Contents

Net.io

Networking

The basics of networking consist of something called Sockets, the socket is a cornerstone element of the networking world, each machine can expose any number of socket connections to itself to allow for other machines to connect to it, or receive information from it. The Socket or also called `Berlkey` Socket was invented in the UNIX days back in the 1970s. The purpose of a Socket is to represent and endpoint for receiving and sending information (duplex, mode, allows it to do both at the same time). A Socket is defined by two main components - IP Address - provides a unique address for the device on the network, and a Port number - identifies a specific process or service running on that device.

The IP address of a machine is usually defined by the `IPv4` or `IPv6` versions, which are mostly the same with the only main difference being the range of addresses each can represent, while the `v4` can only represent 4 octets of - for a total of 32 bytes, or about 4 billion combinations, the `IPv6` can represent many more or about 126 bytes of information, ranging up to the trillions of unique addresses.

The basic socket lifetime starts with an application creating a socket, for example a web server, creates a socket on a specific port, meaning that this process or application will be listening to this port, for incoming connections and information, it can also send information on the same socket, which is usually the case for web servers. When a socket is created it is `bound` meaning that the underlying `operating system` starts to listen for incoming connections on that port.

When a client desires to connect to the serving application it has to create its own socket with the server's unique ip address and the port defining the underlying application on the server machine that the client wants to listen to for data. The operating system where the server runs then might accept or deny the connection to itself, further more the process running on that port (the web server) might itself deny the connection to that port unless some sort of condition is met. When the connection is established between the client and server sockets, then the communication process can start

There are two types of Sockets on the internet in wide use, both use different types of protocols to transmit data, while the IP protocol is used to transmit data on a lower level, it does not guarantee any data integrity it simply streams the data from one socket (server) to the other (client) or vice versa. To guarantee data transmission two types of protocols exist in wide spread use.

- TCP - Transmission Control Protocol to provide reliable, ordered and error checked delivery of a stream of bytes. This is mostly used in applications where the delivery of the information is crucial, web browsing, file transfers etc.
- UDP - User Datagram Protocol - unlike TCP the UDP is not streaming based protocol it is sending messages without establishing a connection, resulting in lower overhead, it is suitable for applications

that require fast real-time communication like video streaming, or online gaming, where data loss is acceptable.

`TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.`

InetAddress

That class is used to encapsulate both the numerical IP address and the domain name for that address. One can interact with the class by using the name of the IP host, which is more convinient and understandable than its IP addresss, The InetAddress can handle both IPv4 and IPv6. To create an instance of the InetAddress one can use one of the factory methods for that purpose, since the InetAddress has no exposed public constructor

```
static InetAddress getLocalHost() throws UnknownHostException
static InetAddress getByName(String hostName) throws UnknownHostException
static InetAddress[] getAllByName(String hostName) throws
  UnknownHostException
```

The static methods provide a way to create an instance from a given host name, or domain name. The internal implementation will use DNS lookup to make sure the actual IP address is discovered, for the given host, the way this works is that Java in this case is using the underlying operating system and requests it to resolve the given domain name to an IP address. The operating system is using the configured DNS server to contact the server and try to resolve the actual address.

How that works, is usually the ISP (internet service provider) will have default DNS servers which are automatically configured to a given machine when it is connected to the internet, this is done through DHCP (dynamic host configuration protocol). DHCP is a networking management protocol, it is the first exchange that a machine does when it is connected to the internet, it requests an IP address, and other configuration settings from the network, one of which is also the default DNS server address. This DNS is usually the one configured and owned by the ISP, which itself is configured to look up in the global DNS servers, but also might provide some specific features for the customer (like parental control, internet control panel server etc)

1. `Connection`: When you connect your device (like a computer, smartphone, or tablet) to a network (wired or wireless), it sends a broadcast message to the local network, requesting an IP address and other configuration settings.
2. `Discover`: The device sends a DHCP Discover message to find available DHCP servers on the network.
3. `Offer`: Any DHCP server on the network that receives the Discover message responds with a DHCP Offer message. This offer includes an available IP address, subnet mask, default gateway, and DNS server information.
4. `Request`: The device receives one or more DHCP offers and selects one of them. It then sends a DHCP Request message back to the selected server, indicating that it has accepted the offer.
5. `Acknowledgment`: The DHCP server acknowledges the request with a DHCP Acknowledgment (ACK) message, confirming that the device can use the provided IP address and other settings, including the DNS server information.
6. `Completion`: Once the DHCP ACK is received, the device configures itself with the IP address, subnet mask, default gateway, and DNS server(s) as specified by the DHCP server. The DNS server address is now set for use in resolving domain names to IP addresses.

Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of InetAddress were created: Inet4Address and Inet6Address. Inet4Address represents a traditional-style IPv4 address. Inet6Address encapsulates a newer IPv6 address. Because they are subclasses of InetAddress, an InetAddress reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use InetAddress when working with IP addresses because it can accommodate both styles.

Byte order - Endianness

The order of the bytes which are transmitted on a network, is also well defined, in the world of computing there are two ways to order bytes. Note that certain types of processors have different byte order representation internally, however this is often not exposed to the user and the operating system takes care of being consistent about this.

- `Big Endianness` - a multi byte value or data is ordered from the lowest to the highest bytes, in other words, the lower bytes of a multi byte value are put at a lower address than the higher ones meaning that the hex value of 0x123456 will be represented as the following 3 bytes - 12 34 56
- `Little Endianness` - a multi byte value or data is ordered from the highest to the lowest bytes, in other words, the higher bytes of a multi byte value are put at a lower address than the lower ones, meaning that the hex value of 0x123456 will be represented as the following 3 bytes 56 34 12

In the networking world the Big Endianness approach is used always, meaning that the order of the bytes in multi byte values is the same as its actual representation

Sockets

In java there are two types of sockets, meant for different purposes one is the `ServerSocket`, the other is the `Socket` class for clients. The main difference is that the `ServerSocket` is blocking since it is a listener, it is waiting for clients to connect to it, otherwise there is nobody to serve information to.

When a `Socket` (read client socket) is created, it implicitly establishes connection to the server socket on the provided address, meaning that there are no explicit methods that expose a `connect` action, one can start sending information on that socket if the creation is successful immediately.

```
Socket Socket(InetAddress host, int port, boolean stream) throws
   IOException // create a socket to the given inet address and port
Socket Socket(String hostName, int port) throws UnknownHostException,
   IOException // create a socket to the named host and port
Socket Socket(InetAddress ipAddress, int port) throws IOException //
   create a socket to the address and port
```

The example below provides a very minimal code snippet which connects to the public `whois` host/domain on port 43, then obtains the input and output streams from the socket which are used to send and then read back the information written back from the server. Note that the code below is using the output stream to write to the socket, and the input stream to read information back from the socket.

```
class Whois {
    public static void main(String args[]) throws Exception {
        int c;
        // Create a socket connected to internic.net, port 43.
        Socket s = new Socket("whois.internic.net", 43);
        // Obtain input and output streams.
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();
        // Construct a request string.
        String str = (args.length == 0 ? "MHProfessional.com" : args[0]) +
           "\n";
        // Convert to bytes.
        byte buf[] = str.getBytes();
        // Send request.
        out.write(buf);
        // Read and display response.
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
        // the output of this call might look something like that
        // Domain Name: MHPROFESSIONAL.COM
        // Registry Domain ID: 479181747_DOMAIN_COM-VRSN
        // Registrar WHOIS Server: whois.corporatedomains.com
        // Registrar URL: http://cscdbs.com
        // Updated Date: 2024-06-05T05:13:49Z
        // Creation Date: 2006-06-09T16:44:39Z
        // Registry Expiry Date: 2025-06-09T16:44:39Z
        // Registrar: CSC Corporate Domains, Inc.
        // Registrar IANA ID: 299
        // Registrar Abuse Contact Email: domainabuse@cscglobal.com
        // Registrar Abuse Contact Phone: 8887802723
        // Domain Status: clientTransferProhibited
           https://icann.org/epp#clientTransferProhibited
        // Name Server: PDNS85.ULTRADNS.BIZ
        // Name Server: PDNS85.ULTRADNS.COM
        // Name Server: PDNS85.ULTRADNS.NET
        // Name Server: PDNS85.ULTRADNS.ORG
        // DNSSEC: unsigned
        // URL of the ICANN Whois Inaccuracy Complaint Form:
           https://www.icann.org/wicf/
        s.close();
    }
}
```

WHOIS is a protocol used for querying databases that store registered domain names and IP addresses. It allows you to retrieve information about who owns a domain or an IP address, as well as registration dates, contact details, and the registrar responsible for the domain.

URL

The uniform resource locator is a way to locate a given resource on the internet as well as provide means to interact with that resource. The URL consists of a few components which are described below

- the scheme - the first part of the URL, specifies which protocol is to be used when the resource is accessed, there are many, but the ones in wide use such as http, https, file, gopher and ftp are well familiar to most people
- the host - this component is in the middle, it specifies the host or domain name where the resource can

be found, this is usually then passed down to the operating system through a DNS service to resole the actual ip address of the domain, but one can also include an IP directly in the URL instead of a domain name.

- the port - this one is optional, and is usually deduced, if omitted, based on the protocol that is used, for example the HTTP protocol or scheme reserves port 80 by default, HTTPS - reserves port 443, FTP reserves 23 and so on. That is why the very first 1024 ports are reserved for special purposes, like that

Protocol	Default Port	Example URL with Default Port
HTTP	80	`http://example.com`
HTTPS	443	`https://example.com`
FTP	21	`ftp://ftp.example.com`
File	N/A	`file:///C:/example.txt`
SFTP	22	`sftp://sftp.example.com`
SMTP	25, 587	`smtp://mail.example.com`
Telnet	23	`telnet://telnet.example.com`
WebSocket	80, 443	`ws://example.com`

The java networking package provides means of constructing URL objects and instances, to help one work with them, the base class is called URL, the class has overloaded constructors to create an url from a variety of sources, mostly strings.

Constructor	Description
URL(String spec)	Creates a `URL` object from a string representing the full URL (e.g., “https://www.example.com”).
URL(String protocol, String host, String path)	Creates a `URL` from a protocol, host, and path (the path includes the path and query string).
URL(String protocol, String host, int port, String path)	Creates a `URL` by specifying the protocol, host, port, and path.
URL(URL context, String spec)	Creates a `URL` by resolving a relative URL spec within a base URL context.
URL(URL context, String spec, URLStreamHandler handler)	Creates a `URL` object with a base context and custom stream handler for managing protocols.

`Note that all of those constructor calls might throw MalformedURLException, which might occur when an invalid url format string was provided to the constructor arguments`

```java
class URLDemo {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.HerbSchildt.com/WhatsNew");
            System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());
        System.out.println("Host: " + hp.getHost());
        System.out.println("Path: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
    }
}
```

When you run this, you will get the following output.

```
Protocol: http
Port: -1
Host: www.HerbSchildt.com
Path: /WhatsNew
Ext:http://www.HerbSchildt.com/WhatsNew
```

The `URLConnection` is a special type of object that can be obtained from an URL instance object, by calling the `openConnection` method, which will create an instance of `URLConnection` object/instance to actually establish a proper connection internally, this method initiates the process of establishing a connection to the URL, often by creating a socket connection (especially for HTTP/HTTPS, FTP, etc.). The socket is responsible for sending and receiving data to and from the resource's server.

`Based on the type of protocol used URLConnection is a high level abstraction layer which uses protocol handlers for example for basic HTTP and HTTPS requests a wrapper around the Socket API is used, for other schemes and protocols different handlers are used, in essence it allows one to abstract away most of the cruft when trying to communicate with a resource server which requires the communication to be done through the use of specific protocol such as http, https, ftp and so on.`

Connection Establishment happens when the socket is created when you call `connect()`, `getInputStream()`, or `getOutputStream()` on the `URLConnection`. This establishes a communication link between your application and the remote server.

Making a regular get request, fetching data from the target URL

```java
URL url = new URL("http://www.example.com");
URLConnection urlConnection = url.openConnection();
InputStream inputStream = urlConnection.getInputStream();
BufferedReader reader = new BufferedReader(new
   InputStreamReader(inputStream));

String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
reader.close();
```

Making a POST request, sending data to the target URL

```java
URL url = new URL("http://www.example.com/login");
HttpURLConnection httpURLConnection = (HttpURLConnection)
   url.openConnection();
httpURLConnection.setRequestMethod("POST");
httpURLConnection.setDoOutput(true);
httpURLConnection.setRequestProperty("Content-Type",
   "application/x-www-form-urlencoded");
String postData = "username=user&password=pass";
try (OutputStream os = httpURLConnection.getOutputStream()) {
    byte[] input = postData.getBytes("utf-8");
    os.write(input, 0, input.length);
}
```

Below are listed some of the classes which implement the `URLConnection`, each of which takes care of knowing how to communicate and connect to URLs with different schemes and communication protocols.

Class Name	Description
HttpURLConnection	allows for HTTP-specific features, such as handling different HTTP methods (GET, POST, etc.) and managing HTTP headers and responses.
HttpsURLConnection	extends off of HttpURLConnection and supports secure HTTPS connections, enabling SSL/TLS communication.
JarURLConnection	for accessing resources in JAR files, providing a way to read entries from a JAR file.
FileURLConnection	for accessing local files via the `file://` protocol, allowing file reading.
FtpURLConnection	for accessing resources via the FTP protocol, enabling file transfers over FTP.
DataSourceURLConnection	designed for reading data from `DataSource` objects, commonly used in JavaMail for reading resources.

URI

The URI class or type is actually a super-type, to the URL type. The URI describes a unique resource identifier alone. While the URL describes not only the unique resource identifier or locator but also how to access this resource. That can also be seen in the name of both types. While the URI - is identifier, the URL - is locator. The difference is quite is that URLs provide information such as the schema for example, which tells the clients using an URL how (the locator) to access the given resource, not just where (the identifier)

```
An URI that contains a scheme locator is by definition classified as an URL, https://www.exampl
a URI that contains a name identifier is called URN or unique resource name, urn:isbn:12345
```

Cookie

Cookies provide a way to establish a `stateful` connection between a server and a client, a cookie is usually send over on each request, and it represents some sort of unique identifier which the server can use to identify a client by. On each request/response cycle between the server and the client, the server would send a cookie header along with the response, and the client will send back the same cookie with the requests to the server, that way the server can identify different clients uniquely

Besides regular cookies, there usually companion cookies such as `CSRF` tokens which are used to secure the cookies making sure that a cookie that is sent back to the server is coming from the actual client, instead of malicious actions or actors.

The Cookie class is under the java class called `HttpCookie` which contains the implementation for a generic `http` cookie that can be used along with the `CookieStore` interface and the `CookieManager` which serves as a bridge between `HttpURLConnection` and an `HttpCookie`, the `CookieManager` does use a `CookieStore` implementation internally to store and manage the cookies for many `HttpURLConnection` responses and requests. One would typically set a default `CookieHandler`, like so `CookieHandler.setDefault(new CookieManager())` this would make sure that all requests and responses by default are routed to the target `CookieManger` instance.

The actual link between the `HttpURLConnection` is done in it's actual default platform specific implementation under `sun.net.www.protocol.http.HttpURLConnection`, which is internally using the `CookieHandler.getDefault()` to extract the default `CookieManager`, this is the bridge / glue that makes sure that each `HttpURLConnection` is correctly being populated with cookies

```
HttpCookie HttpCookie(String name, String value, String header, long
    creationTime)
HttpCookie HttpCookie(String name, String value, String header)
HttpCookie HttpCookie(String name, String value);

HttpURLConnection HttpURLConnection(URL u, String host, int port)
    throws IOException
    HttpURLConnection HttpURLConnection (URL u)
```

```
 *                    use
 * CookieHandler <------- HttpURLConnection
 *       ^
 *       | impl
 *       |          use
 * CookieManager -------> CookiePolicy
 *             |   use
 *             |--------> HttpCookie
 *             |              ^
 *             |              | use
 *             |   use        |
 *             |--------> CookieStore
 *                            ^
 *                            | impl
 *                            |
 *                    Internal in-memory implementation
```

```
// Set up a CookieManager with a default CookieStore
CookieManager cookieManager = new CookieManager();
CookieHandler.setDefault(cookieManager);

// Make an HTTP connection
URL url = new URL("https://example.com");
HttpURLConnection connection = (HttpURLConnection) url.openConnection();

// Send a request (cookies from the store are automatically added if
   applicable)
connection.setRequestMethod("GET");

// Receive the response
int responseCode = connection.getResponseCode();

// Check for any cookies sent by the server (they are automatically added
   to the store)
Map<String, List<String>> headers = connection.getHeaderFields();
for (String header : headers.keySet()) {
    if ("Set-Cookie".equalsIgnoreCase(header)) {
        List<String> cookies = headers.get(header);
        for (String cookie : cookies) {
            System.out.println("Received cookie: " + cookie);
        }
    }
}

// Retrieve cookies from the store for the same URL
URI uri = url.toURI();
CookieStore store = cookieManager.getCookieStore();
List<HttpCookie> storedCookies = store.get(uri);
for (HttpCookie storedCookie : storedCookies) {
    System.out.println("Stored cookie: " + storedCookie);
}
```

CSRF Those special types of tokens are an essential security mechanism to protect web applications from unauthorized, malicious actions initiated by third-party sites. By embedding a unique, unpredictable token in each state-changing request and validating it on the server side, web applications can ensure that only legitimate actions performed by the authenticated user are allowed. CSRF tokens work because they are tied to both the user's session and the specific site from which the form is sent. Since an attacker's website cannot access the legitimate site's CSRF token, they cannot include the correct token in any malicious request they try to initiate.

1. When the user loads the page, the server sets two cookies: one HTTP-only cookie for the session, and one HTTP-only cookie for the CSRF token.
2. When the user makes an API call (via fetch() or XHR), the Browser itself reads the HTTP-only CSRF token cookie and includes it in a header (X-CSRF-Token).
3. The server verifies both the session (via the regular HTTP-only cookie) and the CSRF token (via X-CSRF-Token header) to ensure that the request is legitimate.

CORS Is a security feature built on top of the Same-Origin Policy, and it allows web servers to explicitly permit cross-origin requests. When a website needs to make a request to another domain, the browser uses CORS to check whether the server hosting the resource allows that request to be made. The browser automatically determines if a request is cross-origin based on the origin (protocol, domain, and port) of the requesting page and the requested resource. If the server doesn't respond with the proper CORS headers or explicitly denies cross-origin access, the browser blocks the response from being made available to JavaScript. The request might still be sent, but the browser will refuse to pass the response data back to the page's JavaScript if the CORS policy isn't met.

1. When a script (like an AJAX call or a fetch() request) on a web page tries to request a resource from a different origin, the browser first checks if the request adheres to the Same-Origin Policy.
2. If the request is cross-origin, the browser sends a special HTTP request known as a preflight request (for certain methods) to the server, asking if the actual request is allowed.

   ```
   OPTIONS /resource HTTP/1.1
   Host: http://original.com
   Origin: https://example.com
   Access-Control-Request-Method: POST
   Access-Control-Request-Headers: X-PINGOTHER, Content-Type
   ```

3. The preflight request asks the server if it allows cross-origin requests by sending headers like Access-Control-Request-Method and Access-Control-Request-Headers.
4. If the server is configured to allow cross-origin requests, it responds with CORS headers like Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers.

   ```
   HTTP/1.1 204 No Content
   Access-Control-Allow-Origin: https://example.com
   Access-Control-Allow-Methods: POST, GET, OPTIONS
   Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
   ```

5. The browser checks the CORS headers returned by the server to ensure the request is allowed. If the headers are missing or don't match the request (e.g., the origin isn't allowed), the browser will block the response and not allow the JavaScript on the page to read the response.

Datagram

Unlike the regular `Socket` which is meant for regular `TCP` packet communication in java, the `Datagram` socket implementation is meant to be used with `UDP` style packets, the `Datagram api` provides multiple ways to construct a `datagram` socket, but the gist is that the constructors are pretty much the same as for the regular `TCP` sockets, which either can accept a port or/and host address, the `Datagram` socket can also be created without any constructor arguments which would bind to the first available socket on the machine that is not reserved

```
DatagramSocket DatagramSocket( ) throws SocketException
DatagramSocket DatagramSocket(int port) throws SocketException
DatagramSocket DatagramSocket(int port, InetAddress ipAddress) throws
   SocketException
DatagramSocket DatagramSocket(SocketAddress address) throws SocketException
```

```
void send(DatagramPacket packet) throws IOException
void receive(DatagramPacket packet) throws IOException
```

The `DatagramPacket` is meant to receive information from a DatagramSocket, from the constructor of the DatagramPacket below one can see that the information must be stored in a temporary buffer which is filled when a packet is either sent or received.

```
DatagramPacket DatagramPacket(byte data [], int size)
DatagramPacket DatagramPacket(byte data [], int offset, int size)
DatagramPacket DatagramPacket(byte data [], int size, InetAddress
   ipAddress, int port)
DatagramPacket DatagramPacket(byte data [], int offset, int size,
   InetAddress ipAddress, int port)
```

`From the constructors above, the first two constructors are used when receiving information from a`DatagramSocket`, while the last two are used when sending, since the recipient has to be known host, therefore a correct`InetAddresss`has to be provided otherwise there would be no way to know where to send the information to`

```
class WriteServer {
    public static int serverPort = 998;
    public static int clientPort = 999;
    public static int bufferSize = 1024;

    public static DatagramSocket ds;
    public static byte buffer[] = new byte[bufferSize];

    public static void TheServer() throws Exception {
        int pos=0;
        while (true) {
            // collect input from the server on stdin
            int c = System.in.read();
            switch (c) {
                case -1:
                    System.out.println("Server Quits.");
                    ds.close();
                    return;
                case '\r':
                    break;
                case '\n':
                    // send whatever was accumulated in the buffer to the
                       client
                    ds.send(new DatagramPacket(buffer,pos,
                        InetAddress.getLocalHost(),clientPort));
                    // reset the pointer, back to 0, no data is in the
                       buffer
                    pos=0;
                    break;
                default:
                    // write each byte to the buffer from stdid, and
                       increment the pointer
                    buffer[pos++] = (byte) c;
            }
        }
    }
    public static void TheClient() throws Exception {
        while(true) {
            // receive from the server, into the buffer, at most
               `bufferSize`
            DatagramPacket p = new DatagramPacket(buffer, buffer.length);
            ds.receive(p);
            // print the received information to the screen, that would be
               whatever was entered on the server side
            System.out.println(new String(p.getData(), 0, p.getLength()));
        }
    }
    public static void main(String args[]) throws Exception {
        // based on the passed in number arguments start either a server
           emitting packets or a client receiving them
        if(args.length == 1) {
            // start a server type of application
            ds = new DatagramSocket(serverPort);
            TheServer();
        } else {
            // start a client type of application
            ds = new DatagramSocket(clientPort);
            TheClient();
        }
    }
}
```

To use the same application as both the server and client, just pass in a dummy argument to the execution

```
java WriteServer   # to start a client
java WriteServer 1 # to start a server
```