

به نام خدا



برنامه‌سازی پیشرفته

دانشگاه شهید بهشتی . دانشکده مهندسی و علوم کامپیوتر

دکتر مجتبی وحیدی اصل

برنامه‌نویسی شبکه - بخش اول

قائم علی‌آبادی

فهرست مطالب

- 1. آدرس IP (IP Address)
- 2. پروتکل (Protocol)
- 3. شماره پورت (Port Number)
- 4. آدرس MAC (MAC Address)
- 5. پروتکل‌های اتصال‌گرا و بدون اتصال (Connection-oriented / Connection-less)
- 6. سوکت (Socket)

مقدمه

در این بخش از درس برنامه‌سازی پیشرفته، با مفاهیم پایه‌ای برنامه‌نویسی شبکه آشنا می‌شویم و یاد می‌گیریم که چطور با استفاده از زبان برنامه‌نویسی جاوا، برنامه‌های شبکه‌ای ساده بنویسیم. در پایان این بخش، قادر خواهیم بود که:

- برنامه‌نویسی سوکت TCP انجام بدیم.
 - داده‌های یک صفحه وب را نمایش دهیم.
 - آدرس IP یک دامنه (مثل www.google.com) را به دست آوریم.
 - برنامه‌نویسی سوکت UDP انجام بدیم.
-

شبکه در جاوا (Java Networking)

- شبکه در جاوا مفهومی برای اتصال دو یا چند دستگاه محاسباتی است تا بتوانیم [منابع](#) را به اشتراک بگذاریم.
- برنامهنویسی [سوکت](#) در جاوا امکان تبادل داده بین دستگاههای مختلف را فراهم میکند.
- مزایای Java Networking
 - با اشتراک‌گذاری منابع
 - متمنکزسازی مدیریت نرم‌افزار



آدرس IP

- آدرس IP یک عدد یکتاست که به هر گره (node) در شبکه اختصاص داده می‌شود؛ مثل **192.168.0.1**. برای نمایش، معمولاً از قالب نقطه‌گذاری شده استفاده می‌کنیم (IPv4).
- از چهار بخش (Octet) تشکیل شده که هر کدام بین **0** تا **255** هستند. هر اکتت در واقع یک بایت است و جمع آن‌ها 32 بیت را می‌سازد.
- یک آدرس **منطقی** است و می‌تواند تغییر کند.



پروتکل (Protocol)

پروتکل یعنی «قواعد گفت‌وگو» بین دستگاه‌ها: پیام چه شکلی باشد، از کدام پورت برود و ارتباط کی شروع و تمام شود؛ اگر این قواعد نباشند، دو طرف حتی با داشتن شبکه مشترک هم علاوه‌تر حرف یکدیگر را نمی‌فهمند.

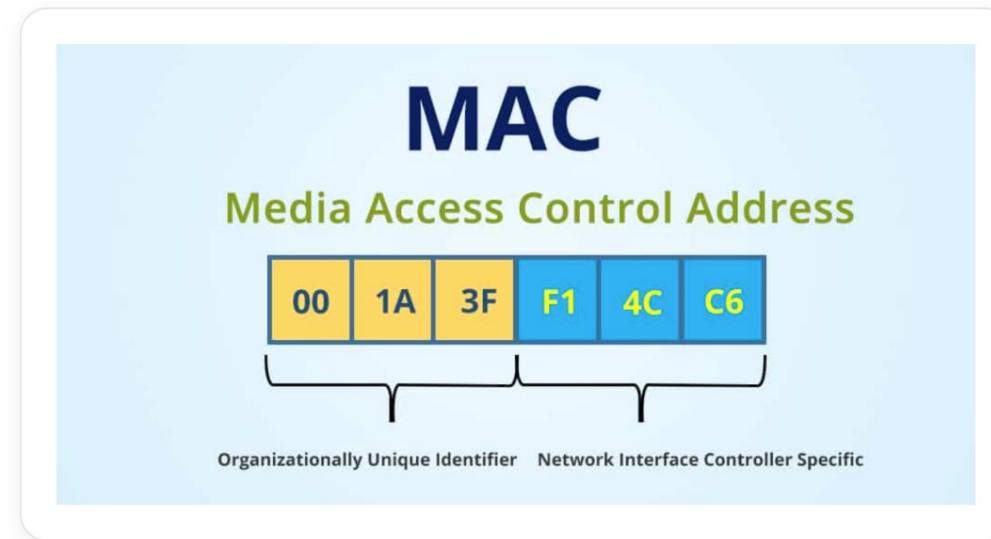
- **TCP**: این پروتکل ابتدا ارتباط را ایجاد و پایدار می‌کند (handshake سه مرحله‌ای) و بعد داده را به قطعات کوچک تقسیم می‌کند؛ برای هر قطعه رسید می‌گیرد، در صورت گم شدن دوباره می‌فرستد و با شماره‌گذاری مطمئن می‌شود همه بسته‌ها دقیقاً به همان ترتیبی که ارسال شده‌اند به مقصد می‌رسند. TCP در کنار **IP** کار می‌کند تا مسیر درست انتخاب شود و به خاطر اطمینان بالا، برای وب و ایمیل و هر جایی که «درست رسیدن» مهم‌تر از «سریع رسیدن» است، انتخاب رایجی به شمار می‌آید.
- در UDP هیچ مذاکره اولیه‌ای برای ساختن ارتباط انجام نمی‌شود و فرستنده بسته‌ها را مستقیم راهی می‌کند؛ خبری از رسید و ارسال مجدد یا تضمین ترتیب نیست و همین سادگی باعث می‌شود سربار بسیار کم و تاخیر پایین باشد. به همین دلیل، برای پخش زنده، تماس صوتی/ تصویری و بازی‌های آنلاین که «به موقع رسیدن» مهم‌تر از «بی‌نقص بودن» است، معمولاً مناسب‌تر از TCP است.
- پروتکلی برای انتقال فایل که امکاناتی مثل فهرست‌کردن، ساخت پوشش، از سرگیری دانلود/آپلود و جابه‌جایی فایل بین کلاینت و سرور را فراهم می‌کند؛ در سناریوهای حساس معمولاً از گونه‌های امن آن مثل **FTP** یا **SFTP** استفاده می‌شود تا هم محتوا و هم نام کاربری/گذرواژه رمزگاری شوند.
- راهی قدیمی برای دسترسی به خط فرمان سیستم‌های راه دور است و اجازه می‌دهد مثل کنسول محلی دستور اجرا کنید؛ اما چون هیچ رمزگاری‌ای انجام نمی‌دهد، شنود اطلاعات بسیار ساده است و به همین خاطر امروز تقریباً همیشه **SSH** به عنوان جایگزین امن توصیه می‌شود.
- ستون فقرات ارسال ایمیل در اینترنت است؛ کلاینت‌ها پیام را از طریق **SMTP** به سرور می‌سپارند و سرورها نیز بین خودشان پیام را جابه‌جا می‌کنند تا به صندوق گیرنده برسد. پورت‌های رایج 25 و 587 و 465 هستند که حالت‌های امن با **TLS/SSL** را هم پوشش می‌دهند.
- پروتکل دریافت ایمیل که معمولاً پیام‌ها را از سرور دانلود می‌کند و روی دستگاه کاربر نگه می‌دارد؛ این روش برای استفاده تک‌دستگاه ساده و مناسب است، اما اگر همگام‌سازی پوشش‌ها روی چند دستگاه را می‌خواهید، **IMAP** تجربه بهتری می‌دهد. نسخه POP3 روی پورت‌های 110 و 995 (نسخه امن) کار می‌کند.

شماره پورت (Port Number)

- شماره پورت برای **تشخیص یکتای برنامه‌ها** روی یک میزبان (Host) استفاده می‌شود! چند برنامه می‌توانند روی یک IP باشند ولی هر کدام پورت خودشان را دارند.
 - پورت در عمل **نقطه انتهایی ارتباط** بین دو برنامه است! داده‌ها به «IP + پورت» مقصد ارسال می‌شوند و به برنامه درست تحویل داده می‌شوند.
 - برای ارتباط، معمولاً ترکیب آدرس IP به همراه **شماره پورت** مبدأ/مقصد مشخص می‌شود؛ مثلا **192.168.1.10:8080**.
-

آدرس MAC

- شناسه یکتای کارت شبکه یا همان **NIC** است! هر رابط شبکه (اترنت/وایفای و ...) یک MAC مخصوص به خودش دارد.
- روی یک گره شبکه ممکن است چند کارت شبکه وجود داشته باشد، اما **هر کارت شبکه MAC** یکتای خودش را دارد.



روش‌های مختلف اتصال

- در پروتکل‌های اتصال‌گرا (مثل TCP) گیرنده برای بسته‌های دریافت شده تأییدیه (ACK) می‌فرستد؛ بنابراین ارتباط قابل اعتماد است اما به خاطر کنترل‌ها و رفت‌وبرگشت‌های بیشتر، معمولاً **کندتر** از حالت بی‌اتصال عمل می‌کند.
- در پروتکل‌های بی‌اتصال (مثل UDP) گیرنده تأییدیه‌ای نمی‌فرستد؛ پس ارتباط **سریع** است اما **تضمين تحويل و ترتیب سریع** ندارد و ممکن است بسته‌ها گم شوند یا جابه‌جا برسند.

بسته **java.net**

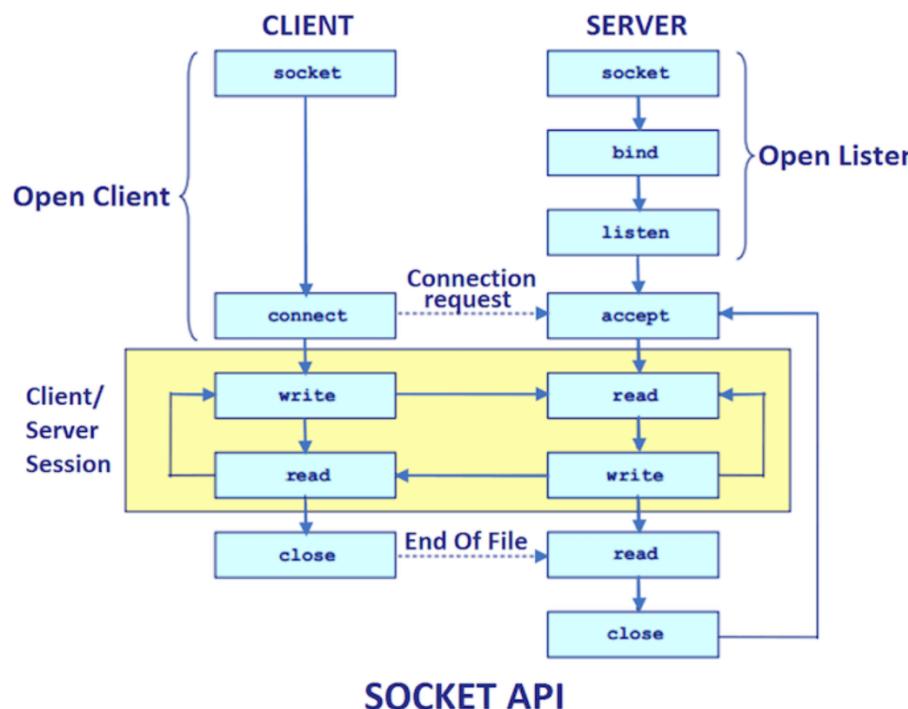
بسته **java.net** مجموعه‌ای از کلاس‌ها را برای ساخت برنامه‌های شبکه در جاوا فراهم می‌کند.

| | | | |
|------------------|-----------------------|------------------------|--------------------|
| URLDecoder | ResponseCache | InetSocketAddress | Authenticator |
| URLEncoder | SecureCacheResponse | InetAddress | CacheRequest |
| URLStreamHandler | ServerSocket | Inet4Address | CacheResponse |
| | Socket | Inet6Address | ContentHandler |
| | SocketAddress | IDN | CookieHandler |
| | SocketImpl | HttpURLConnection | CookieManager |
| | SocketPermission | HttpCookie | DatagramPacket |
| | StandardSocketOptions | NetPermission | DatagramSocket |
| | URI | NetworkInterface | DatagramSocketImpl |
| | URL | PasswordAuthentication | InterfaceAddress |
| | URLClassLoader | Proxy | JarURLConnection |
| | URLConnection | ProxySelector | MulticastSocket |

این فهرست کامل نیست؛ برای جزئیات هر کلاس و بقیه موارد در صورت نیاز به مستندات رسمی جاوا مراجعه کنید.

برنامه‌نویسی سوکت در جاوا

- از **Java Socket programming** برای ارتباط بین برنامه‌های استفاده می‌شود که روی **JRE**‌های متفاوت اجرا می‌شوند.
- این ارتباط می‌تواند **اتصال‌گرا** (Connection-oriented) یا **بی‌اتصال** (Connection-less) باشد.
- برای حالت اتصال‌گرا از کلاس‌های **ServerSocket** و **Socket** و **DatagramSocket** استفاده می‌کنیم؛ و برای حالت بی‌اتصال از **DatagramPacket**.
- کلاینت برای برقراری ارتباط باید **دو اطلاعات** را بداند:
 - آدرس IP سرور
 - شماره پورت



متدها و سازندهای کلاس ServerSocket

- کلاس `java.net.ServerSocket` برای برنامه‌های **سرور** به کار می‌رود تا روی یک **پورت** خاص گوش بدهند و درخواست‌های کلاینت را بپذیرند.
- این کلاس **چهار سازنده** دارد. اگر در سازنده **exception** رخ ندهد، یعنی برنامه با موفقیت به پورت مشخص شده **bind** شده و آماده دریافت اتصال است.

| | | |
|--|--|-----|
| تلاش می‌کند سرور سوکتی بسازد که به پورت داده شده متصل است. اگر آن پورت قبلاً توسط برنامه دیگری اشغال شده باشد، IOException رخ می‌دهد. | <code>public ServerSocket(int port)</code> | (1) |
| مثل سازنده قبلی، با این تفاوت که پارامتر backlog مشخص می‌کند چند اتصال ورودی در حال انتظار در صف نگهداری شوند. | <code>public ServerSocket(int port, int backlog)</code> | (2) |
| به جز تعیین پورت و صف انتظار، با پارامتر InetAddress مشخص می‌کنید سرور روی کدام IP محلی <code>bind</code> شود (برای سیستم‌هایی که چند IP دارند). | <code>public ServerSocket(int port, int backlog, InetAddress address)</code> | (3) |
| یک سرورسوکت غیرمتصل می‌سازد؛ بعدها با متدهای <code>bind()</code> آن را به IP/Port موردنظر متصل کنید. | <code>()public ServerSocket</code> | (4) |

راه‌اندازی ساده Client/Server

- در سناریوی ساده یک‌طرفه، کلاینت وصل می‌شود و پیام می‌فرستد و سرور پیام‌ها را روی اتصال سوکت نمایش می‌دهد. جزئیات سطح پایین را بسته شبکه جاوا ([java.net](#)) مدیریت می‌کند و کار برنامه‌نویس آسان‌تر می‌شود..
- برای اتصال به یک نود دیگر، باید یک **اتصال سوکت** برقرار شود؛ یعنی دو نود از **آدرس IP** و **پورت TCP** یکدیگر اطلاع داشته باشند. کلاس `java.net.Socket` نماینده یک سوکت در سمت کلاینت است.

```
Socket socket = new Socket("127.0.0.1", 6666);
```

- آدرس IP سرور. مقدار **127.0.0.1** همان localhost است و وقتی کلاینت و سرور روی یک دستگاه هستند به کار می‌آید.
- شماره پورت TCP. فقط یک عدد برای مشخص کردن برنامه مقصد؛ مثلا HTTP روی پورت **80** است. بازه پورت‌ها از **0** تا **65535** است.

ایجاد سرور (Server)

- برای ساخت برنامه سرور، باید یک نمونه از کلاس **ServerSocket** بسازیم.
- در این مثال از پورت **6666** برای ارتباط بین کلاینت و سرور استفاده می‌کنیم (می‌توانید هر پورت آزاد دیگری را هم انتخاب کنید).
- متده **accept()** منتظر اتصال کلاینت می‌ماند؛ اگر کلاینت با پورت داده شده وصل شود، یک نمونه **Socket** برمی‌گرداند. این متده مسدود کننده است و تا زمان اتصال کلاینت، اجرای برنامه را نگه می‌دارد.

```
ServerSocket ss = new ServerSocket(6666);
Socket s = ss.accept(); // establishes connection and waits for the client
```

ایجاد کلاینت (Client)

- برای ساخت برنامه کلاینت، باید یک نمونه از کلاس **Socket** بسازیم.
- لازم است **آدرس IP** (یا نام میزبان) سرور و **شماره پورت** را بدانیم. در این مثال چون سرور روی همان سیستم اجرا می‌شود از **localhost** استفاده می‌کنیم.

```
Socket s = new Socket("localhost", 6666);
```

در ادامه یک نمونه ساده از برنامه‌نویسی سوکت را خواهیم داشت که کلاینت متنی را ارسال می‌کند و سرور آن را دریافت کرده و چاپ می‌کند.

ارتباط روی سوکت و بستن اتصال

- برای تبادل داده روی اتصال سوکت از **Stream** استفاده می‌کنیم: `(().getInputStream() و .getOutputStream()` برای کار راحت‌تر از `DataInputStream/DataOutputStream` یا `BufferedReader/PrintWriter` نیز استفاده می‌کنیم.
- در یک برنامه سرور، عملاً با **دو سوکت** سروکار داریم:
 - یک **ServerSocket** که منتظر اتصال کلاینت‌ها می‌ماند (متده `accept()`).
 - یک **Socket** که نتیجه `accept()` است و برای ارسال/دریافت داده با همان کلاینت استفاده می‌شود.
- پس از اتمام کار، جریان‌های ورودی/خروجی و سپس خود **Socket** را بیندید تا منابع آزاد شوند؛ استفاده از `-try`-`with-resources` جلوی هدرفت منابع را خواهد گرفت.

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

public class MyServer {
    public static void main(String[] args) {
        try {
            ServerSocket ss = new ServerSocket(6666);
            Socket s = ss.accept(); // establishes connection

            DataInputStream dis = new DataInputStream(s.getInputStream());
            String str = dis.readUTF();
            System.out.println("message= " + str);

            dis.close();
            s.close();
            ss.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

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

public class MyClient {
    public static void main(String[] args) {
        try {
            Socket s = new Socket("localhost", 6666);

            DataOutputStream dout = new DataOutputStream(s.getOutputStream());
            dout.writeUTF("Hello Server");
            dout.flush();

            dout.close();
            s.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

مثال سوکت در جاوا (خواندن و نوشتن دو طرفه)

- در این مثال، **کلاینت** ابتدا به سرور پیام می‌فرستد و **سرور** آن را دریافت و چاپ می‌کند.
 - سپس **سرور** پاسخی برای کلاینت می‌نویسد و **کلاینت** آن را دریافت و چاپ می‌کند.
 - این روند به صورت رفت‌وبرگشتی ادامه می‌یابد تا زمانی که یکی از طرفین عبارت **stop** را ارسال کند و ارتباط خاتمه یابد.
-

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

class MyServer {
    public static void main(String args[]) throws Exception {
        ServerSocket ss = new ServerSocket(3333);
        Socket s = ss.accept();

        DataInputStream din = new DataInputStream(s.getInputStream());
        DataOutputStream dout = new DataOutputStream(s.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        String str = "", str2 = "";
        while (!str.equals("stop")) {
            str = din.readUTF();
            System.out.println("client says: " + str);
            str2 = br.readLine();
            dout.writeUTF(str2);
            dout.flush();
        }

        din.close();
        s.close();
        ss.close();
    }
}
```

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

class MyClient {
    public static void main(String args[]) throws Exception {
        Socket s = new Socket("localhost", 3333);

        DataInputStream din = new DataInputStream(s.getInputStream());
        DataOutputStream dout = new DataOutputStream(s.getOutputStream());
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        String str = "", str2 = "";
        while (!str.equals("stop")) {
            str = br.readLine();
            dout.writeUTF(str);
            dout.flush();
            str2 = din.readUTF();
            System.out.println("Server says: " + str2);
        }

        dout.close();
        s.close();
    }
}
```

خودمون رو بسنجیم

این بخش برای این طراحی شده که در پایان مطالعه این اسلاید، بتونی خودت رو محک بزنی و ببینی آیا مفاهیم رو به خوبی یاد گرفتی یا نه. سوالات زیر را مرور کن و سعی کن بدون نگاه کردن به متن درس، به اون ها پاسخ بدی.

- آدرس IP چه ساختاری دارد و هر اکت چند بیت است؟
- تفاوت آدرس **IP** و آدرس **MAC** چیست؟
- شماره پورت چه کاربردی دارد؟ یک نمونه **IP:Port** مثال بزن.
- **UDP** و **TCP** از نظر اطمینان، ترتیب و تأخیر چه تفاوتی دارند؟
- برای اتصال سوکت کلاینت به سرور، چه اطلاعاتی لازم داریم؟
- متد `() accept` در **ServerSocket** چه می‌کند و چرا مسدودکننده است؟

پایان

در صورت هرگونه سوال یا پیشنهاد می‌توانی با من
در ارتباط باشی:

Email: me@ghaem.me
Telegram: @Ghaem