a URL?

If you've been surfing the Web, you have undoubtedly heard the term URL and have used URLs to access HTML pages from the Web.

It's often easiest, although not entirely accurate, to think of a URL as the name of a file on the World Wide Web because most URLs refer to a file on some machine on the network. However, remember that URLs also can point to other resources on the network, such as database queries and command output.

Definition:

URL is an acronym for *Uniform Resource Locator* and is a reference (an address) to a resource on the Internet.

A URL has two main components:

- Protocol identifier: For the URL `http://example.com`, the protocol identifier is `http`.
- Resource name: For the URL `http://example.com`, the resource name is `example.com`.

Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.

The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:

Host Name

The name of the machine on which the resource lives.

Filename

The pathname to the file on the machine.

Port Number

The port number to which to connect (typically optional).

Reference

A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

For many protocols, the host name and the filename are required, while the port number and reference are optional. For example, the resource name for an HTTP URL must specify a server on the network (Host Name) and the path to the document on that machine (Filename); it also can specify a port number and a reference.

▼ Creating a URL

The easiest way to create a `URL` object is from a `String` that represents the human-readable form of the URL address. This is typically the form that another person will use for a URL. In your Java program, you can use a `String` containing this text to create a `URL` object:

`URL myURL = new URL("http://example.com/");`

The `URL` object created above represents an *absolute URL*. An absolute URL contains all of the information necessary to reach the resource in question. You can also create `URL` objects from a *relative URL* address.

Creating a URL Relative to Another

A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL.

Relative URL specifications are often used within HTML files. For example, suppose you write an HTML file called `JoesHomePage.html`. Within this page, are links to other pages, `PicturesOfMe.html` and `MyKids.html`, that are on the same machine and in the same directory as `JoesHomePage.html`. The links to `PicturesOfMe.html` and `MyKids.html` from `JoesHomePage.html` could be specified just as file names, like this:

```
<a href="PicturesOfMe.html">Pictures of Me</a>
<a href="MyKids.html">Pictures of My Kids</a>
```

These URL addresses are *relative URLs*. That is, the URLs are specified relative to the file in which they are contained — `JoesHomePage.html`.

In your Java programs, you can create a `URL` object from a relative URL specification. For example, suppose you know two URLs at the site `example.com`:

```
http://example.com/pages/page1.html
http://example.com/pages/page2.html
```

You can create `URL` objects for these pages relative to their common base URL: `http://example.com/pages/` like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

This code snippet uses the `URL` constructor that lets you create a `URL` object from another `URL` object (the base) and a relative URL specification. The general form of this constructor is:

```
URL(URL baseURL, String relativeURL)
```

The first argument is a `URL` object that specifies the base of the new `URL`. The second argument is a `String` that specifies the rest of the resource name relative to the base. If `baseURL` is null, then this constructor treats `relativeURL` like an absolute URL specification. Conversely, if `relativeURL` is an absolute URL specification, then the constructor ignores `baseURL`.

This constructor is also useful for creating `URL` objects for named anchors (also called references) within a file. For example, suppose the `page1.html` **file has a named anchor** called `BOTTOM` at the bottom of the file. You can use the relative URL constructor to create a `URL` object for it like this:

```
URL page1BottomURL = new URL(page1URL," #BOTTOM ");
```

Other URL Constructors

The `URL` class provides two additional constructors for creating a `URL` object. These constructors are useful when you are working with URLs, such as HTTP URLs, that have host name, filename, port number, and reference components in the resource name portion of the URL. These two constructors are useful when you do not have a String containing the complete URL specification, but you do know various components of the URL.

For example, suppose you design a network browsing panel similar to a file browsing panel that allows users to choose the protocol, host name, port number, and filename. You can construct a `URL` from the panel's components. The first constructor creates a `URL` object from a protocol, host name, and filename. The following code snippet creates a `URL` to the `page1.html` file at the `example.com` site:

```
new URL(" http ", "example.com", "/pages/page1.html");
```

This is equivalent to

`new URL("http :// example.com/pages/page1.html");`

The first argument is the protocol, the second is the host name, and the last is the pathname of the file. Note that the filename contains a forward slash at the beginning. This indicates that the filename is specified from the root of the host.

The final `URL` constructor adds the port number to the list of arguments used in the previous constructor:

`URL gamelan = new URL("http", "example.com", 80, "pages/page1.html");`

This creates a `URL` object for the following URL:

`http://example.com:80/pages/page1.html`

If you construct a `URL` object using one of these constructors, you can get a `String` containing the complete URL address by using the `URL` object's `toString` method or the equivalent `toExternalForm` method.

URL addresses with Special characters

Some URL addresses contain special characters, for example the space character. Like this:

`http://example.com/hello world/`

To make these characters legal they need to be encoded before passing them to the URL constructor.

`URL url = new URL("http://example.com/hello%20world");`

Encoding the special character(s) in this example is easy as there is only one character that needs encoding, but for URL addresses that have several of these characters or if you are unsure when writing your code what URL addresses you will need to access, you can use the multi-argument constructors of the `java.net.URI` class to automatically take care of the encoding for you.

`URI uri = new URI("http", "example.com", "/hello world/", "");`

And then convert the URI to a URL.

`URL url = uri.toURL();`

MalformedURLException

Each of the four `URL` constructors throws a `MalformedURLException` if the arguments to the constructor refer to a `null` or unknown protocol. Typically, you want to catch and handle this exception by embedding your URL constructor statements in a `try`/`catch` pair, like this:

```
try {
    URL myURL = new URL(...);
}
catch (MalformedURLException e) {
    // exception handler code here
    // ...
}
```

See Exceptions for information about handling exceptions.

Note:

`URL`s are "write-once" objects. Once you've created a `URL` object, you cannot change any of its attributes (protocol, host name, filename, or port number).

▼ Parsing a URL

The `URL` class provides several methods that let you query `URL` objects. You can get the protocol, authority, host name, port number, path, query, filename, and reference from a URL using these accessor methods:

`getProtocol` Returns the protocol identifier component of the URL. **`getAuthority`** Returns the authority component of the URL. **`getHost`** Returns the host name component of the URL. **`getPort`** Returns the port number component of the URL. The `getPort` method returns an integer that is the port number. If the port is not set, `getPort` returns -1. **`getPath`** Returns the path component of this URL. **`getQuery`** Returns the query component of this URL. **`getFile`** Returns the filename component of the URL. The `getFile` method returns the same as `getPath`, plus the concatenation of the value of `getQuery`, if any. **`getRef`** Returns the reference component of the URL.

Note:

Remember that not all URL addresses contain these components. The URL class provides these methods because HTTP URLs do contain these components and are perhaps the most commonly used URLs. The URL class is somewhat HTTP-centric.

You can use these `get` *`xxx`* methods to get information about the URL regardless of the constructor that you used to create the URL object.

The URL class, along with these accessor methods, frees you from ever having to parse URLs again! Given any string specification of a URL, just create a new URL object and call any of the accessor methods for the information you need. This small example program creates a URL from a string specification and then uses the URL object's accessor methods to parse the URL:

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

public class ParseURL {
    public static void main(String[] args) throws Exception {

        URL aURL = new URL("http://example.com:80/docs/books/tutorial"
                           + "/index.html?name=networking#DOWNLOADING");

        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("authority = " + aURL.getAuthority());
        System.out.println("host = " + aURL.getHost());
        System.out.println("port = " + aURL.getPort());
        System.out.println("path = " + aURL.getPath());
        System.out.println("query = " + aURL.getQuery());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("ref = " + aURL.getRef());
    }
}
```

Here is the output displayed by the program:

```
protocol = http
authority = example.com:80
host = example.com
port = 80
path = /docs/books/tutorial/index.html
query = name=networking
```

`filename = /docs/books/tutorial/index.html?name=networking`
`ref = DOWNLOADING`

▼ Reading directly from URL

After you've successfully created a `URL`, you can call the `URL`'s `openStream()` method to get a stream from which you can read the contents of the URL. The `openStream()` method returns a `java.io.InputStream` object, so reading from a URL is as easy as reading from an input stream.

The following small Java program uses `openStream()` to get an input stream on the URL `http://www.motadata.com/`. It then opens a `BufferedReader` on the input stream and reads from the `BufferedReader` thereby reading from the URL. Everything read is copied to the standard output stream:

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

public class URLReader {
    public static void main(String[] args) throws Exception {

        URL motadata = new URL("http://www.motadata.com/");
        BufferedReader in = new BufferedReader(
        new InputStreamReader(motadata.openStream()));

        String inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

When you run the program, you should see, scrolling by in your command window, the HTML commands and textual content from the HTML file located at `http://www.motadata.com/`. Alternatively, the program might hang or you might see an exception stack trace. If either of the latter two events occurs, you may have to set the proxy host so that the program can find the Motadata server.

▼ Connecting to a URL

After you've successfully created a `URL` object, you can call the `URL` object's `openConnection` method to get a `URLConnection` object, or one of its protocol specific subclasses, for example `java.net.HttpURLConnection`

You can use this `URLConnection` object to setup parameters and general request properties that you may need before connecting. Connection to the remote object represented by the URL is only initiated when the `URLConnection.connect` method is called. When you do this you are initializing a communication link between your Java program and the URL over the network. For example, the following code opens a connection to the site `motadata.com`:

```
try {
    URL myURL = new URL("https://www.motadata.com/");
    URLConnection myURLConnection = myURL.openConnection();
    myURLConnection.connect();
}
catch (MalformedURLException e) {
    // new URL() failed
    // ...
}
catch (IOException e) {
    // openConnection() failed
    // ...
}
```

A new `URLConnection` object is created every time by calling the `openConnection` method of the protocol handler for this URL.

You are not always required to explicitly call the `connect` method to initiate the connection. Operations that depend on being connected, like `getInputStream`, `getOutputStream`, etc, will implicitly perform the connection, if necessary.

Now that you've successfully connected to your URL, you can use the `URLConnection` object to perform actions such as reading from or writing to the connection. The next section shows you how.

▼ Read from and write to URLConnection

1. Create a `URL`.
2. Retrieve the `URLConnection` object.
3. Set output capability on the `URLConnection`.
4. Open a connection to the resource.
5. Get an output stream from the connection.
6. Write to the output stream.
7. Close the output stream.

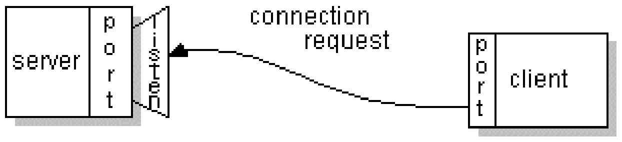
```
URL url = new URL("---url---");
URLConnection connection = url.openConnection();
connection.setDoOutput(true);
//A URL connection can be used for input and/or output. Setting the
//setDoOutput flag to true indicates that the application intends to
//write data to the URL connection.

//output
OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());
out.write("string=" + myString);
out.close();

//input
BufferedReader in = new BufferedReader(
                    new InputStreamReader(connection.getInputStream()));
String decodedString;
while ((decodedString = in.readLine()) != null)
{
    System.out.println(decodedString);
}
in.close();
```

▼ Socket

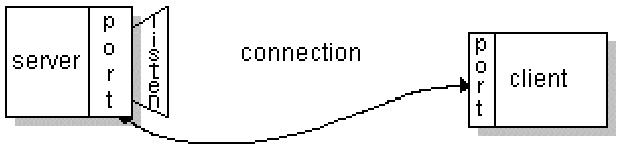
▼ What is socket?

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 socket for a client to make a connection request.**

On the client-side: The client knows the **hostname** of the machine on which the **server is running and the port number** on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself 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.

Definition:

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

An endpoint is a combination of an IP address and a port number. **Every TCP connection can be uniquely identified by its two endpoints**. That way you can have multiple connections between your host and the server.

The `java.net` package in the Java platform provides a class, `Socket`, that implements one side of a two-way connection between your Java program and another program on the network. The `Socket` class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the `java.net.Socket` class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, `java.net` includes the `ServerSocket` class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the `Socket` and `ServerSocket` classes.

If you are trying to connect to the Web, the `URL` class and related classes (`URLConnection`, `URLEncoder`) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation. See Working with URLs for information about connecting to the Web via URLs.

▼ Reading from and writing to a socket

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the `Socket` class and then, how the client can send data to and receive data from the server through the socket.

The example program implements a client, `EchoClient`, that connects to an echo server. The echo server receives data from its client and echoes it back. The example `EchoServer` implements an echo server. (Alternatively, the client can connect to any host that supports the Echo Protocol.)

The `EchoClient` example creates a socket, thereby getting a connection to the echo server. It reads input from the user on the standard input stream, and then forwards that text to the echo server by writing the text to the socket. The server echoes the input back through the socket to the client. The client program reads and displays the data passed back to it from the server.

Note that the `EchoClient` example both writes to and reads from its socket, thereby sending data to and receiving data from the echo server.

Let's walk through the program and investigate the interesting parts. The following statements in the `try`-with-resources statement in the `EchoClient` example are critical. These lines establish the socket connection between the client and the server and open a `PrintWriter` and a `BufferedReader` on the socket:

```
String hostName = args[0];
int portNumber = Integer.parseInt(args[1]);

try {
    Socket echoSocket = new Socket(hostName, portNumber);        // 1st statement
    PrintWriter out =                                            // 2nd statement
        new PrintWriter(echoSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(                      // 3rd statement
            new InputStreamReader(echoSocket.getInputStream()));
    BufferedReader stdIn = new BufferedReader(                   // 4th statement
            new InputStreamReader(System.in))
}
```

LINE 1: The first statement in the `try`-with resources statement creates a new `Socket` object and names it `echoSocket`. The `Socket` constructor used here requires the name of the **computer and the port number to which you want to connect**. The example program uses the first command-line argument as the name of the computer (the host name) and the second command line argument as the port number. When you run this program on your computer, make sure that the host name you use is the fully qualified IP name of the computer to which you want to connect. For example, if your echo server is running on the computer `echoserver.example.com` and it is listening on port number 7, first run the following command from the computer `echoserver.example.com` if you want to use the `EchoServer` example as your echo server:

`java EchoServer 7`

Afterward, run the `EchoClient` example with the following command:

`java EchoClient echoserver.example.com 7`

LINE 2: The second statement in the `try`-with resources statement gets the socket's output stream and opens a `PrintWriter` on it named `out`.

LINE 3: Similarly, the third statement gets the socket's input stream and opens a `BufferedReader` on it named `in`. The example uses readers and writers so that it can write Unicode characters over the socket. If you are not yet familiar with the Java platform's I/O classes, you may wish to read Basic I/O.

The next interesting part of the program is the `while` loop. The loop reads a line at a time from the standard input stream with the `BufferedReader` object `stdIn`, which is created in the fourth statement in the `try`-with resources statement. The loop then immediately sends the line to the server by writing it to the `PrintWriter` connected to the socket:

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
    out.println(userInput);
    System.out.println("echo: " + in.readLine());
}
```

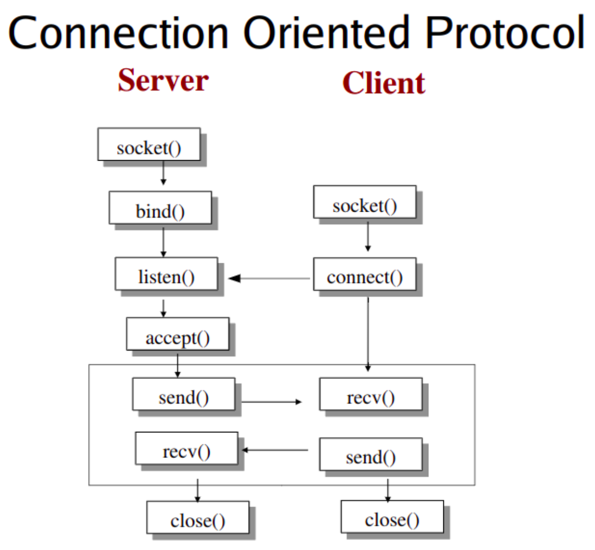
The last statement in the `while` loop reads a line of information from the `BufferedReader` connected to the socket. The `readLine` method waits until the server echoes the information back to `EchoClient`. When `readline` returns, `EchoClient` prints the information to the standard output.

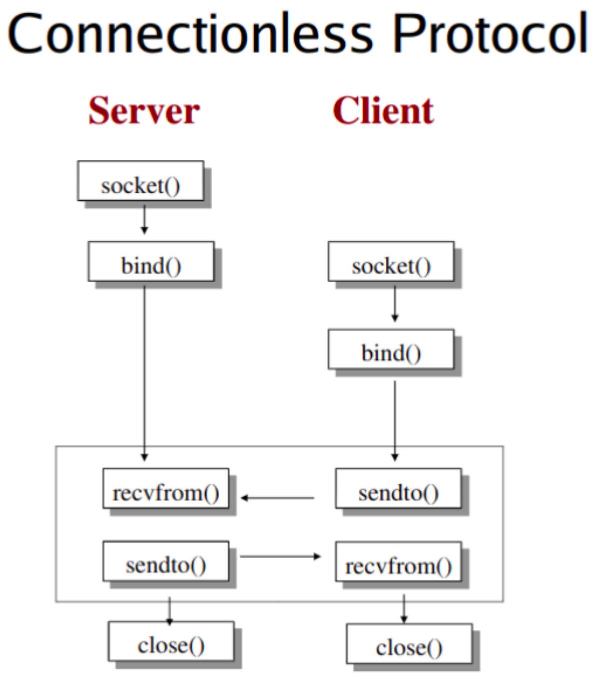
The `while` loop continues until the user types an end-of-input character. That is, the `EchoClient` example reads input from the user, sends it to the Echo server, gets a response from the server, and displays it, until it reaches the end-of-input. (You can type an end-of-input character by pressing **Ctrl-C**.) The `while` loop then terminates, and the Java runtime automatically closes the readers and writers connected to the socket and to the standard input stream, and it closes the socket connection to the server. The Java runtime closes these resources automatically because they were created in the `try`-with-resources statement. The Java runtime closes these resources in reverse order that they were created. (This is good because streams connected to a socket should be closed before the socket itself is closed.)

This client program is straightforward and simple because the echo server implements a simple protocol. The client sends text to the server, and the server echoes it back. When your client programs are talking to a more complicated server such as an HTTP server, your client program will also be more complicated. However, the basics are much the same as they are in this program:

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.

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





PrintWriter(OutputStream out, boolean autoFlush)

Data doesn't usually get sent right away on the socket, it is buffered up to a certain point and then sent all-at-once.

Auto-flushing means data goes right through the buffer and then flushed out, not kept in the buffer waiting for other data to arrive and accumulate.

As simple as that.

Without auto-flush:

```
Tick | DATA sent|Socket Buffer| DATA received
.....|..........|............|.............
1    | XX       | XX          | (nothing)
2    | yy       | yyXX        | (nothing)
3    | ZZZ      | ZZZyyXX     | (nothing)
4    | t        | (empty)     | tZZZyyXX
```

With auto-flush:

```
Tick | DATA sent | Socket Buffer | DATA received
.....|...........|...............|.............
1    | XX        | ()            | XX
2    | yy        | ()            | yy
3    | ZZZ       | ()            | ZZZ
4    | t         | ()            | t
```

- Socket Buffer size: 8 chars
- Very simple example, you might raise some other questions after seeing it - a big part of it is also implementation-dependent. Moreover, buffering can happen at various levels (sender, receiver, application, etc..)

▼ Localhost 127.0.0.1

When you call an IP address on your computer, you try to contact another computer on the internet but when you call the IP address 127.0.0.1 then you are communicating with the local host. **Localhost** is always your own computer. Your computer is talking to itself when you call the local host. Your computer does not always directly identify the local host. **Within your personal network localhost has a separate IP address like 192.168.0.1. (for most cases) which is different from the one you use on the internet i.e. the public IP address.** This is usually dynamically assigned by the internet service provider (ISP). Localhost can be seen as a server that is used on your own computer.

This term is generally used in the context of networks. Localhost is not just the name for the virtual server but it is also its **domain name**. Just like .example, .test, or .invalid, ., .localhost is a top-level domain reserved for documentation and testing purposes. While accessing the domain, a loopback is triggered. If you access “http://localhost” in the browser, the request will not be forwarded to the internet through the router. It will instead remain in your own system. Localhost has the IP address 127.0.0.1. This **refers back to your own server**.

127.0.0.1 – how does loopback work?

To communicate with each other within a network IP addresses are used. The participants in the network have their own unique addresses. Using TCP/IP data packets are able to reach the correct destination. The protocol pair Transmission Control Protocol (TCP) and Internet Protocol (IP) are some of the main features of the internet. TCP/IP is also used outside of the internet in local networks. The Internet Protocol is responsible for allowing the IP address and subnet mask to address subscribers in a network during the transmission.

The allocation of public IP addresses is regulated by an international organization which is the Internet Corporation for Assigned Names and Numbers (ICANN). **ICANN** is also responsible for the allocation of domain names called the Domain Name System (DNS). But certain address ranges are reserved for special purposes, like the range from 127.0.0.0 to 127.255.255.255. There is no reliable information on why that range was chosen. IP addresses on the internet are divided into different classes. The first class Class A started with 0.0.0.0 (reserved address) and ended with 127.255.255.255. 127 is the last block of the Class A network. Its important position could have been the reason for its selection.

Within this address range, a Localnet can be set up. The special thing about this range is that IP addresses are not uniquely assigned in it, as is usually the case. Also, it was reserved by ICANN.

If you enter an IP address or corresponding domain name in your browser, the router forwards your request to the internet which connects you to the server. This means that if you enter 172.217.0.0, you will reach the Google homepage but the situation is different with 127.0.0.1. The requests to this address will not be forwarded to the internet. **TCP/IP recognizes from the first block (127) that you don't want to access the internet, you are calling yourself instead. This then triggers the loopback.**

The reason why a loopback device is created is so that the backlink to your own computer works. Through the operating system, this is a virtual interface that is created. The interface is called lo or lo0 and can also be displayed using the ifconfig command in Unix systems. A similar command for Windows is ipconfig.

What is localhost used for?

Developers use the local host to test web applications and programs. Network administrators use the loopback to test network connections. Another use for the localhost is the host's file, where you can use the loopback to block malicious websites.

For Testing Purposes – Web servers mainly use the local host for the programming applications that need to communicate over the internet. During development, it is important to find out whether the application actually works as developed once it has internet access. Localhosts' other functions are only possible if the required files can be found on the internet. As we can see that there is a difference between opening an HTML document on your PC or loading it onto a server and accessing it. Releasing a product without testing it doesn't make sense. So loopback is used by developers to test them. They can stimulate a connection while also avoiding network errors. The connection just stays completely inside their own system.

Another advantage of using localhost for testing purposes is the speed. Usually, more than 100 milliseconds are taken when you send a request over the internet. The maximum transmission time is just one millisecond for sending a ping to localhost. The correctness of the internet protocol can also be implemented using this technology.

If you want to set up your own test server on your PC to address it through the local host, the right software is needed. Softwares such as XAMPP specifically designed for use as localhost can be used.

To block websites – Localhost can also block the host's files. This file is a predecessor of the Domain Name System (DNS). In this IP addresses can be assigned to the corresponding domains. The domain name is translated into an IP address when you enter a website address in the browser. It used to be the host file, but today usually the global DNS is used but the host file is still present in most operating systems. In Windows, the file is found under \system32\drivers\etc\hosts whereas, with macOS and other Unix systems, it is found under /etc/hosts.

There are probably these two entries left if there are no file changes done:

```
127.0.0.1       localhost

::1             localhost
```

The name resolution for the localhost need not have to be done over the internet. Localhost can also use the host file to block certain websites. For this, the website to be blocked must be entered into the list and the IP address 127.0.0.1 must be assigned to the domain. If you or a malicious script try to call up the locked domain, the browser will check the host's file first and will find your entry there. The domain name 0.0.0.0 can also be used.

The browser will then try to access the corresponding website on the server with 127.0.0.1. However, it is unlikely that the browser will be able to locate it, as the requested file will not be there. However, if your own test server is set up, then the browser may find home.html, which is just your own file. An error message appears instead of the requested website if you have not set up your own test server. Ad inserts throughout the system can be switched off using this technology. To avoid every entry manually, you can find finished and regularly extended host files on the Internet.

▼ 0.0.0.0

https://www.youtube.com/watch?v=my2vBYZ1RFw

https://www.geeksforgeeks.org/difference-between-127-0-0-1-and-0-0-0-0/

In Internet Protocol version 4, the address 0.0.0.0 is a non-routable meta-address used to designate an invalid, unknown, or non applicable target. To give a special meaning to an otherwise invalid piece of data is an application of in-band signaling.

In the context of servers, 0.0.0.0 means *all IPv4 addresses on the local machine*. If a host has two IP addresses, 192.168.1.1 and 10.1.2.1, and a server running on the host listens on 0.0.0.0, it will be reachable at both of those IPs (*Note: This particular text is repeated from above as part of the overall answer*).

In the context of routing, 0.0.0.0 usually means the default route, i.e. the route which leads to 'the rest of' the Internet instead of somewhere on the local network.

Uses Include:

- The address a host claims as its own when it has not yet been assigned an address. Such as when sending the initial DHCPDISCOVER packet when using DHCP.
- The address a host assigns to itself when an address request via DHCP has failed, provided the host's IP stack supports this. This usage has been replaced with the APIPA mechanism in modern operating systems.

- A way to specify *any IPv4-host at all*. It is used in this way when specifying a default route.
- A way to explicitly specify that the target is unavailable. *Source:* 127.0.0.1 – What Are its Uses and Why is it Important?
- A way to specify *any IPv4 address at all*. It is used in this way when configuring servers (i.e. when binding listening sockets). This is known to TCP programmers as INADDR_ANY. [*bind(2) binds to addresses, not interfaces.*]

  In IPv6, the all-zeros-address is written as *::*

▼ LocalPort

Eg. [Practical: e*choTCPOneToOne*] Server is on 8080 port. sout(serverSocket)

ServerSocket[addr=0.0.0.0/0.0.0.0,localport=8080]

Client's socket is formed by passing server's port and ip address. If we sout(socket) from client then we get

Socket[addr=khush-HP-ProBook-430-G3/127.0.1.1,port=8080,localport=35606]

While is we sout(client) which is accepted at server side using client=serverSocket.accept()

Socket[addr=/127.0.0.1,port=35606,localport=8080]

Port is server's port to which the client is connected.

LocalPort is the port with which the client's socket is connected and running.

Try: cd /etc/

Then: less hosts

```
127.0.0.1       localhost
127.0.1.1       khush-HP-ProBook-430-G3

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

---

https://stackoverflow.com/questions/13503929/java-socket-local-port#:~:text=your main question%2C-,local port is nothing but a socket descriptor,in which might prevent this

local port is nothing but **a socket descriptor (or a very high numbered port)**
Whenever a process makes an API call, the request, via the kernel, creates a socket descriptor so as to bind (and identify) this particular request's response.

The **local port is on the same computer as the SSH client**, and this port is the "forwarded port". On the same computer, any client that wants to connect to the same destination host and port can be configured to connect to the forwarded port (rather than directly to the destination host and port).

The SocketImpl class is an abstract class that defines the bulk of the methods that make the Socket and ServerSocket classes work. Thus, SocketImpl is used to create both client and server sockets. Non-public subclasses of SocketImpl provide platform-specific implementations of stream-based socket communication. A plain socket implements the methods in SocketImpl as described; other implementations could provide socket communication through a proxy or firewall.
docstore.mik.ua/orelly/java/fclass/ch15_18.htm

▼ Behaviour of closing a stream

Closes the stream and releases any system resources associated with it. Once the stream has been closed, further read(), ready(), mark(), reset(), or skip() invocations will throw an IOException. Closing a previously closed stream has no effect.

Closes this output stream and releases any system resources associated with this stream. The general contract of `close` is that it closes the output stream. **A closed stream cannot perform output operations and cannot be reopened.**

Closing the returned `OutputStream` will close the associated socket.

Closing the returned `InputStream` will close the associated socket.

Closing this socket will also close the socket's `InputStream` and `OutputStream`.

▼ TCP and UDP

Main difference conceptually:

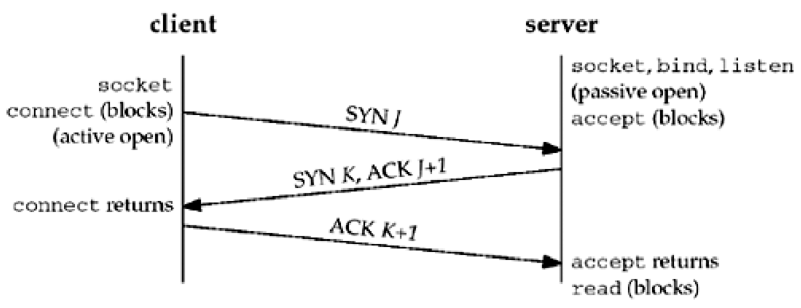
- When computers communicate via TCP, all received packets are acknowledged by sending back a packet with an ACK bit set. **The UDP protocol does not wait for any acknowledgement and is unable to detect any lost packets**.

  Does TCP send a SYN/ACK on every packet or only on the first connection?

Does TCP send a SYN/ACK on every packet or only on the first connection?

SYN is only at the beginning. ACK is on subsequent segments in either direction. The ACK will also define a window size. If the window size is 100 for example, the sender can send 100 segments before it expects to receive an ACK.

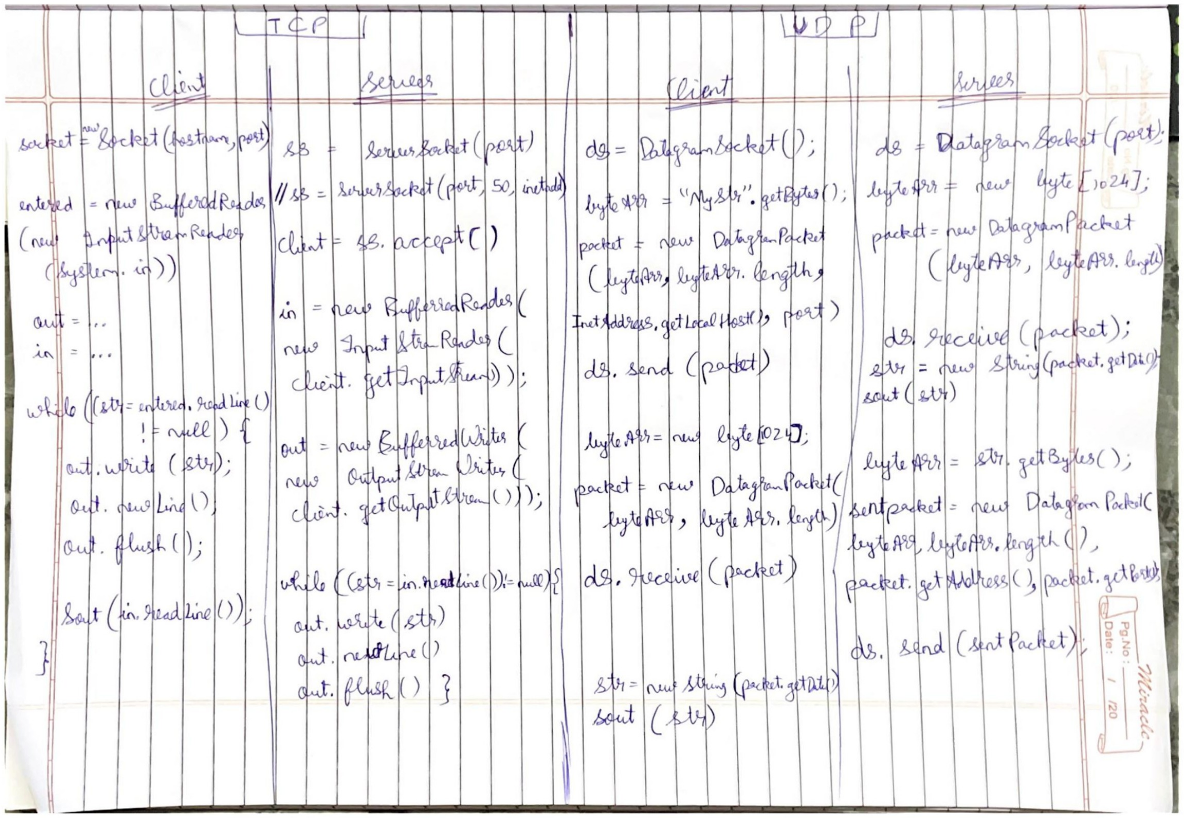
https://stackoverflow.com/questions/3604485/does-tcp-send-a-syn-ack-on-every-packet-or-only-on-the-first-connection



- Start client before server in a TCP connection: it would through exception

  Start client before server established in UDP: no error would be thrown

See this as a PDF.



TCP

Client

```
socket = new Socket(hostname, port)

entered = new BufferedReader
(new InputStreamReader
  (System.in))

out = ...
in = ...

while ((str = entered.readLine()
            != null) {
   out.write(str);
   out.newLine();
   out.flush();

   Sout(in.readLine());
}
```

Server

```
ss = ServerSocket(port)
// ss = ServerSocket(port, 50, inetAdd)

Client = ss.accept()

in = new BufferedReader(
 new InputStreamReader(
  client.getInputStream()));

out = new BufferedWriter(
 new OutputStreamWriter(
  client.getOutputStream()));

while ((str = in.readLine()) != null) {
   out.write(str)
   out.newLine()
   out.flush() }
```

UDP

Client

```
ds = DatagramSocket();

byteArr = "MyStr".getBytes();
packet = new DatagramPacket
 (byteArr, byteArr.length,
InetAddress.getLocalHost(), port)
ds.send(packet)

byteArr = new byte[1024];
packet = new DatagramPacket(
  byteArr, byteArr.length)
ds.receive(packet)

str = new String(packet.getData())
sout(str)
```

Server

```
ds = DatagramSocket(port);
byteArr = new byte[1024];
packet = new DatagramPacket
       (byteArr, byteArr.length)

 ds.receive(packet);
str = new String(packet.getData());
sout(str)

byteArr = str.getBytes();
sentpacket = new DatagramPacket(
byteArr, byteArr.length(),
packet.getAddress(), packet.getPort());

ds.send(sentPacket);
```

Pg.No: Date: / /20 Miracle

No much difference in **UDP** server and client. Both uses DatagramSocket and DatagramPacket.

In **TCP** server we use serversocket while at client side socket is used.

▼ Input and Output

https://docs.oracle.com/javase/tutorial/essential/io/index.html

See `com.java.language.basics.ByteStreamCopyFile.java`

See `com.java.language.basics.AddNumberInFile.java`

▼ Byte array size

Generally, we take it as 1024. Why?

Load testing might have resulted in this number. They might have tested for all the cases and this would have been the outcome which would have handled or performed best in most cases.

Eg.

```
byte[] receive = new byte[1024];

packet = new DatagramPacket(receive, receive.length);
```

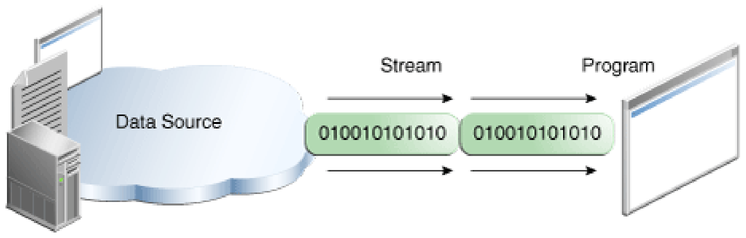
▼ I/O streams

An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

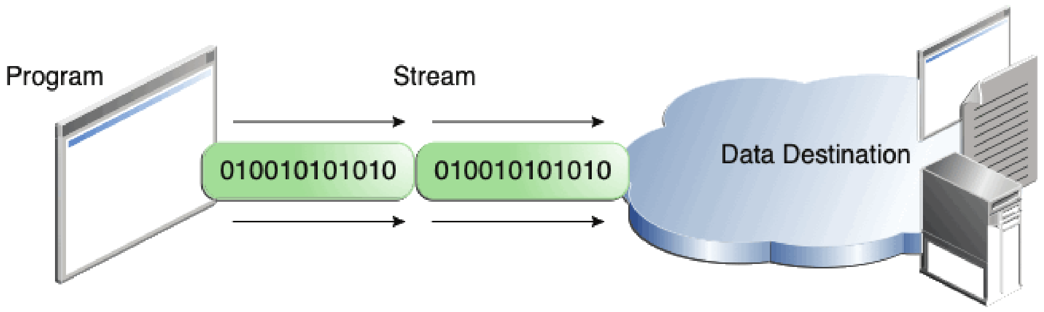
No matter how they work internally, **all streams present the same simple model to programs that use them: A stream is a sequence of data**. A program uses an *input stream* to read data from a source, one item at a time:

Reading information into a program.



A program uses an *output stream* to write data to a destination, one item at time:

Writing information from a program.



▼ Byte streams

Programs use *byte streams* to perform input and output of 8-bit bytes. All byte stream classes are descended from `InputStream` and `OutputStream`.

There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, `FileInputStream` and `FileOutputStream`. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

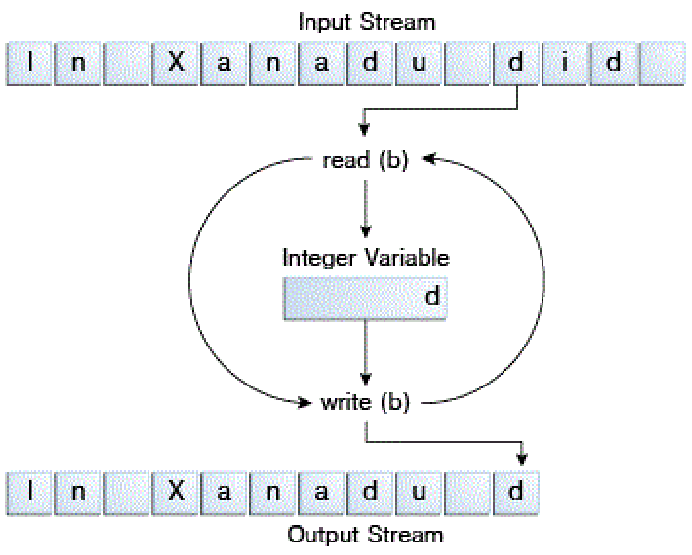
Using Byte Streams

We'll explore `FileInputStream` **and** `FileOutputStream` by examining an example program named `CopyBytes`, which uses byte streams to copy `xanadu.txt`, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {

        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

`CopyBytes` spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure. {A computer system normally stores characters using the ASCII code. Each character is stored using **eight bits** of information, giving a total number of 256 different characters (2**8 = 256).}

Simple byte stream input and output.



Always Close Streams

Closing a stream when it's no longer needed is very important — so important that `CopyBytes` uses a `finally` block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that `CopyBytes` was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial `null` value. That's why `CopyBytes` makes sure that each stream variable contains an object reference before invoking `close`.

When Not to Use Byte Streams

`CopyBytes` seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since `xanadu.txt` contains character data, the best approach is to use character streams, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? Because all other stream types are built on byte streams.

▼ Character streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. See the Internationalization trail for more information.

Using Character Streams

All character stream classes are descended from `Reader` and `Writer`. As with byte streams, there are character stream classes that specialize in file I/O: `FileReader` and `FileWriter`.
The `CopyCharacters` example illustrates these classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {

        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

`CopyCharacters` is very similar to `CopyBytes`. The most important difference is that `CopyCharacters` **uses `FileReader` and `FileWriter`** for input and output in place of `FileInputStream` and `FileOutputStream`. Notice that both `CopyBytes` and `CopyCharacters` use an `int` variable to read to and write from.

> However, in `CopyCharacters`, the `int` variable holds a character value in its last 16 bits; in `CopyBytes`, the `int` variable holds a `byte` value in its last 8 bits.

Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. `FileReader`, for example, uses `FileInputStream`, while `FileWriter` uses `FileOutputStream`.

There are two general-purpose byte-to-character "bridge" streams: `InputStreamReader` and `OutputStreamWriter`. Use them to create character streams when there are no prepackaged character stream classes that meet your needs. The sockets lesson in the networking trail shows how to create character streams from the byte streams provided by socket classes.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence (`"\r\n"`), a single carriage-return (`"\r"`), or a single line-feed (`"\n"`). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the `CopyCharacters` example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, `BufferedReader` and `PrintWriter`. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The `CopyLines` example invokes `BufferedReader.readLine` and `PrintWriter.println` to do input and output one line at a time.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Invoking `readLine` returns a line of text with the line. `CopyLines` outputs each line using `println`, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

▼ Buffered streams

Most of the examples we've seen so far use ***unbuffered* I/O. This means each read or write request is handled directly by the underlying OS.** This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

Native APIs are APIs of OS available with OS to execute input / output tasks. Your programming language is a layer above OS to interact with those APIs and buffered streams are created to use these Native APIs efficiently.

To reduce this kind of overhead, the Java platform implements *buffered* I/O streams. **Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty**. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the `CopyCharacters` example to use buffered I/O:

```
inputStream = new BufferedReader(new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams: `BufferedInputStream` and `BufferedOutputStream` create **buffered byte streams**, while `BufferedReader` and `BufferedWriter` create **buffered character streams**.

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as *flushing* the buffer.

Some buffered output classes support *autoflush*, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush `PrintWriter` object flushes the buffer on every invocation of `println` or `format`. See Formatting for more on these methods.

To flush a stream manually, invoke its `flush` method. The `flush` method is valid on any output stream, but has no effect unless the stream is buffered.

▼ Scanner

Objects of type `Scanner` are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for `Character.isWhitespace`.) To see how scanning works, let's look at `ScanXan`, a program that reads the individual words in `xanadu.txt` and prints them out, one per line.

```
import java.io.*;
import java.util.Scanner;
```

```
public class ScanXan {
    public static void main(String[] args) throws IOException {

        Scanner s = null;

        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));

            while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Notice that `ScanXan` invokes `Scanner`'s `close` method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

The output of `ScanXan` looks like this:

```
In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome
...
```

To use a different token separator, invoke `useDelimiter()`, specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke,

`s.useDelimiter(",\\s*");`

Translating Individual Tokens

The `ScanXan` example treats all input tokens as simple `String` values. `Scanner` also supports tokens for all of the Java language's primitive types (except for `char`), as well as `BigInteger` and `BigDecimal`. Also, numeric values can use thousands separators. Thus, in a `US` locale, `Scanner` correctly reads the string "32,767" as representing an integer value.

We have to mention the locale, because thousands separators and decimal symbols are locale specific. So, the following example would not work correctly in all locales if we didn't specify that the scanner should use the `US` locale. That's not something you usually have to worry about, because your input data usually comes from sources that use the same locale as you do. But this example is part of the Java Tutorial and gets distributed all over the world.

The `ScanSum` example reads a list of `double` values and adds them up. Here's the source:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;

public class ScanSum {
    public static void main(String[] args) throws IOException {

        Scanner s = null;
        double sum = 0;

        try {
            s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")));
            s.useLocale(Locale.US);

            while (s.hasNext()) {
                if (s.hasNextDouble()) {        //all numbers are double
                    sum += s.nextDouble();
                } else {
                    s.next();                   //this is for end of line
                }
            }
        } finally {
            s.close();
        }

        System.out.println(sum);
    }
}
```

And here's the sample input file, `usnumbers.txt`

```
8.5
32,767
3.14159
1,000,000.1
```

The output string is "1032778.74159". The period will be a different character in some locales, because `System.out` is a `PrintStream` object, and that class doesn't provide a way to override the default locale. We could override the locale for the whole program — or we could just use formatting, as described in the next topic, Formatting.

▼ Streams and Characters conversion

https://www.youtube.com/watch?v=iVbgcByqFic

A socket is an abstraction that you use to talk to something across the network. See diagram below...

In Java, to send data via the socket, you get an `OutputStream` (1) from it, and write to the `OutputStream` (you output some data).

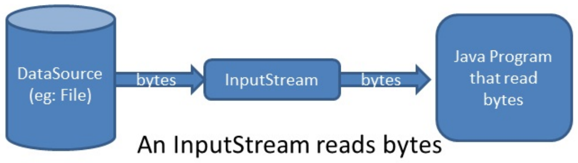
To read data from the socket, you get its `InputStream`, and read input from this second stream.

You can think of the streams as a pair of *one-way* pipes connected to a socket on the wall. What happens on the other side of the wall is not your problem!

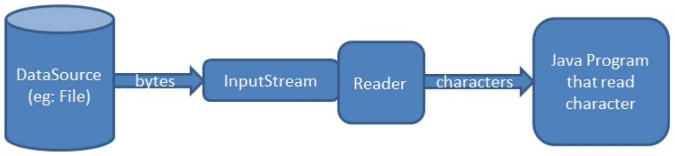
You can write to a **sockets input and output stream at the same time**.

Since the input stream and the output stream are separate objects within the Socket, the only thing you might concern yourself with is, what happens if you had 2 threads trying to read or write (two threads, same input/output stream) at the same time? **The read/write methods of the InputStream/OutputStream classes are not synchronized**.

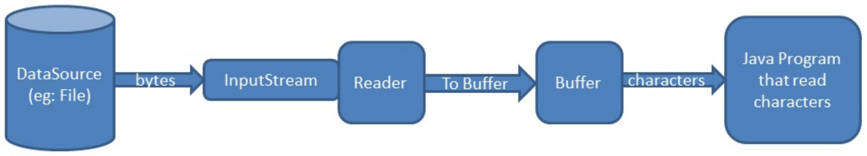
It is possible, however, that if you're using a sub-class of InputStream/OutputStream whose reading/writing methods you're calling are synchronized. You can check the javadoc for whatever class/methods you're calling, and find that out pretty quick.



An InputStream reads bytes



A Reader works with an InputStream and reads characters



A BufferedReader pre-fetches characters in to a buffer for performance

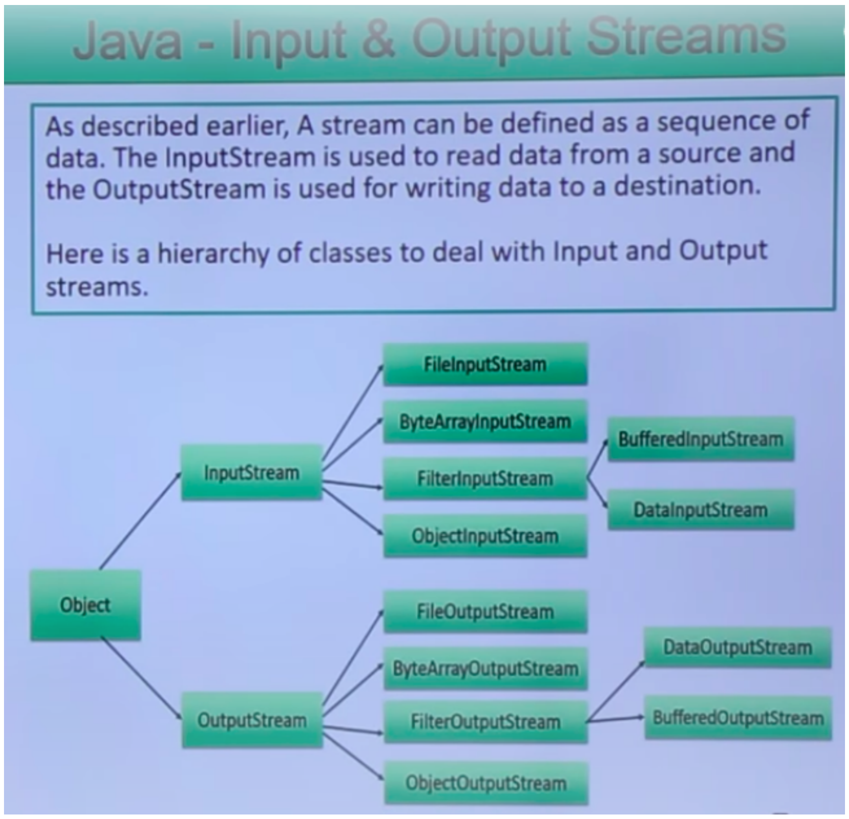
An **InputStreamReader** is **a bridge from byte streams to character streams**: It reads bytes and decodes them into characters using a specified charset . The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

BufferedReader

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

Without buffering, each invocation of read() [in read it reads one character in form of integer or in .read(b=new byte[1]) in this case it would read one byte] or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

```
/**
 * The "standard" input stream. This stream is already
 * open and ready to supply input data. Typically this stream
 * corresponds to keyboard input or another input source specified by
 * the host environment or user.
 */
public final static InputStream in = null;
```



Java - Input & Output Streams

As described earlier, A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.

```
package com.java.socket.programming.basics;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;

public class BufferedReaderInput
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println("Waiting for input");

            InputStream entered = System.in;                //we get bytes

            InputStreamReader inputStreamReader = new InputStreamReader(entered);   //converted to characters

            BufferedReader bufferedReader = new BufferedReader(inputStreamReader);  //creates buffer

            String string = bufferedReader.readLine();      //returns a string i.e. collection of characters

            System.out.println(string);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

▼ Readers (BufferedReader and Scanner)

BufferedReader is synchronized (it is thread-safe using locks) while Scanner is not. Scanner can parse primitive types and strings using regular expressions.

Scanner

Scanner is a class in java.util package **used for obtaining the input of the primitive types like int, double, etc. and strings**. It is the easiest way to read input in a Java program, though **not very efficient** if you want an input method for scenarios where time is a constraint like in competitive programming.

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

BufferedReader vs Scanner

Scanner and BufferReader both classes are used to read input from external system. Scanner is normally used **when we know input** is of type string or of primitive types and BufferReader is used to read text from character streams while buffering the characters for efficient reading of characters.

Sr. No.	Key	Scanner Class	BufferReader Class
1	Synchronous	Scanner is not syncronous in nature and should be used only in single threaded case.	BufferReader is syncronous in nature. During multithreading environment, BufferReader should be used.
2	Buffer Memory	Scanner has little buffer of 1 KB char buffer.	BufferReader has large buffer of 8KB byte Buffer as compared to Scanner.
3	Processing Speed	Scanner is bit slower as it need to parse data as well.	BufferReader is faster than Scanner as it only reads a character stream.
4	Methods	Scanner has methods like nextInt(), nextShort() etc.	BufferReader has methods like parseInt(), parseShort() etc.
5	Read Line	Scanner has method nextLine() to read a line.	BufferReader has method readLine() to read a line.

▼ Writers (BufferedWriter and PrintWriter)

BufferedWriter is better as it is efficient and also gives exception when found.

- The main reason to use the PrintWriter is to get access to the printXXX methods like println(). You can essentially use a PrintWriter to write to a file just like you would use System.out to write to the console.

  BUT A BufferedWriter is an efficient way to write to a file (or anything else), as it will buffer the characters in Java memory before (probably, depending on the implementation) dropping to C to do the writing to the file.

- `PrintWriter` gives more methods (`println`), but the most important (and worrying) difference to be aware of is that **it swallows exceptions**.

You can call `checkError` later on to see whether any errors have occurred, but typically you'd use `PrintWriter` for things like writing to the console - or in "quick 'n dirty" apps where you don't want to be bothered by exceptions (and where long-term reliability isn't an issue).

> There is nothing "PrintReader"; the closest you will get is probably java.util.Scanner.

▼ Main Differences Betwen Java NIO and IO

The table below summarizes the main differences between Java NIO and IO.

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

Stream Oriented vs. Buffer Oriented

The first big difference between Java NIO and IO is that IO is stream oriented, where NIO is buffer oriented. So, what does that mean?

Java IO being stream oriented means that you **read one or more bytes at a time**, from a stream. What you do with the read bytes is up to you. They are not cached anywhere. Furthermore, you **cannot move forth and back in the data in a stream**. If you need to move forth and back in the data read from a stream, you will need to cache it in a buffer first.

Java NIO's buffer oriented approach is slightly different. Data is read into a buffer from which it is later processed. You can move forth and back in the buffer as you need to. This gives you a bit more flexibility during processing. However, you also need to check if the buffer contains all the data you need in order to fully process it. And, you need to make sure that when reading more data into the buffer, you do not overwrite data in the buffer you have not yet processed.

Blocking vs. Non-blocking IO

Java IO's various streams are blocking. That means, that when a thread invokes a `read()` or `write()`, that thread is blocked until there is some data to read, or the data is fully written. The thread can do nothing else in the meantime.

Java NIO's non-blocking mode enables a thread to request reading data from a channel, and only get what is currently available, or nothing at all, if no data is currently available. Rather than remain blocked until data becomes available for reading, the thread can go on with something else.

The same is true for non-blocking writing. A thread can request that some data be written to a channel, **but not wait for it to be fully written**. The thread can then go on and do something else in the mean time.

What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

▼ Datagram

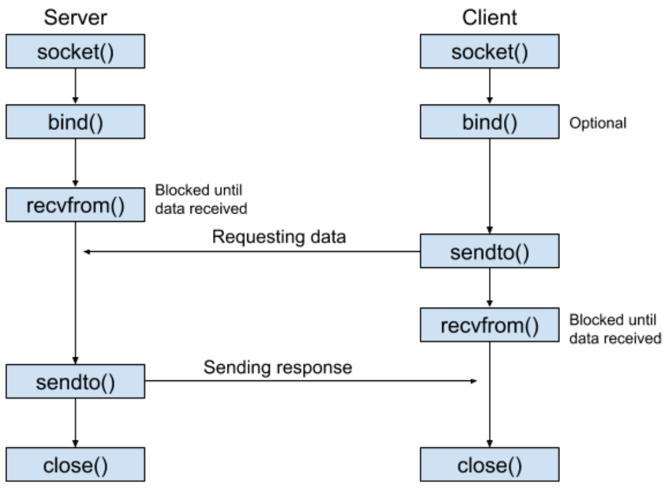
▼ What is datagram?

Clients and servers that communicate via a reliable channel, such as a TCP socket, have a dedicated point-to-point channel between themselves, or at least the illusion of one. To communicate, they establish a connection, transmit the data, and then close the connection. All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.

In contrast, 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.

Definition:

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



▼ Program

Code:

- In package echoUDPOneToOne
- In package squareUDPOneToOne

Important Points

- Server application makes a ServerSocket on a specific port. This starts our Server listening for client requests coming in the port.
- Then Server makes a new Socket to communicate with the client.

```
socket = server.accept()
```

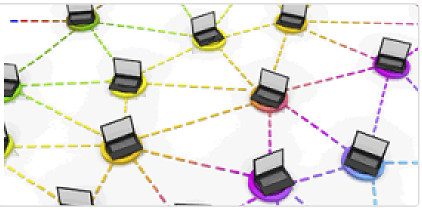
- The accept() method blocks(just sits there) until a client connects to the server.
- Then we take input from the socket using getInputStream() method.
- After we're done we close the connection by closing the socket and the input stream.
- To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

▼ Multicast using java socket

How to Multicast Using Java Sockets | Developer.com

The Java Socket APIs enable network communication between remote hosts in the client-server paradigm. The communication can be established in three ways: one-to-one communication (client-server), one-to-all communication (broadcast), and one-to-

https://www.developer.com/design/how-to-multicast-using-java-sockets/



▼ Client to Client without server

If you are using a TCP socket programming, you need central server to facilitate communication between clients.

Reason - You cannot connect to a port on one client from every other client. All clients can connect to one server on a particular port and server can facilitate the communication among clients.

HENCE, we are using UDP for that connection.

There is **no much difference between server and client of UDP**.

The UDP server:

- Runs first (to open a listen port)
- Opens a specific port
- Typically can accept multiple client connections

The UDP client:

- Runs second (assuming the server is already running)
- Indicates a specific target port
- Randomly (usually) choose a source port that the server will reply to
- Typically single-threaded and handles only one connection to the UDP server

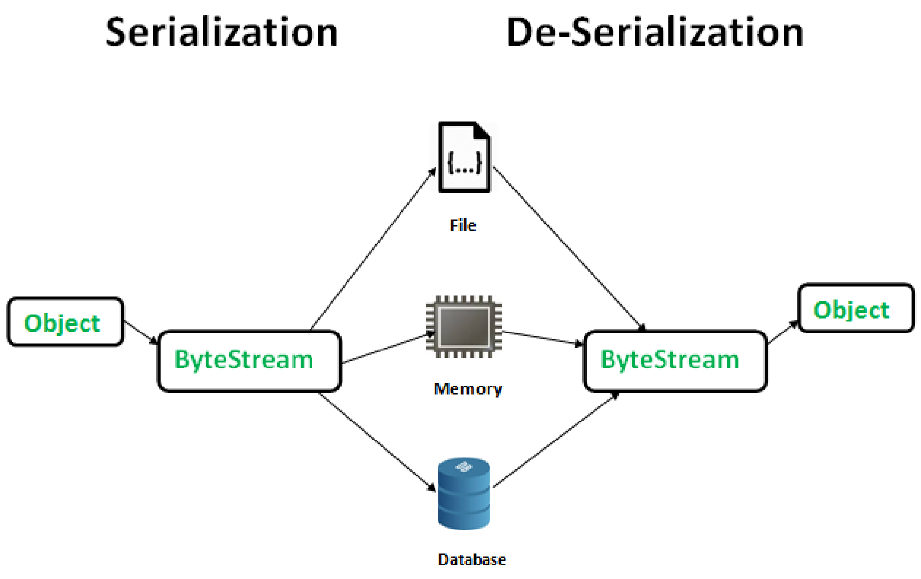
In fact, it's not that much different from TCP, except there is no three-way handshake nor any flow-control and congestion-control.

▼ Sharing objects using socket

Serialization is **the process of converting an object into a stream of bytes to store the object or transmit it to memory**, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

A stream is **a sequence of objects that supports various methods which can be pipelined to produce the desired result**. The features of Java stream are – A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

There are two types of streams in Java: **byte and character**. When an I/O stream manages 8-bit bytes of raw binary data, it is called a byte stream. And, when the I/O stream manages 16-bit Unicode characters, it is called a character stream.



Reference:

how to send/receive Objects using sockets in java

When I execute my code in the CMD window, it doesn't work in Client mode, exactly at the line: ObjectInputStream ois = new ObjectInputStream(socket.getInputStream()); ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream()); the

https://stackoverflow.com/questions/27736175/how-to-send-receive-objects-using-sockets-in-java

https://www.infoworld.com/article/2077539/java-tip-40--object-transport-via-datagram-packets.html

▼ My code

See package: `echoObjectUDPManyToOne`

```
package com.java.socket.programming.echoObjectUDPManyToOne;
```

```
import java.io.*;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

public class Client
{
    public static void main(String[] args)
    {
        DatagramSocket datagramSocket = null;

        BufferedReader in = null;

        ObjectOutputStream objectOutputStream = null;

        ObjectInputStream objectInputStream = null;

        int port = 9002;

        try
        {
            datagramSocket = new DatagramSocket();

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

            System.out.println("Enter integer value: ");

            int valueInput = Integer.parseInt(in.readLine());

            System.out.println("Enter a string: ");

            String sendString = in.readLine();

            myObject writeObj = new myObject(valueInput, sendString);

            ByteArrayOutputStream outputStream = new ByteArrayOutputStream(1024);

            objectOutputStream = new ObjectOutputStream(new BufferedOutputStream(outputStream));

            objectOutputStream.flush();

            objectOutputStream.writeObject(writeObj);

            objectOutputStream.flush();

            byte[] sendData = outputStream.toByteArray();

            DatagramPacket sentPacket = new DatagramPacket(sendData, sendData.length,
                    InetAddress.getLocalHost(), port);

            datagramSocket.send(sentPacket);

            //Once sent the server is sending the same packet back
            //so no need to understand that code.

            byte[] receiveData = new byte[1024];

            DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);

            datagramSocket.receive(receivePacket);

            ByteArrayInputStream inputStream = new ByteArrayInputStream(receiveData);

            objectInputStream = new ObjectInputStream(new BufferedInputStream(inputStream));
            myObject readObject = (myObject) objectInputStream.readObject();

            System.out.println("Received object:");

            System.out.println(readObject);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        finally
        {

            if (datagramSocket != null)
            {
                datagramSocket.close();
            }
            try
            {
                if (objectInputStream != null)
                {
                    objectInputStream.close();
                }
                if (objectOutputStream != null)
                {
                    objectOutputStream.close();
                }
                if (in != null)
                {
                    in.close();
                }
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }

        }
    }
}

/*
package com.java.socket.programming.echoObjectUDPManyToOne;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

public class Client
{
    public static void main(String[] args)
    {
        DatagramSocket datagramSocket = null;

        BufferedReader in = null;

        int port = 9002;
        try
        {
            datagramSocket = new DatagramSocket();

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

            System.out.println("Enter integer value: ");

            String str = in.readLine();

            byte[] sent = str.getBytes();

            DatagramPacket sentPacket = new DatagramPacket(sent, sent.length,
                    InetAddress.getLocalHost(), port);

            datagramSocket.send(sentPacket);

            DatagramPacket receivePacket = new DatagramPacket(sent, sent.length);

            datagramSocket.receive(receivePacket);

            str = new String(receivePacket.getData());

            System.out.println(str);
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
        finally
        {

            if (datagramSocket != null)
            {
                datagramSocket.close();
            }
            try
            {
                if (in != null)
                {
                    in.close();
                }
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }

        }
    }
}
 */
```

▼ Reference code

ClientClass:

```
package client;

// Organize imports

public class Client {
    public static void main(String[] args) throws UnknownHostException,
            IOException, ClassNotFoundException {
        System.out.println("welcome client");
        Socket socket = new Socket("localhost", 4444);
        System.out.println("Client connected");
        ObjectOutputStream os = new ObjectOutputStream(socket.getOutputStream());
        System.out.println("Ok");
        Message message = new Message(new Integer(15), new Integer(32));
        os.writeObject(message);
        System.out.println("Envoi des informations au serveur ...");

        ObjectInputStream is = new ObjectInputStream(socket.getInputStream());
        Message returnMessage = (Message) is.readObject();
        System.out.println("return Message is=" + returnMessage);
        socket.close();
    }
}
```

Server

Here is the code of the `Server` class

```
package Sockets;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {
    public static final int port = 4444;
    private ServerSocket ss = null;

    public void  runServer() throws IOException, ClassNotFoundException{
        ss = new ServerSocket(port);
        System.out.println("le systeme est pret pour accepter les connexions");
        Socket socket = ss.accept();
        ObjectInputStream is = new ObjectInputStream(socket.getInputStream());
        ObjectOutputStream os = new ObjectOutputStream(socket.getOutputStream());

        Message m = (Message) is.readObject();
        doSomething(m);

        os.writeObject(m);
        socket.close();
    }

    private void doSomething(Message m) {
        m.setResult(new Integer(m.getA().intValue()*m.getB().intValue()));
    }

    public static void main(String[] args) throws ClassNotFoundException, IOException {
        new Server().runServer();
    }
}
```

Message

And here is my object , I made it `Serializable` by implementing the interface `Serializable`

```
import java.io.Serializable;

public class Message implements Serializable {
    private static final long serialVersionUID = -5399605122490343339L;

    private Integer A;
    private Integer B;
    private Integer Result;

    public Message(Integer firstNumber, Integer secondNumber ){
        this.A = firstNumber;
        this.B = secondNumber;
    }

    public Integer getA() {
        return A;
    }

    public Integer getB() {
        return B;
    }

    public void setResult(Integer X)  {
        Result = X;
    }
}
```

> **serialVersionUID**
> The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException. A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long:
> static final long serialVersionUID = 42L;

> If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization.

▼ How whatsapp works?

WhatsApp uses client-server which enables a user to communicate. Here, basically, the user who wants to send a message sends the text or media file to the server. Then the server queues the message on the receiving number. The receiver will get a notification when they have an internet connection. Once the receiver gets the message it is instantly deleted from the server. This helps the server to keep WhatsApp's resources to the minimum. The data transfer on WhatsApp is an end to end encrypted which means the messages don't get stored on the server. These messages are only decoded when the receiver gets the message.

Port forwarding

Port Forwarding Explained

This is an animated port forwarding tutorial. It explains how to port forward and it also explains what are network ports.#portforwarding #portsWifi router ...

https://www.youtube.com/watch?v=2G1ueMDgwxw

▼ Netty

Netty Tutorial

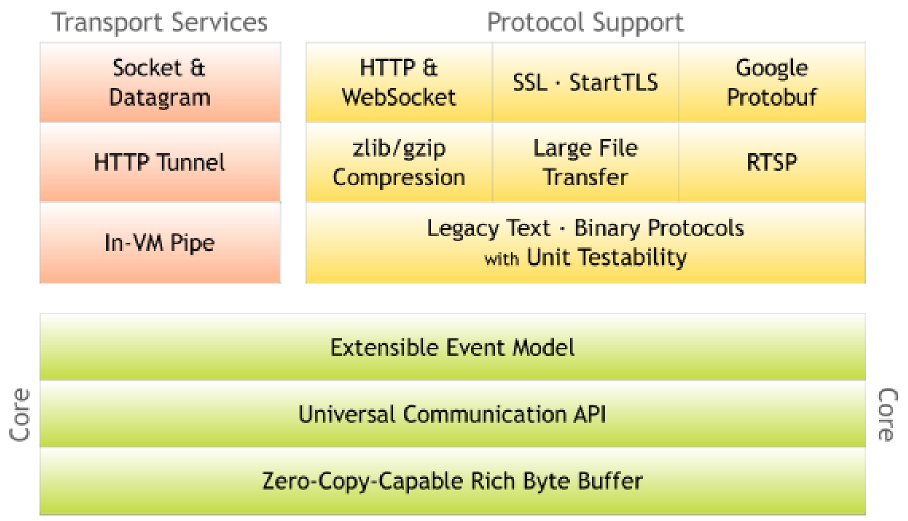
Netty is a high performance IO toolkit for Java. Netty is open source, so you can use it freely, and even contribute to it if you want to. This Netty tutorial will explain how Netty works, and how to get started with Netty. This tutorial will not cover every single detail of

http://tutorials.jenkov.com/netty/index.html

Reference video: https://www.youtube.com/watch?v=tsz-assb1X8

Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients. It greatly simplifies and streamlines network programming such as TCP and UDP socket server.

'Quick and easy' doesn't mean that a resulting application will suffer from a maintainability or a performance issue. Netty has been designed carefully with the experiences earned from the implementation of a lot of protocols such as FTP, SMTP, HTTP, and various binary and text-based legacy protocols. As a result, Netty has succeeded to find a way to achieve ease of development, performance, stability, and flexibility without a compromise.



Netty Advantages

In general, Netty makes it a lot easier to build scalable, robust networked applications compared to implemeting the same using standard Java. Netty also contains some OS specific optimizations, like using EPOLL on Linux etc.

Understanding Netty is Important

Even though Netty is fairly easy to use, it is necessary to understand how Netty works internally. Netty uses a single-threaded concurrency model, and is designed around non-blocking IO. This results in a significantly different programming model than when implementing Java EE applications. It takes a while getting used to, but once you get the hang out it, it's not too big a hazzle to work with.

Bootstrap

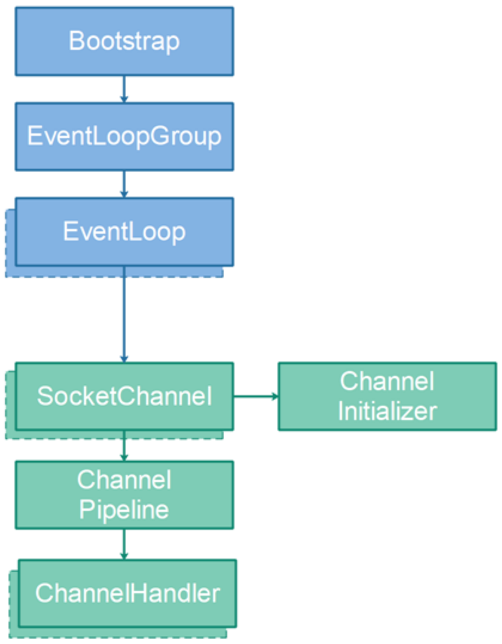
In computing, a bootstrap loader is the first piece of code that runs when a machine starts, and is responsible for loading the rest of the operating system. In modern computers it's stored in ROM, but I recall the bootstrap process on the PDP-11, where you would poke bits via the front-panel switches to load a particular disk segment into memory, and then run it. Needless to say, the bootstrap loader is normally pretty small.

In order to understand how Netty works it is useful to get an overview of Netty's internal design. There are several central concepts you need to know about. These are:

- Bootstrap
- EventLoopGroup
- EventLoop
- SocketChannel
- ChannelInitializer

- ChannelPipeline
- ChannelHandler

How these concepts are related to each other is illustrated here:



Each of these parts of Netty's design will be described in more detail in the following sections.

Bootstrap

The `Bootstrap` classes in Netty take care of bootstrapping Netty. The bootstrapping process includes starting threads, opening sockets etc. Bootstrapping will be explained in more detail in its own tutorial.

EventLoopGroup

A Netty `EventLoopGroup` is a group of `EventLoop`'s . Multiple `EventLoop`'s can be grouped together. This way the `EventLoop` shares some resources like threads etc.

EventLoop

A Netty `EventLoop` is a **loop that keeps looking for new events**, e.g. incoming data from network sockets (from `SocketChannel)` instances). When an event occurs, the event is passed on to the appropriate event handler, for instance a `ChannelHandler`.

SocketChannel

A Netty `SocketChannel` represents a **TCP connection to another computer** over a network. Whether you are using Netty as client or server, all data **exchanged** with other computers on the network are passed through a `SocketChannel` instance **representing the TCP connection** between the computers.

A `SocketChannel` is managed by an `EventLoop`, and always only by that same `EventLoop`. Since an `EventLoop` is always executed by the same thread, a `SocketChannel` instance is also only accessed by the same thread. Therefore you don't have to worry about synchronization when reading from a `SocketChannel`.

ChannelInitializer

A Netty `ChannelInitializer` is a special `ChannelHandler` which is attached to the `ChannelPipeline` of a `SocketChannel` when the `SocketChannel` is created. The `ChannelInitializer` is then called so it can initialize the `SocketChannel`.

After initializing the `SocketChannel` the `ChannelInitializer` removes itself from the `ChannelPipeline`.

ChannelPipeline

1. Each Netty `SocketChannel` has a `ChannelPipeline`.
2. The `ChannelPipeline` contains a list of `ChannelHandler` instances.
3. When the `EventLoop` reads data from a `SocketChannel` the data is passed to the first `ChannelHandler` in the `ChannelPipeline`. The first `ChannelHandler` processes the data and can choose to forward it to the next `ChannelHandler` in the `ChannelPipeline`, which then also processes the data and can choose to forward it to the next `ChannelHandler` in the `ChannelPipeline` etc.
4. When writing data out to a `SocketChannel` the written data is also passed through the `ChannelPipeline` before finally being written to the `SocketChannel`. Exactly how this works will be described in a separate tutorial.

ChannelHandler

A Netty `ChannelHandler` handles the data that is received from a Netty `SocketChannel`. A `ChannelHandler` can also handle data that is being written out to a `SocketChannel`. Exactly how this works will be described in a separate tutorial.

▼ Web socket

WebSocket provides an alternative to the limitation of **efficient communication between the server and the web browser** by providing bi-directional, full-duplex, real-time client/server communications. **The server can send data to the client at any time (while in http the client requests and then the server**

responds). Because it runs over TCP, it also provides a low-latency low-level communication and reduces the overhead of each message**.**

▼ Difference between http and web socket

Web applications were originally developed around a client/server model, where the Web **client is always the initiator of transactions**, requesting data from the server. Thus, there was **no mechanism for the server to independently send, or push, data to the client** without the client first making a request.

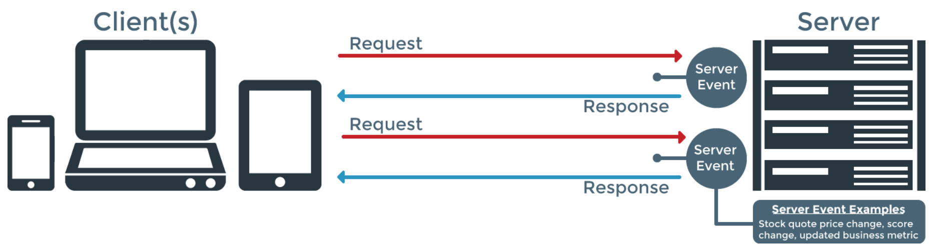
To overcome this deficiency, Web app developers can implement a technique called HTTP **long polling**, where the client polls the server requesting new information with a long timeout period and the server uses that long timeout to push the data. The server holds the request open until new data is available. Once available, the server responds and sends the new information. When the **client receives the new information, it immediately sends another request**, and the operation is repeated. This effectively emulates a server push feature. A simple diagram below:

WebSockets, on the other hand, allow for sending message-based data, similar to UDP, but with the reliability of TCP. **WebSocket uses HTTP as the initial transport mechanism, but keeps the TCP connection alive after the HTTP response is received** so that it can be used for sending messages between client and server. WebSockets allow us to build “real-time” applications without the use of long-polling.

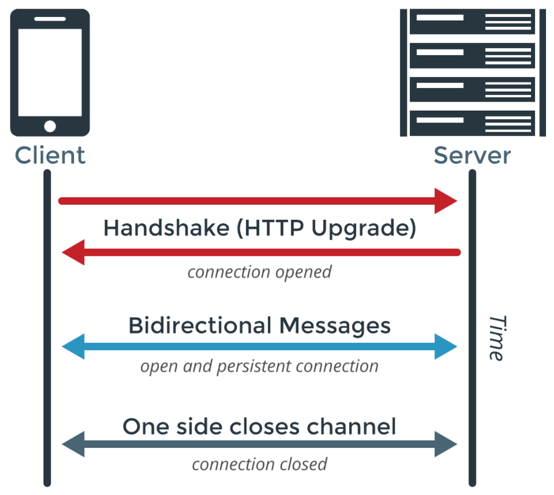
Differences between HTTP and WebSocket Connection:

HTTP Connection	WebSocket Connection
The HTTP protocol is a unidirectional protocol that works on top of TCP protocol which is a connection-oriented transport layer protocol, we can create the connection by using HTTP request methods after getting the response HTTP connection get closed.	WebSocket is a bidirectional communication protocol that can send the data from the client to the server or from the server to the client by **reusing the established connection channel**. The connection is kept alive until terminated by either the client or the server.
Simple RESTful application uses HTTP protocol which is stateless.	Almost all the real-time applications like (trading, monitoring, notification) services use WebSocket to receive the data on a single communication channel.
When we do not want to retain a connection for a particular amount of time or reuse the connection for transmitting data; An HTTP connection is slower than WebSockets.	All the frequently updated applications used WebSocket because it is faster than HTTP Connection.

HTTP Long Polling Diagram



WebSockets Diagram



▼ @ServerEndpoint

```
public @interface ServerEndpoint
```

This class level annotation declares that the class it decorates is a web socket endpoint that will be deployed and made available in the URI-space of a web socket server. The annotation allows the developer to define the URL (or URI template) which this endpoint will be published, and other important properties of the endpoint to the websocket runtime, such as the encoders it uses to send messages.

A server can open WebSocket connections with multiple clients—even multiple connections with the same client. It can then message one, some, or all of these clients. Practically, this means multiple people can connect to our chat app, and we can message some of them at a time.

▼ Events

Once the socket is created, we should listen to events on it. There are totally 4 events:

- `open` – connection established,
- `message` – data received,
- `error` – websocket error,
- `close` – connection closed.

- *@ServerEndpoint:* If decorated with *@ServerEndpoint,* the **container ensures availability of the class as a *WebSocket* server listening to a specific URI space**
- *@ClientEndpoint*: A class decorated with this annotation is treated as a *WebSocket* client

- *@OnOpen*: A Java method with *@OnOpen* is invoked by the container when a new *WebSocket* connection is initiated
- *@OnMessage*: A Java method, annotated with *@OnMessage,* **receives the information from the *WebSocket* container** when a message is sent to the endpoint
- *@OnError*: A method with *@OnError* is invoked when there is a problem with the communication
- *@OnClose*: Used to decorate a Java method that is called by the container when the *WebSocket* connection closes

When a new user logs in (*@OnOpen*) is immediately mapped to a data structure of active users. Then, a message is created and sent to all endpoints using the *broadcast* method.

This method is also used whenever a new message is sent (*@OnMessage*) by any of the users connected – this is the main purpose of the chat.

If at some point an error occurs, the method with the annotation *@OnError* handles it. You can use this method to log the information about the error and clear the endpoints.

Finally, when a user is no longer connected to the chat, the method *@OnClose* clears the endpoint and broadcasts to all users that a user has been disconnected.

https://www.baeldung.com/java-websockets

▼ Opening a websocket

When `new WebSocket(url)` is created, it starts connecting immediately.

During the connection the browser (using headers) asks the server: “Do you support Websocket?” And if the server replies “yes”, then the talk continues in WebSocket protocol, which is not HTTP at all.

Here’s an example of browser headers for request made by `new WebSocket("wss:// javascript.info/chat")`.

- [x] ~~Streams~~
- [x] ~~BufferedReader and scanner~~
- [x] ~~BufferedWriter and PrintWriter~~
- [x] ~~UDP one to one - string and integer~~
- [x] ~~TCP one to one - string and date&time~~
- [x] ~~UDP transfer object from multiple client to one server~~
- [x] ~~TCP date and time multiple client to one server~~
- [x] ~~Chat one to one UDP (for TCP server is must)~~
- [x] ~~Chat app using TCP~~
- [x] ~~Chat group~~
- [x] ~~server bootstrap socket tcp~~
- [x] ~~Web socket (java to javascript)~~