شبیه سازی پروکسی سرور Multi-Threaded با NAT/PAT و یک فایل سرور

۱) معرفی پروژه

در این پروژه یک سیستم ساده‌ی انتقال فایل با سه برنامه‌ی جاوا پیاده‌سازی شده است:

- **Client (کلاینت)**: فرمان‌های `LIST` و `DOWNLOAD <filename>` را به پروکسی ارسال می‌کند و پاسخ را دریافت می‌کند.
- **ProxyServer (پروکسی سرور)**: نقش یک پروکسی دارای NAT/PAT را بازی می‌کند؛ برای ارتباط با سرور یک پورت جدید `(Proxy_Port_New)` ایجاد می‌کند، نگاشت را در جدول NAT ذخیره می‌کند، درخواست‌های کلاینت را به سرور ارسال می‌کند و پاسخ‌های سرور را با استفاده از جدول NAT به کلاینت درست برمی‌گرداند.
- **Server (فایل‌سرور)**: روی پوشه‌ی `files/` کار می‌کند و امکان نمایش لیست فایل‌ها و دانلود فایل را فراهم می‌کند.

هدف اصلی پروژه شبیه‌سازی رفتار **NAT/PAT** است.

۲) توپولوژی شبکه و پورت‌ها

- **پروکسی → کلاینت**: `127.0.0.1:4000`
- **سرور → پروکسی**: `127.0.0.1:5000`

پروکسی برای اتصال به سرور از یک پورت محلی آزاد که توسط سیستم‌عامل انتخاب می‌شود استفاده می‌کند، این پورت را در پروژه `Proxy_Port_New` نام‌گذاری می‌کنیم.

در این پروژه هر دو ارتباط زیر از نوع **TCP**هستند:

- **Client ↔ Proxy** روی پورت `4000`
- **Proxy ↔ Server** روی پورت `5000`

بنابراین قبل از ارسال دستورات `LIST` و `DOWNLOAD`، ابتدا بین هر جفت، TCP 3-way Handshake انجام می‌شود.

۳) پروتکل برنامه (فرمان‌ها و پاسخ‌ها)

۳.۱ دستور LIST

درخواست کلاینت:

`LIST\n` (در cmd در نظر گرفته شده که پس از دیدن منو، کاربر با فشردن 1، به LIST دست یابد.)

پاسخ سرور:

سرور نام فایل‌ها را خط به خط می‌فرستد و در پایان `END` را می‌فرستد. برای مثال چیزی که کلاینت می بیند:

```
Files:
file1.txt
file2.pdf
```

اگر هم فایلی وجود نداشته باشد، کلاینت هیچ موردی را نمی بیند و سرور فقط خط `END` را ارسال می‌کند.

```
Files:
no files
```

۳.۲ دستور DOWNLOAD

درخواست کلاینت:

`DOWNLOAD filename\n` (در cmd کاربر با فشردن 2، میتواند به DOWNLOAD دست یابد که در پیام بعدی، اسم فایل مدنظر از کاربر پرسیده می‌شود.)

پاسخ سرور:

اگر فایل وجود نداشته باشد، سرور `-1` را ارسال می‌نماید و برای کاربر پیام زیر نمایش داده می‌شود:

```
File not found.
```

اگر فایل وجود داشته باشد، ابتدا اندازه فایل را می‌فرستد (size) سپس دقیقاً به تعداد <size> بایت، محتوای فایل را (باینری) ارسال می‌کند. سپس فایل در دایرکتوری جاری پس از دانلود قرار می‌گیرد و برای کاربر پیامی شبیه پیام زیر نشان داده می‌شود:

```
File saved to: D:\proxy server\proxy server stimulation\src\hello.txt
```

۴) توضیح کلاس‌ها

۴.۱ Server.java (فایل‌سرور)

```
private static final int PORT = 5000;
```

این پورت، پورت ثابت فایل‌سرور است که طبق پروژه سرور باید روی یک پورت مشخص گوش بدهد (اینجا 5000).

```
File dir = new File("files");
if (!dir.exists() || !dir.isDirectory()) {
    System.out.println("Folder files not found !");
    return;
}
```

سرور باید فایل‌ها را از پوشه‌ی `files` بخواند. اگر پوشه وجود نداشته باشد، ادامه دادن معنی ندارد، پس برنامه متوقف می‌شود.

```
try (ServerSocket serverSocket = new ServerSocket(PORT)) {
    System.out.println("Server listening on port " + PORT);
    while (true) {
        Socket client = serverSocket.accept();
        ...
    }
}
```

`ServerSocket(PORT)` یعنی سرور روی پورت 5000 آماده پذیرش اتصال TCP است. `accept()` یعنی منتظر می‌ماند تا یک اتصال جدید وارد شود.

```
new Thread(() -> {
    try {
        Client(client);
    } catch (IOException e) {
        System.out.println("An error occurred...");
    }
}).start();
```

برای اینکه اگر یک کلاینت/پروکسی در حال دانلود فایل بزرگ باشد، بقیه اتصال‌ها معطل نشوند، برای هر اتصال یک Thread جدا ساخته می‌شود.

```
private static void Client(Socket client) throws IOException {
        try (Socket c = client;
             InputStream in = new BufferedInputStream(c.getInputStream());
             OutputStream out = new BufferedOutputStream(c.getOutputStream())) {
            while (true) {
                String Line = readLineFromStream(in);
                if (Line == null) return;
                ...
            }
        }
    }
```

از روی سوکت، ورودی/خروجی گرفته می‌شود. `while(true)` یعنی سرور تا وقتی اتصال بسته نشده، پشت‌سرهم دستورها را می‌خواند. اگر `readLineFromStream` مقدار `null` بدهد یعنی طرف مقابل قطع شده و این Thread تمام می‌شود.`try-with-resources` باعث می‌شود سوکت و استریم‌ها در پایان `try` خودکار بسته شوند.

```
if (Line.equals("LIST")) {
    LIST(out);
}
else if (Line.startsWith("DOWNLOAD ")) {
    String fileName = Line.substring("DOWNLOAD ".length()).trim();
    DOWNLOAD(out, fileName);
} else {
    out.write("-1\n".getBytes(StandardCharsets.UTF_8));
    out.flush();
}
```

این قسمت پروتکل پروژه را پیاده می‌کند. اگر دستور `LIST` باشد، لیست فایل‌ها را می‌فرستد. اگر دستور با `DOWNLOAD` شروع شود، نام فایل استخراج می‌شود و دانلود انجام می‌گیرد. اگر دستور ناشناخته باشد، `-1` ارسال می‌شود.

`getBytes(StandardCharsets.UTF_8)` یعنی این رشته را به آرایه‌ی بایت‌ها با کُدگذاری UTF-8 تبدیل کن تا بتوانیم آن را داخل `OutputStream` بنویسیم چون `OutputStream` فقط بایت می‌فرستد، نه String.

```
private static void LIST(OutputStream out) throws IOException {
        File dir = new File("files");
        File[] files = dir.listFiles();
        if (files != null) {
            for (File f : files) {
                if (f.isFile()) {
                    out.write((f.getName() + "\n").getBytes(StandardCharsets.UTF_8));
                }
            }
        }
        out.write("END\n".getBytes(StandardCharsets.UTF_8));
        out.flush();
}
```

این متد وظیفه دارد اسم تمام فایل‌های داخل پوشه‌ی `files` را برای کلاینتپروکسی ارسال کند و در پایان با ارسال `END` اعلام کند که لیست تمام شده است. در ابتدا یک شیء `File` ساخته می‌شود که به پوشه‌ای با نام `files` (نسبت به مسیر اجرای برنامه) اشاره می‌کند. سپس محتوای داخل این پوشه (فایل‌ها و فولدرها) به صورت یک آرایه برگردانده می‌شود و اگر به هر دلیلی امکان لیست گرفتن نباشد ممکن است مقدار `null` برگردد. بعد از آن ابتدا بررسی می‌شود که خروجی `null` نباشد تا برنامه خطا ندهد. سپس با حلقه‌ای روی همه‌ی آیتم‌های داخل پوشه پیمایش انجام می‌شود. داخل حلقه، فقط آیتم‌هایی که واقعاً فایل هستند انتخاب می‌شوند و فولدرها یا موارد دیگر ارسال نمی‌شوند. در خط بعد نام فایل با `getName()` گرفته می‌شود، یک

کاراکتر خط جدید \n به انتهای آن اضافه می‌شود تا هر فایل در یک خط جدا ارسال شود، سپس این رشته با getBytes(StandardCharsets.UTF_8) به بایت تبدیل می‌شود چون OutputStream فقط بایت ارسال می‌کند و در نهایت روی خروجی نوشته می‌شود. بعد از اینکه همه‌ی فایل‌ها ارسال شدند، خط END اجرا می‌شود تا علامت پایان لیست فرستاده شود؛ این بخش مهم است چون گیرنده نمی‌داند چند خط قرار است بیاید و با دیدن END متوجه می‌شود لیست کامل شده است. در پایان out.flush(); اجرا می‌شود تا مطمئن شویم تمام داده‌هایی که در بافرجمع شده‌اند فوراً روی شبکه ارسال شوند و گیرنده منتظر نماند.

```
private static void DOWNLOAD(OutputStream out, String fileName) throws IOException {
        File file = new File("files", fileName);
        if (!file.exists() || !file.isFile()) {
            out.write("-1\n".getBytes(StandardCharsets.UTF_8));
            out.flush();
            return;
        }
        long size = file.length();
        out.write((size + "\n").getBytes(StandardCharsets.UTF_8));
        out.flush();
        try (InputStream fileIn = new BufferedInputStream(new
FileInputStream(file))) {
            byte[] buffer = new byte[8192];
            int read;
            while ((read = fileIn.read(buffer)) != -1) {
                out.write(buffer, 0, read);
            }
            out.flush();
        }
    }
```

این متد وظیفه دارد یک فایل مشخص را از داخل پوشه‌ی files پیدا کند و برای گیرنده ارسال کند. ابتدا یک شیء File ساخته می‌شود که مسیر فایل موردنظر را به صورت files/<fileName> می‌سازد. سپس در شرط if بررسی می‌شود که آیا فایل واقعاً وجود دارد و «فایل» است (نه فولدر). اگر وجود نداشته باشد، در خط بعد مقدار -1 به عنوان پیام خطا طبق پروتکل ارسال می‌شود، بعد با out.flush() ; فوراً روی شبکه فرستاده می‌شود و با return متد تمام می‌شود. اگر فایل معتبر بود، در خط بعدی اندازه‌ی فایل بر حسب بایت گرفته می‌شود سپس ابتدا اندازه‌ی فایل به صورت متن (مثلاً 12345) ارسال می‌شود تا طرف مقابل(پروکسی) بداند دقیقاً چند بایت باید دریافت کند. سپس flush زده می‌شود تا اندازه‌ی فایل حتماً قبل از شروع ارسال داده‌ی باینری، ارسال و دریافت شود. بعد یک ورودی برای خواندن فایل از دیسک باز می‌شود (و چون try-with-resources است، در پایان خودکار بسته می‌شود). در خط بعد یک بافر 8KB ساخته می‌شود تا فایل تکه‌تکه خوانده شود (این کار هم سریع‌تر است و هم حافظه را یک‌جا مصرف نمی‌کند و خود عدد 8KB هم عدد معقولی برای این کار است). سپس متغیر تعداد بایت‌های خوانده‌شده تعریف می‌شود. سپس در حلقه‌ی while هر بار تعدادی بایت از فایل داخل buffer خوانده می‌شود و مقدارش در read قرار می‌گیرد؛ وقتی -1 شود یعنی فایل تمام شده است. داخل حلقه در خط out.write(buffer, 0, read) همان تعداد بایتِ خوانده‌شده (از 0 تا read) روی خروجی نوشته می‌شود و به سمت گیرنده ارسال می‌گردد. بعد از اتمام حلقه، flush اجرا می‌شود تا مطمئن شویم آخرین بخش‌های فایل که ممکن است در بافر خروجی مانده باشند نیز ارسال شده‌اند. در نهایت با تمام شدن بلاک try، استریم fileIn به صورت خودکار بسته می‌شود.

```java
private static String readLineFromStream(InputStream in) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        while (true) {
            int b = in.read();
            if (b == -1) {
                if (baos.size() == 0) return null;
                break;
            }
            if (b == '\n') break;
            baos.write(b);
        }
        String s = baos.toString(StandardCharsets.UTF_8);
        if (s.endsWith("\r")) s = s.substring(0, s.length() - 1);
        return s;
 }
```

این متد برای این نوشته شده که از `InputStream` (مثل سوکت شبکه) یک خط متنی بخواند؛ یعنی بایت‌ها را یکی‌یکی می‌خواند و داخل `ByteArrayOutputStream` جمع می‌کند تا وقتی به `\n` برسد (پایان خط). اگر `in.read()` مقدار `-1` بدهد یعنی اتصال/استریم تمام شده؛ اگر هیچ داده‌ای جمع نشده باشد `null` برمی‌گرداند، وگرنه همان چیزی که تا آن لحظه جمع شده را به عنوان خط برمی‌گرداند. بعد از اینکه خط کامل شد، بایت‌های جمع‌شده با UTF-8 به `String` تبدیل می‌شوند. در بعضی سیستم‌ها (ویندوز) پایان خط به صورت `\n\r` است؛ چون ما روی `\n` قطع می‌کنیم ممکن است `\r` در آخر رشته بماند (مثلاً `"LIST\r"` که این مشکل ساز است.)، برای همین اگر رشته با `\r` تمام شود آن را حذف می‌کنیم تا نتیجه تمیز باشد. در نهایت همان خط (بدون `\n`و بدون `\r`) برگردانده می‌شود.

۴.۲. Client.java (کلاینت)

```java
String host = "127.0.0.1";
int port = 4000;
```

این قسمت مشخص می‌کند کلاینت به IP و پورت پروکسی وصل شود (پورت 4000).

```java
try (Socket socket = new Socket(host, port);
     InputStream in = new BufferedInputStream(socket.getInputStream());
     OutputStream outBytes = new BufferedOutputStream(socket.getOutputStream());
     BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in)))
{
```

`Socket(host,port)`یک اتصال TCP به پروکسی می‌سازد. `in` برای دریافت پاسخ‌های پروکسی است.
`outBytes` برای ارسال دستورها به پروکسی است. `keyboard` برای گرفتن ورودی کاربر از کنسول است. چون `try-with-resources` استفاده شده، همه‌ی این منابع در پایان به‌صورت خودکار بسته می‌شوند.

```java
while (true) {
    System.out.println("menu:");
    ...
    String reply = keyboard.readLine().trim();
```

کلاینت در یک حلقه بی‌نهایت منو را نشان می‌دهد و انتخاب کاربر را می‌گیرد.

```java
if (reply.equals("1")) {
                    outBytes.write("LIST\n".getBytes(StandardCharsets.UTF_8));
                    outBytes.flush();
                    LIST(in);
} else if (reply.equals("2")) {
                    System.out.println("Please enter file name:");
                    String fileName = keyboard.readLine().trim();
                    outBytes.write(("DOWNLOAD " + fileName +
                    "\n").getBytes(StandardCharsets.UTF_8));
                    outBytes.flush();
                    DOWNLOAD(in, fileName);
} else {
                    System.out.println("Unknown command, try again");
}
```

اگر دستور کاربر1باشد، `LIST\n` به‌صورت متن (UTF-8) ارسال می‌شود. `flush`باعث می‌شود پیام فوراً ارسال شود و در بافر نماند. سپس متد `LIST(in)` شروع می‌کند پاسخ را از ورودی بخواند و نمایش دهد. اگر دستور کاربر2باشد، نام فایل از کاربر گرفته می‌شود. دستور `DOWNLOAD filename\n` ارسال می‌شود. سپس متد `DOWNLOAD` پاسخ را می‌خواند و فایل را ذخیره می‌کند. اگر کاربر چیز دیگری وارد کند، پیام خطا چاپ می‌شود.

```java
public static void LIST(InputStream in) throws IOException {
        String firstLine = readLineFromStream(in);
        if (firstLine == null || firstLine.trim().isEmpty()) {
            System.out.println("An error occurred...");
            return;
        }
        if (firstLine.equals("-1")) {
            System.out.println("Unknown command.");
            return;
        }
        System.out.println("Files:\n");
        if (firstLine.equals("END")) {
            System.out.println("no files\n");
            return;
        }
        System.out.println(firstLine + "\n");
        while (true) {
            String line = readLineFromStream(in);
            if (line == null) {
                System.out.println("Stream ended early.");
                return;
            }
            if (line.equals("END")) break;
            System.out.println(line + "\n");
        }
    }
```

این متد وظیفه دارد پاسخ دستور LIST را از ورودی شبکه بخواند و لیست فایل‌ها را روی خروجی کنسول چاپ کند. در ابتدا اولین خط پاسخ از سرور/پروکسی خوانده می‌شود. سپس در شرط if بررسی می‌شود که آیا جریان داده قطع شده یا خط خالی آمده است؛ اگر این اتفاق افتاده باشد یعنی مشکلی در ارتباط وجود دارد، پس پیام "An error occurred" چاپ می‌شود و متد تمام می‌شود. بعد در شرط اینکه مقدار اولین خط 1- باشد چک می‌شود که آیا سرور 1- فرستاده است یا نه؛ 1 طبق پروتکل یعنی دستور نامعتبر بوده، بنابراین "Unknown command." چاپ می‌شود و متد برمی‌گردد. سپس در شرط اینکه اولین خط بررسی می‌شود که آیا اولین خط END است یا نه؛ اگر END باشد یعنی هیچ فایلی وجود ندارد و لیست بلافاصله تمام شده است، پس "no files" چاپ می‌شود و متد تمام می‌شود. اگر خط اول نه END بود و نه خطا، در خط بعد، همان اولین نام فایل چاپ می‌شود. سپس وارد حلقه‌ی while می‌شویم تا بقیه خطوط لیست خوانده شوند. داخل حلقه در اولین خط، هر بار یک خط جدید خوانده می‌شود و اگر line == null شود یعنی استریم زودتر از انتظار بسته شده است، پس پیام "Stream ended early" چاپ می‌شود و متد پایان می‌یابد. بعد در شرط ifاگر به END برسیم یعنی لیست تمام شده و از حلقه خارج می‌شویم. در غیر این صورت هر نام فایل جدید چاپ می‌شود و این روند ادامه پیدا می‌کند تا در نهایت END دریافت شود.

```java
    public static void DOWNLOAD(InputStream in, String fileName) throws IOException
{
        String firstLine = readLineFromStream(in);
        if (firstLine == null || firstLine.trim().isEmpty()) {
            System.out.println("An error occurred...");
            return;
        }
        long size;
        try {
            size = Long.parseLong(firstLine.trim());
        } catch (NumberFormatException e) {
            System.out.println("Invalid size received: " + firstLine);
            return;
        }
        if (size < 0) {
            System.out.println("File not found.");
            return;
        }
        File finalFile = new File(System.getProperty("user.dir"), fileName);
        try (OutputStream fileOut = new BufferedOutputStream(new
FileOutputStream(finalFile))) {
            byte[] buffer = new byte[8192];
            long remaining = size;
            while (remaining > 0) {
                int toRead = (int) Math.min(buffer.length, remaining);
                int read = in.read(buffer, 0, toRead);
                if (read == -1) {
                    System.out.println("Stream ended early.");
                    return;
                }
                fileOut.write(buffer, 0, read);
                remaining -= read;
            }
            fileOut.flush();
        }
        System.out.println("File saved to: " + finalFile.getAbsolutePath());
    }
```

این متد وظیفه دارد پاسخ دستور دانلود را از ورودی شبکه بخواند و فایل دریافتی را روی سیستم ذخیره کند. در ابتدا اولین خط پاسخ از سرور/پروکسی خوانده می‌شود که طبق پروتکل باید **اندازه فایل** باشد (یا 1-). سپس در شرط if بررسی می‌شود که آیا ارتباط قطع شده یا خط خالی دریافت شده است؛ اگر چنین باشد پیام خطا چاپ می‌شود و متد تمام می‌شود. بعد متغیر size

تعریف می‌شود تا اندازه فایل داخل آن قرار بگیرد. در بلوک `try` تلاش می‌شود مقدار خط اول به عدد تبدیل شود، چون باید عدد اندازه فایل باشد؛ اگر تبدیل به عدد ممکن نباشد، یعنی پاسخ معتبر نیست، پس در `catch` پیام `"Invalid size received"` چاپ می‌شود و متد برمی‌گردد. سپس بررسی می‌شود که آیا اندازه منفی است یا نه؛ اندازه منفی یعنی فایل وجود ندارد، بنابراین `"File not found"` چاپ می‌شود و متد پایان می‌یابد. اگر اندازه معتبر بود، مسیر ذخیره فایل ساخته می‌شود، یعنی فایل با همان نام داخل مسیر اجرای برنامه ذخیره خواهد شد. بعد در `try` یک خروجی برای نوشتن فایل روی دیسک باز می‌شود و چون try-with-resources است، در پایان خودکار بسته می‌شود. داخل این بلاک، یک بافر 8KB تعریف می‌شود تا داده‌ها تکه‌تکه دریافت شوند و حافظه زیاد مصرف نشود. سپس در `remaining` مقدار باقی‌مانده‌ی بایت‌هایی که باید دریافت شوند (برابر اندازه فایل) قرار می‌گیرد تا دقیقاً به همان تعداد بایت خوانده شود. بعد در حلقه `while` تا زمانی که هنوز بایتی باقی مانده باشد ادامه می‌دهیم. در ابتدا مشخص می‌شود این دور حلقه حداکثر چند بایت باید خوانده شود (یا اندازه بافر، یا مقدار باقی‌مانده اگر کمتر از بافر باشد). سپس در خط بعد داده از شبکه خوانده می‌شود و تعداد بایت واقعی خوانده‌شده در `read` قرار می‌گیرد. اگر `read == -1` شود یعنی استریم زودتر از حد انتظار تمام شده است، پس پیام `"Stream ended early"` چاپ می‌شود و متد خاتمه می‌یابد. اگر داده دریافت شد، در خط بعد باید همان بایت‌ها روی فایل نوشته می‌شوند و سپس مقدار باقی‌مانده کم می‌شود تا در نهایت به صفر برسد. بعد از اتمام حلقه، `flush` اجرا می‌شود تا مطمئن شویم همه‌ی داده‌های نوشته‌شده واقعاً روی دیسک ذخیره شده‌اند. پس از خروج از بلاک `try` فایل به صورت خودکار بسته می‌شود و در نهایت مسیر نهایی ذخیره فایل چاپ می‌شود تا کاربر بداند فایل کجا ذخیره شده است.

۴.۳. ProxyServer.java (پروکسی سرور)

```java
private static final int PROXY_PORT = 4000;
    private static final String FILE_SERVER_HOST = "127.0.0.1";
    private static final int FILE_SERVER_PORT = 5000;
    private static final Map<Integer, NatEntry> natTable = new ConcurrentHashMap<>();

    private static class NatEntry {
        final SocketAddress clientAddr;
        final InputStream clientIn;
        final OutputStream clientOut;
        final InputStream serverIn;
        final OutputStream serverOut;

        NatEntry(Socket clientSocket, Socket serverSocket) throws IOException {
            this.clientAddr = clientSocket.getRemoteSocketAddress();
            this.clientIn  = new BufferedInputStream(clientSocket.getInputStream());
            this.clientOut = new BufferedOutputStream(clientSocket.getOutputStream());
            this.serverIn  = new BufferedInputStream(serverSocket.getInputStream());
            this.serverOut = new BufferedOutputStream(serverSocket.getOutputStream());
        }
    }
```

تنظیمات کلی :

`PROXY_PORT = 4000` همان پورتی که پروکسی روی آن منتظر اتصال کلاینت‌هاست. `FILE_SERVER_HOST = 127.0.0.1` و `FILE_SERVER_PORT = 5000` : آدرس و پورت سرور فایل (Server.java).

`natTable` یک `ConcurrentHashMap` است که نگاشت NAT را نگه می‌دارد:

کلید: `proxyPortNew` (پورتی که سیستم‌عامل برای اتصال Proxy→Server انتخاب می‌کند)
مقدار: یک `NatEntry` که همه چیز لازم برای forward کردن داده را در خودش دارد.

ساختار `NatEntry` :

- `clientAddr` : فقط برای اینکه بدانیم این ارتباط مربوط به کدام کلاینت (IP:Port) بوده.
- `clientIn` و `clientOut` : ورودی/خروجی اتصال **Client↔Proxy**.
- `serverIn` و `serverOut` : ورودی/خروجی اتصال **Proxy↔Server**.

متد `main` (گوش دادن به کلاینت‌ها)

1. پروکسی یک `ServerSocket` روی پورت 4000 می‌سازد.
2. داخل حلقه `while(true)` هر بار با `accept()` یک کلاینت جدید را می‌پذیرد.
3. برای هر کلاینت یک Thread جدا می‌سازد تا همزمان چند کلاینت پشتیبانی شود:
   - هر Thread می‌رود داخل `handleClient(client)` و کار NAT و forwarding را انجام می‌دهد.

متد `handleClient`

این متد برای یک کلاینت اجرا می‌شود.

```java
private static void handleClient(Socket clientSocket) throws IOException {
        Socket serverSocket = new Socket(FILE_SERVER_HOST, FILE_SERVER_PORT);
        int proxyPortNew = serverSocket.getLocalPort();
        NatEntry entry = new NatEntry(clientSocket, serverSocket);
        natTable.put(proxyPortNew, entry);
        try {
            while (true) {
                String request = readLineFromStream(entry.clientIn);
                if (request == null) return;
                request = request.trim();
                if (request.isEmpty()) continue;
                sendLineToServer(proxyPortNew, request);
                forwardServerResponse(proxyPortNew);
            }

        } finally {
            natTable.remove(proxyPortNew);
            try { serverSocket.close(); } catch (IOException ignored) {}
            try { clientSocket.close(); } catch (IOException ignored) {}
        }
    }
```

این متد برای هر کلاینتی که به پروکسی وصل می‌شود اجرا می‌شود و وظیفه‌اش این است که یک اتصال جداگانه از پروکسی به سرور فایل بسازد، نگاشت NAT را ایجاد کند، سپس درخواست‌های کلاینت را به سرور ارسال کند و پاسخ سرور را دوباره به

همان کلاینت برگرداند. در ابتدا پروکسی یک اتصال TCP جدید به سرور فایل روی `127.0.0.1:5000` ایجاد می‌کند تا بتواند درخواست‌های این کلاینت را به سرور بفرستد. در خط بعد شماره پورت محلی این اتصال (سمت پروکسی) گرفته می‌شود که همان `Proxy_Port_New` است و سیستم‌عامل آن را به صورت خودکار انتخاب کرده است. سپس یک NAT entry ساخته می‌شود که داخلش ورودی/خروجی‌های ارتباط کلاینت-پروکسی و پروکسی-سرور قرار می‌گیرد و در نهایت این entry داخل جدول NAT ذخیره می‌شود تا کلید آن `proxyPortNew` باشد و هر وقت لازم شد با این کلید بتوانیم به استریم‌های مربوط به این کلاینت دسترسی داشته باشیم. بعد وارد بخش `try` می‌شود و در حلقه‌ی `while` به طور مداوم درخواست‌های کلاینت را پردازش می‌کند. در داخل حلقه، یک خط دستور از سمت کلاینت خوانده می‌شود (مثل `LIST` یا `DOWNLOAD` `name`). اگر `request == null` شود یعنی کلاینت اتصال را بسته یا قطع شده است، پس متد `return` می‌کند و کار این کلاینت تمام می‌شود. سپس ادامه می‌دهیم و با `request = request.trim()` فاصله‌های ابتدا و انتهای خط حذف می‌شود و اگر بعد از آن درخواست خالی باشد نادیده گرفته می‌شود و حلقه ادامه پیدا می‌کند. بعد از دریافت یک درخواست معتبر، همان دستور از طریق اتصال پروکسی به سرور با پورت جدید توسط `sendLineToServer` برای سرور ارسال می‌شود. سپس با `forwardServerResponse` پاسخ کامل سرور خوانده می‌شود و مطابق پروتکل (LIST یا DOWNLOAD) به کلاینت فوروارد می‌گردد، بنابراین هر درخواست کلاینت دقیقاً یک پاسخ کامل از سرور می‌گیرد. در انتها بخش `finally` تضمین می‌کند که چه خطا رخ دهد چه اتصال قطع شود، منابع پاکسازی شوند؛ ابتدا ورودی مربوطه از جدول NAT حذف می‌شود تا نگاشت قدیمی باقی نماند، سپس اتصال پروکسی به سرور بسته می‌شود و بعد هم اتصال کلاینت به پروکسی بسته می‌شود تا منابع سیستم آزاد شوند.

متد `forwardServerResponse`

```java
private static void forwardServerResponse(int proxyPortNew) throws IOException {
        NatEntry entry = natTable.get(proxyPortNew);
        if (entry == null) return;
        String firstLine = readLineFromStream(entry.serverIn);
        if (firstLine == null) return;
        sendLineToClient(proxyPortNew, firstLine);
        Long sizeMaybe = null;
        try {
            sizeMaybe = Long.parseLong(firstLine.trim());
        } catch (NumberFormatException ignored) {}

        if (sizeMaybe != null) {
            long size = sizeMaybe;
            if (size < 0) {
                return;
            }
            forwardExactBytes(proxyPortNew, entry.serverIn, size);
        }
        else {
            if (firstLine.equals("END") || firstLine.equals("-1")) return;
            while (true) {
                String line = readLineFromStream(entry.serverIn);
                if (line == null) return;
                sendLineToClient(proxyPortNew, line);
                if (line.equals("END")) break;
            }
        }
    }
```

این متد وظیفه دارد پاسخ سرور را بخواند و دقیقاً همان پاسخ را به کلاینت مربوط به همان `proxyPortNew` فوروارد کند. در ابتدای متد با `get`، از جدول NAT داده های منتسب به کلید `proxyPortNew` را می‌گیرد. تا به استریم‌های مربوط به

این ارتباط دسترسی داشته باشیم و اگر `entry == null` باشد یعنی نگاشت NAT وجود ندارد یا حذف شده است، پس متد با `return` متوقف می‌شود. سپس اولین خط پاسخ سرور از استریم `serverIn` خوانده می‌شود و اگر `firstLine == null` باشد یعنی اتصال سمت سرور قطع شده یا استریم تمام شده است، پس متد پایان می‌یابد. بعد همان خط اول پاسخ بلافاصله برای کلاینت ارسال می‌شود تا کلاینت دقیقاً همان چیزی را ببیند که سرور فرستاده است. بعد متغیر `sizeMaybe` تعریف می‌شود و در بلوک `try` تلاش می‌شود خط اول به عدد تبدیل شود؛ دلیلش این است که در پروتکل ما، پاسخ دانلود همیشه با یک عدد شروع می‌شود (یا اندازه فایل، یا `-1`) اما پاسخ لیست فایل‌ها با نام فایل یا `END` شروع می‌شود، پس اگر تبدیل به عدد ممکن نبود وارد `catch` می‌شود و یعنی این پاسخ از نوع `LIST` است. اگر `sizeMaybe != null` شود یعنی خط اول عدد بوده و پس پاسخ از نوع `DOWNLOAD` است؛ در این حالت `size = sizeMaybe` میشود و اگر `size < 0` باشد یعنی مقدار `-1` آمده و فایل وجود ندارد، پس با `return` متد تمام می‌شود و چیزی بیشتر ارسال نمی‌گردد. اگر `size` مثبت باشد، دقیقاً به اندازه `size` بایت از `serverIn` خوانده می‌شود و به کلاینت فوروارد می‌شود تا کلاینت دقیقاً همان فایل را دریافت کند. اما اگر `sizeMaybe == null` باشد یعنی پاسخ از نوع `LIST` است؛ در این حالت ابتدا با بررسی می‌شود که آیا لیست بلافاصله تمام شده (`END`) یا خطا (`-1`) بوده است، که اگر یکی از این‌ها باشد نیاز به خواندن خطوط بیشتر نیست و متد تمام می‌شود. در غیر این صورت وارد حلقه `while` می‌شود و در هر دور یک خط جدید از لیست از سمت سرور خوانده می‌شود و اگر `line == null` شود یعنی اتصال قطع شده و متد پایان می‌یابد. سپس همان خط برای کلاینت ارسال می‌شود و در نهایت اگر به پایان لیست رسیده باشیم باید با `break` از حلقه خارج می‌شویم.

متد `forwardExactBytes`

```java
private static void forwardExactBytes(int proxyPortNew, InputStream serverIn, long
size) throws IOException {
        NatEntry entry = natTable.get(proxyPortNew);
        if (entry == null) return;

        byte[] buffer = new byte[8192];
        long remaining = size;
        while (remaining > 0) {
            int toRead = (int) Math.min(buffer.length, remaining);
            int read = serverIn.read(buffer, 0, toRead);
            if (read == -1) return;
            entry.clientOut.write(buffer, 0, read);
            remaining -= read;
        }
        entry.clientOut.flush();
    }
```

این تابع وقتی سرور گفته است یک فایل با اندازه‌ی مشخص (مثلاً `size` بایت) می‌فرستد، دقیقاً همان مقدار بایت را از ورودی سرور (`serverIn`) می‌خواند و بدون تغییر به خروجی کلاینت (`entry.clientOut`) ارسال می‌کند. ابتدا با `natTable.get(proxyPortNew)` ورودی NAT مربوط به این اتصال پیدا می‌شود تا بدانیم خروجی کلاینت کدام است و اگر پیدا نشود (`entry == null`) تابع تمام می‌شود. بعد یک بافر 8KB ساخته می‌شود تا داده‌ها تکه‌تکه منتقل شوند و حافظه زیاد مصرف نشود. متغیر `remaining` برابر `size` قرار می‌گیرد تا مشخص شود چند بایت هنوز باید منتقل شود. سپس تا زمانی که `remaining > 0` است، هر بار به اندازه‌ی کوچک‌ترِ بین طول بافر و مقدار باقی‌مانده (`toRead`) از `serverIn` خوانده می‌شود؛ اگر `read == -1` شود یعنی استریم قبل از کامل شدن فایل قطع شده و تابع متوقف می‌شود. در غیر این صورت همان تعداد بایت خوانده‌شده برای کلاینت نوشته می‌شود و `remaining` کم می‌شود تا در نهایت صفر شود. در پایان هم `flush` انجام می‌شود تا مطمئن شویم آخرین بایت‌ها هم واقعاً روی شبکه به سمت کلاینت ارسال شده‌اند.

متد sendLineToServer و sendLineToClient

```java
private static void sendLineToServer(int proxyPortNew, String line) throws IOException {
        NatEntry entry = natTable.get(proxyPortNew);
        if (entry == null) return;

        if (!line.endsWith("\n")) line = line + "\n";
        entry.serverOut.write(line.getBytes(StandardCharsets.UTF_8));
        entry.serverOut.flush();
    }

private static void sendLineToClient(int proxyPortNew, String line) throws IOException {
        NatEntry entry = natTable.get(proxyPortNew);
        if (entry == null) return;

        if (!line.endsWith("\n")) line = line + "\n";
        entry.clientOut.write(line.getBytes(StandardCharsets.UTF_8));
        entry.clientOut.flush();
    }
```

متد `sendLineToServer` اول ورودی مربوط به این ارتباط را از جدول NAT پیدا می‌کند تا به `serverOut` دسترسی داشته باشد؛ اگر چنین ورودی‌ای وجود نداشته باشد (`entry == null`) یعنی این نگاشت NAT حذف شده یا معتبر نیست، پس کاری انجام نمی‌دهد و برمی‌گردد. بعد چک می‌کند اگر متنِ پیام با `\n` تمام نشده باشد، یک `\n` به آخرش اضافه کند تا پیام دقیقاً یک خط کامل باشد (چون پروتکل خط محور است و طرف مقابل با `readLineFromStream` تا `\n` می‌خواند). سپس با `line.getBytes(StandardCharsets.UTF_8)` متن را به بایت‌های UTF-8 تبدیل می‌کند (چون `OutputStream` فقط بایت ارسال می‌کند) و آن را روی اتصال Proxy→Server می‌نویسد. در آخر هم `flush` می‌زند تا مطمئن شود پیام فوراً از بافر خارج شده و روی شبکه به سمت سرور ارسال می‌شود.

متد `sendLineToClient` دقیقاً همین منطق را دارد، با این تفاوت که به جای `serverOut` از `clientOut` استفاده می‌کند؛ یعنی NAT entry را پیدا می‌کند، اگر نبود برمی‌گردد، اگر `\n` در انتهای پیام نبود اضافه می‌کند، پیام را UTF-8 به بایت تبدیل می‌کند و روی خروجی اتصال Proxy→Client می‌نویسد، و در پایان `flush` می‌کند تا پیام سریع به کلاینت برسد.

منطق Multithreading :

در `main` پروکسی روی پورت `4000` همیشه در حالت `accept` منتظر اتصال کلاینت‌هاست. به محض اینکه یک کلاینت وصل می‌شود، برای اینکه پروکسی گیر نکند و بتواند همزمان کلاینت‌های بعدی را هم قبول کند، برای همان کلاینت یک Thread جدا ساخته می‌شود. هر Thread فقط روی همان کلاینت کار می‌کند و متد `handleClient(client)` را اجرا می‌کند. داخل `handleClient` دیگر Thread جدیدی ساخته نمی‌شود؛ همان Thread مخصوص آن کلاینت، کارها را به صورت ترتیبی انجام می‌دهد.

چون ممکن است چند Thread همزمان به جدول NAT دسترسی داشته باشند، از `ConcurrentHashMap` استفاده شده تا عملیات‌های `put/get/remove` روی `natTable` در حالت چندنخی ایمن باشد.
پس یک Thread برای هر کلاینت داریم؛ همین باعث می‌شود چند کلاینت همزمان بتوانند از پروکسی استفاده کنند، بدون اینکه اتصال‌های جدید منتظر تمام شدن کار بقیه بمانند.

۵) نحوه اجرا برنامه

- در کنار فایل `Server.java` یک پوشه به نام `files` بسازید.

- فایل‌هایی که می‌خواهید با دستور `LIST` نمایش داده شوند یا با `DOWNLOAD` دانلود شوند را داخل پوشه `files` قرار دهید.

- ترمینال را در همان فولدری باز کنید که سه فایل اصلی پروژه داخل آن هستند.
- سپس دستور زیر را اجرا کنید:

```
javac Server.java ProxyServer.java Client.java
```

- ابتدا سرور را اجرا کنید:

```
java Server
```

باید پیام مشابه رو به رو را ببینید : Server listening on port 5000

- یک ترمینال جدید باز کنید (سرور را نبندید) و پروکسی را اجرا کنید:

```
java ProxyServer
```

باید پیام مشابه رو به رو را ببینید : Proxy listening on port 4000

- یک ترمینال جدید دیگر باز کنید و کلاینت را اجرا کنید:

```
java Client
```

کلاینت به پروکسی روی پورت 4000 وصل می‌شود و منو را نمایش می‌دهد.
اگر بخواهیم چندتا کلاینت داشته باشیم، تنها کافیست ترمینال های جدید باز کنیم و مجدد فایل Client را در آنها ران کنیم.

- گزینه `1` را بزنید تا دستور `LIST` ارسال شود و لیست فایل‌ها نمایش داده شود.
- گزینه `2` را بزنید و نام یک فایل موجود در پوشه `files` را وارد کنید تا دانلود انجام شود.
- فایل دانلودشده در مسیر اجرای کلاینت (current working directory) ذخیره می‌شود.
- برای متوقف کردن هرکدام از برنامه‌ها (Server / Proxy / Client)، در همان ترمینال مربوطه کلیدهای `Ctrl + C` را فشار دهید.

۶) تحلیل اجرای برنامه با wireshark

در ابتدا که کلاس Server و ProxyServer را ران میکنیم، هیچ فیلدی مشاهده نمیکنیم زیرا تا وقتی هیچ ارتباط TCP و هیچ دیتایی رد و بدل نشه، وایرشارک چیزی برای نشون دادن از ترافیک برنامه‌ی نداره.
حال به محض اینکه Client را ران می کنیم، چنین پکت هایی مشاهده می شوند.

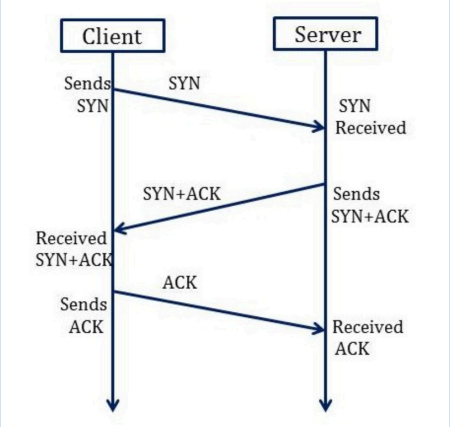
tcp.port == 4000 || tcp.port == 5000

No.	Time	Source	Destination	Protocol	Length	Info
359	269.408166	127.0.0.1	127.0.0.1	TCP	56	58586 → 4000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
360	269.408199	127.0.0.1	127.0.0.1	TCP	56	4000 → 58586 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
361	269.408211	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
362	269.414064	127.0.0.1	127.0.0.1	TCP	56	58587 → 5000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
363	269.414110	127.0.0.1	127.0.0.1	TCP	56	5000 → 58587 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
364	269.414129	127.0.0.1	127.0.0.1	TCP	44	58587 → 5000 [ACK] Seq=1 Ack=1 Win=2619648 Len=0

در کد من، در زمان اتصال کلاینت، دو اتصال TCP برقرار می‌شود.

1. Client ↔ Proxy روی پورت 4000
2. Proxy ↔ Server روی پورت 5000

هر اتصال TCP برای برقرار شدن، ۳ تا پکت handshake دارد (SYN, SYN/ACK, ACK) که این همان Three-way-handshaking است.



اتصال اول: Client ↔ Proxy (پورت 4000)

1. [SYN] 4000 → 58586
   در این مرحله، کلاینت با استفاده از یک پورت موقتی (Ephemeral) مانند 58586 تلاش می‌کند به پروکسی که روی پورت 4000 در حالت Listen قرار دارد اتصال TCP برقرار کند.

2. [SYN, ACK] 58586 → 4000
   پروکسی درخواست اتصال را دریافت کرده و در پاسخ، بسته SYN/ACK را برای پذیرش اتصال و ادامه‌ی فرآیند برقراری ارتباط ارسال می‌کند.

3. [ACK] 4000 → 58586
   کلاینت با ارسال بسته ACK پاسخ پروکسی را تأیید می‌کند. پس از این مرحله، اتصال TCP بین کلاینت و پروکسی برقرار می‌شود.

اتصال دوم: Proxy ↔ Server (پورت 5000)

4. [SYN] 5000 → 58587
این بسته توسط خود پروکسی ارسال می‌شود (نه توسط کلاینت). پروکسی با استفاده از یک پورت موقتی مانند 58587 تلاش می‌کند به سرور فایل که روی پورت 5000 در حال Listen است اتصال TCP برقرار کند. این پورت موقتی همان Proxy_Port_New در سناریوی NAT/PAT محسوب می‌شود و در واقع پورت محلی (Local Port) سوکت پروکسی در ارتباط با سرور است.

5. [SYN, ACK] 58587 → 5000
سرور درخواست اتصال را دریافت کرده و با ارسال بسته SYN/ACK، پذیرش اتصال را اعلام می‌کند.

6. [ACK] 5000 → 58587
پروکسی با ارسال بسته ACK پاسخ سرور را تأیید می‌کند. پس از این مرحله، اتصال TCP بین پروکسی و سرور نیز برقرار می‌شود.

بررسی پکت ها زمانی که no files به کلاینت برگردانده شود:

3749	1255.811702	127.0.0.1	127.0.0.1	TCP	49 58586 → 4000 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=5
3750	1255.811824	127.0.0.1	127.0.0.1	TCP	44 4000 → 58586 [ACK] Seq=1 Ack=6 Win=2619648 Len=0
3751	1255.814224	127.0.0.1	127.0.0.1	TCP	49 58587 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=5
3752	1255.814326	127.0.0.1	127.0.0.1	TCP	44 5000 → 58587 [ACK] Seq=1 Ack=6 Win=2619648 Len=0
3753	1255.815673	127.0.0.1	127.0.0.1	TCP	48 5000 → 58587 [PSH, ACK] Seq=1 Ack=6 Win=2619648 Len=4
3754	1255.815715	127.0.0.1	127.0.0.1	TCP	44 58587 → 5000 [ACK] Seq=6 Ack=5 Win=2619648 Len=0
3755	1255.816092	127.0.0.1	127.0.0.1	TCP	48 4000 → 58586 [PSH, ACK] Seq=1 Ack=6 Win=2619648 Len=4
3756	1255.816135	127.0.0.1	127.0.0.1	TCP	44 58586 → 4000 [ACK] Seq=6 Ack=5 Win=2619648 Len=0

3749. 4000 → 58586
کلاینت به پروکسی داده فرستاده. Len=5 یعنی ۵ بایت دادهٔ برنامه ارسال شده که همان تعداد بایت های LIST\n است. PSH یعنی "این داده را سریع به برنامه تحویل بده"، ACK هم تأیید بایت‌های قبلی است.

3750. 58586 → 4000
پروکسی فقط تأیید می‌کند که دادهٔ کلاینت را گرفت. Ack=6 یعنی تا بایت شماره ۶ را دریافت کرده (یعنی 5 بایت «LIST\n» + شمارش از 1).

3751. 5000 → 58587
این بار خودِ پروکسی همان درخواست را از طرف خودش به سرور می‌فرستد. یعنی NAT/PAT همین‌جا دیده می‌شود: سرور درخواست را از پورت 58587 (Proxy_Port_New) می‌بیند، نه از پورت کلاینت.

3752. 58587 → 5000
سرور تأیید می‌کند که «LIST\n» را دریافت کرده.

3753. 58587 → 5000
سرور پاسخ LIST را می‌فرستد. چون در سناریوی ما "no files" بوده، سرور فقط «END\n» می‌فرستد که ۴ بایت است . پس این بسته دقیقا همان پاسخ "END\n" است.

3754. 5000 → 58587
پروکسی تأیید می‌کند که پاسخ سرور را گرفت (ACK برای «END\n»).از اینجا به بعد ACK=5 چون END به همراه \n دارای4 بیت است و در ACK شمارش از 1 است(5=4+1).

3755. 58586 → 4000
پروکسی همان پاسخ را به کلاینت فوروارد می‌کند: دوباره «END\n» با طول ۴ بایت.

3756. 4000 → 58586
کلاینت تأیید می‌کند که جواب پروکسی را دریافت کرده.

بررسی پکت ها زمانی که 3 به کلاین فایل برگردانده شود:

7938	2141.729849	127.0.0.1	127.0.0.1	TCP	49	58586 → 4000 [PSH, ACK] Seq=6 Ack=5 Win=2619648 Len=5
7939	2141.729900	127.0.0.1	127.0.0.1	TCP	44	4000 → 58586 [ACK] Seq=5 Ack=11 Win=2619648 Len=0
7940	2141.730256	127.0.0.1	127.0.0.1	TCP	49	58587 → 5000 [PSH, ACK] Seq=6 Ack=5 Win=2619648 Len=5
7941	2141.730308	127.0.0.1	127.0.0.1	TCP	44	5000 → 58587 [ACK] Seq=5 Ack=11 Win=2619648 Len=0
7942	2141.732599	127.0.0.1	127.0.0.1	TCP	93	5000 → 58587 [PSH, ACK] Seq=5 Ack=11 Win=2619648 Len=49
7943	2141.732643	127.0.0.1	127.0.0.1	TCP	44	58587 → 5000 [ACK] Seq=11 Ack=54 Win=2619648 Len=0
7944	2141.732989	127.0.0.1	127.0.0.1	TCP	51	4000 → 58586 [PSH, ACK] Seq=5 Ack=11 Win=2619648 Len=7
7945	2141.733024	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [ACK] Seq=11 Ack=12 Win=2619648 Len=0
7946	2141.733330	127.0.0.1	127.0.0.1	TCP	54	4000 → 58586 [PSH, ACK] Seq=12 Ack=11 Win=2619648 Len=10
7947	2141.733353	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [ACK] Seq=11 Ack=22 Win=2619648 Len=0
7948	2141.733488	127.0.0.1	127.0.0.1	TCP	72	4000 → 58586 [PSH, ACK] Seq=22 Ack=11 Win=2619648 Len=28
7949	2141.733501	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [ACK] Seq=11 Ack=50 Win=2619648 Len=0
7950	2141.733582	127.0.0.1	127.0.0.1	TCP	48	4000 → 58586 [PSH, ACK] Seq=50 Ack=11 Win=2619648 Len=4
7951	2141.733593	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [ACK] Seq=11 Ack=54 Win=2619648 Len=0

برای تست برنامه، 3 فایل در پوشه files قرار دادم و کاربر بعد از فشردن 1، نتیجه زیر را می‌بیند:

```
Files:

code.c

hello.txt

Network-MidTerm-Project.pdf
```

No. 7938
58586 → 4000
کلاینت روی اتصال Client↔Proxy داده‌ی برنامه را می‌فرستد (Len=5 یعنی ۵ بایت داده، همان `LIST\n`). فلگ PSH یعنی داده سریع به برنامه تحویل شود. ACK هم یعنی تا این لحظه بایت‌های سمت مقابل را تایید کرده است.

No. 7939
4000 → 58586
پروکسی فقط دریافت داده‌ی ۵ بایتی کلاینت را تایید می‌کند. چون کلاینت قبلاً Seq=6 داشت و ۵ بایت فرستاده، پس Ack باید 11 شود (5+6). این بسته داده‌ی برنامه ندارد (Len=0).

No. 7940
58587 → 5000
این بسته مربوط به اتصال Proxy↔Server است. پروکسی همان درخواست `LIST\n` را به سرور می‌فرستد. نکته NAT/PAT همین‌جاست: سرور درخواست را از پورت موقتی پروکسی (Proxy_Port_New=58587) می‌بیند، نه از پورت کلاینت.

No. 7941
5000 → 58587
سرور دریافت ۵ بایت درخواست `LIST\n` را تایید می‌کند (Ack=11 یعنی ۵ بایت از Seq=6 تا 10 را گرفته است). داده‌ای ارسال نمی‌شود.

No. 7942
5000 → 58587
پاسخ سرور روی اتصال Proxy↔Server ارسال می‌شود. Len=49 یعنی 49 بایت داده‌ی برنامه (لیست فایل‌ها به صورت چند خط + ACK).END\n هم یعنی درخواست کلاینت را همچنان تا Ack=11 تایید دارد. این 49 بایت جمع کاراکتر های اسم هر فایل به علاوه \n بعد از هر کدام و در نهایت END\n است.

No. 7943
58587 → 5000
پروکسی دریافت پاسخ ۴۹ بایتی سرور را تایید می‌کند. چون سرور این پاسخ را از Seq=5 فرستاده و ۴۹ بایت داده داده، پس Ack می‌شود 54 (5+49). داده‌ای در این بسته نیست.

No. 7944
4000 → 58586
پروکسی now پاسخ را روی اتصال Client↔Proxy به کلاینت فوروارد می‌کند. اینجا Len=7 یعنی فقط ۷ بایت از پاسخ برنامه در این بسته آمده (مثلاً ممکن است بخش اول پاسخ مثل یک خط کوتاه یا تکه‌ای از متن باشد). PSH یعنی همین تکه سریع به برنامه کلاینت برسد.

No. 7945
58586 → 4000
کلاینت دریافت ۷ بایت داده از پروکسی را تایید می‌کند. چون داده‌ی ارسالی پروکسی از Seq=5 بوده و ۷ بایت فرستاده، Ack می‌شود 12 (5+7).

No. 7946
4000 → 58586
پروکسی بخش بعدی پاسخ را به کلاینت می‌فرستد. Len=10 یعنی ۱۰ بایت دیگر از پاسخ برنامه را ارسال میکند که این همان اسم فایل دوم است. Seq=12 یعنی ادامه‌ی جریان قبلی است.

No. 7947
58586 → 4000
کلاینت دریافت همین ۱۰ بایت جدید را تایید می‌کند. چون پروکسی از Seq=12، ده بایت فرستاده، Ack می‌شود 22 (10+12).

No. 7948
4000 → 58586
پروکسی یک قطعه‌ی بزرگ‌تر از پاسخ را به کلاینت می‌فرستد. Len=28 یعنی ۲۸ بایت دیگر از همان پاسخ LIST که نام فایل 3 ام است را ارسال می کند.

No. 7949
58586 → 4000
کلاینت دریافت ۲۸ بایت اخیر را تایید می‌کند. چون داده از Seq=22 بوده و ۲۸ بایت آمده، Ack می‌شود 50 (22+28).

No. 7950
4000 → 58586
پروکسی آخرین تکه‌ی پاسخ را می‌فرستد. Len=4 یعنی ۴ بایت پایانی؛ این دقیقاً می‌تواند END\n باشد (۴ کاراکتر).

No. 7951
58586 → 4000
کلاینت دریافت ۴ بایت آخر را تایید می‌کند. چون پروکسی از Seq=50 چهار بایت فرستاده، Ack می‌شود 54 (50+4). این یعنی کل پاسخ LIST کامل به کلاینت تحویل داده شده است.

بررسی پکت ها زمانی که کلاینت برنامه را متوقف کند:

12935	3436.059954	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [FIN, ACK] Seq=11 Ack=54 Win=2619648 Len=0
12936	3436.059988	127.0.0.1	127.0.0.1	TCP	44	4000 → 58586 [ACK] Seq=54 Ack=12 Win=2619648 Len=0
12937	3436.062679	127.0.0.1	127.0.0.1	TCP	44	58587 → 5000 [FIN, ACK] Seq=11 Ack=54 Win=2619648 Len=0
12938	3436.062765	127.0.0.1	127.0.0.1	TCP	44	5000 → 58587 [ACK] Seq=54 Ack=12 Win=2619648 Len=0
12939	3436.063249	127.0.0.1	127.0.0.1	TCP	44	4000 → 58586 [FIN, ACK] Seq=54 Ack=12 Win=2619648 Len=0
12940	3436.063295	127.0.0.1	127.0.0.1	TCP	44	58586 → 4000 [ACK] Seq=12 Ack=55 Win=2619648 Len=0
12941	3436.067307	127.0.0.1	127.0.0.1	TCP	44	5000 → 58587 [FIN, ACK] Seq=54 Ack=12 Win=2619648 Len=0
12942	3436.067359	127.0.0.1	127.0.0.1	TCP	44	58587 → 5000 [ACK] Seq=12 Ack=55 Win=2619648 Len=0

No. 12935
58586 → 4000
کلاینت با ارسال FIN اعلام می‌کند که دیگر داده‌ای روی اتصال Client↔Proxy ارسال نمی‌کند و می‌خواهد اتصال را ببندد. همزمان ACK=54 یعنی تا این لحظه تمام داده‌های دریافتی از سمت پروکسی را هم تایید کرده است.

No. 12936
4000 → 58586
پروکسی دریافت FIN کلاینت را تایید می‌کند. Ack=12 یعنی FIN کلاینت را هم به عنوان یک واحد در توالی حساب کرده و دریافت آن را پذیرفته است. (در TCP، FIN یک شماره توالی مصرف می‌کند.)

No. 12937
58587 → 5000
پروکسی حالا اتصال Proxy↔Server را هم می‌بندد و FIN می‌فرستد. این یعنی پروکسی به سرور اعلام می‌کند دیگر داده‌ای ارسال نخواهد کرد. این بخش نشان می‌دهد بعد از قطع کلاینت، پروکسی ارتباط خودش با سرور را هم می‌بندد.

No. 12938
5000 → 58587
سرور دریافت FIN پروکسی را تایید می‌کند. Ack=12 یعنی FIN پروکسی را هم دریافت کرده و می‌پذیرد که سمت پروکسی دیگر ارسال داده ندارد.

No. 12939
4000 → 58586
پروکسی هم سمت خودش روی اتصال Client↔Proxy را می‌بندد و FIN می‌فرستد. یعنی بعد از اینکه FIN کلاینت را تایید کرد، خودش هم "نیمه‌ی ارسال" اتصال را می‌بندد تا خاتمه اتصال کامل شود.

No. 12940
58586 → 4000
کلاینت دریافت FIN پروکسی را تایید می‌کند. Ack=55 یعنی FIN پروکسی هم یک شماره توالی مصرف کرده (Seq=54 → Ack=55). با این ACK، قطع اتصال سمت Client↔Proxy کامل می‌شود.

No. 12941
5000 → 58587
حالا سرور هم سمت خودش روی اتصال Proxy↔Server را می‌بندد و FIN می‌فرستد. یعنی سرور هم "نیمه‌ی ارسال" خودش را می‌بندد تا پایان اتصال کامل شود.

No. 12942
58587 → 5000
پروکسی دریافت FIN سرور را تایید می‌کند. Ack=55 یعنی FIN سرور نیز یک شماره توالی مصرف کرده (Seq=54 → Ack=55). با این ACK، قطع اتصال سمت Proxy↔Server هم کامل می‌شود.

بررسی پکت ها زمانی که کلاینت دستور دانلود code.c را بدهد:

برای تست برنامه، یک فایل c ایجاد کردم که محتوای آن به شکل زیر است :

```c
#include <stdio.h>

int main(void) {
    printf("Hello, world!\n");
    return 0;
}
```

کاربر با فشردن 2 به منوی دانلود می رود و در آنجا نام فایل مدنظر پرسیده می‌شود و کاربر میگوید code.c
در جواب او چنین چیزی خواهد دید :

```
File saved to: D:\proxy server\proxy server stimulation\src\code.c
```

حال در رصد پکت ها، شرایط زیر را خواهیم داشت:

20236	5010.782188	127.0.0.1	127.0.0.1	TCP	60	57883 → 4000 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=16
20237	5010.782214	127.0.0.1	127.0.0.1	TCP	44	4000 → 57883 [ACK] Seq=1 Ack=17 Win=2619648 Len=0
20238	5010.782556	127.0.0.1	127.0.0.1	TCP	60	57884 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=2619648 Len=16
20239	5010.782574	127.0.0.1	127.0.0.1	TCP	44	5000 → 57884 [ACK] Seq=1 Ack=17 Win=2619648 Len=0
20240	5010.794301	127.0.0.1	127.0.0.1	TCP	47	5000 → 57884 [PSH, ACK] Seq=1 Ack=17 Win=2619648 Len=3
20241	5010.794326	127.0.0.1	127.0.0.1	TCP	44	57884 → 5000 [ACK] Seq=17 Ack=4 Win=2619648 Len=0
20242	5010.794545	127.0.0.1	127.0.0.1	TCP	47	4000 → 57883 [PSH, ACK] Seq=1 Ack=17 Win=2619648 Len=3
20243	5010.794572	127.0.0.1	127.0.0.1	TCP	44	57883 → 4000 [ACK] Seq=17 Ack=4 Win=2619648 Len=0
20244	5010.794983	127.0.0.1	127.0.0.1	TCP	134	5000 → 57884 [PSH, ACK] Seq=4 Ack=17 Win=2619648 Len=90
20245	5010.794999	127.0.0.1	127.0.0.1	TCP	44	57884 → 5000 [ACK] Seq=17 Ack=94 Win=2619648 Len=0
20246	5010.795407	127.0.0.1	127.0.0.1	TCP	134	4000 → 57883 [PSH, ACK] Seq=4 Ack=17 Win=2619648 Len=90
20247	5010.795424	127.0.0.1	127.0.0.1	TCP	44	57883 → 4000 [ACK] Seq=17 Ack=94 Win=2619648 Len=0

No. 20236
57883 → 4000
کلاینت دستور دانلود را به پروکسی می‌فرستد (Len=16 یعنی ۱۶ بایت دادهٔ برنامه (متن دستور) داخل این سگمنت است که این همان DOWNLOAD code.c\n است).فلگ PSH یعنی داده سریع به برنامهٔ پروکسی تحویل شود. ACK هم یعنی کلاینت هرچه تا الان از پروکسی گرفته را تأیید کرده است.

No. 20237
4000 → 57883
پروکسی دریافت دستور کلاینت را تأیید می‌کند. Ack=17 یعنی ۱۶ بایت دادهٔ دستور + ۱ بایت شروع/شمارهگذاری قبلی تا این نقطه را دریافت کرده است.

No. 20238
57884 → 5000

این همان درخواست دانلود است که پروکسی با اتصال خودش به سرور می‌فرستد. پورت 57884 پورت موقتی پروکسی (Proxy_Port_New) در سمت ارتباط Proxy↔Server است. این دقیقاً همان بخش NAT/PAT است: سرور درخواست را از پورت پروکسی می‌بیند، نه از پورت کلاینت.

No. 20239
5000 → 57884
سرور دریافت دستور دانلود را تأیید می‌کند. یعنی «متن درخواست DOWNLOAD کامل رسید».

No. 20240
5000 → 57884
سرور اولین بخش پاسخ DOWNLOAD را می‌فرستد: اندازهٔ فایل به صورت متن + `\n`. چون فایل `code.c` خیلی کوچک است، اندازه‌اش ۳ بایت می‌شود . بنابراین Len=3 نشان می‌دهد اینجا "خط اول پاسخ" یعنی سایز ارسال شده است.

No. 20241
57884 → 5000
پروکسی دریافت خطِ اندازهٔ فایل را تأیید می‌کند. Ack=4 یعنی ۳ بایت داده (سایز + newline) دریافت شد و حالا منتظر بایت بعدی است.

No. 20242
4000 → 57883
پروکسی همان خطِ اندازهٔ فایل را برای کلاینت فوروارد می‌کند (همان ۳ بایت). این همان مرحله‌ای است که با NAT lookup انجام می‌شود: پاسخ سرور که روی Proxy_Port_New آمده، به کلاینت مربوطه هدایت می‌شود.

No. 20243
57883 → 4000
کلاینت دریافت خطِ اندازهٔ فایل را تأیید می‌کند. حالا کلاینت می‌داند باید دقیقاً به اندازهٔ اعلام‌شده، بایت‌های فایل را بخواند.

No. 20244
5000 → 57884
سرور شروع به ارسال محتوای واقعی فایل می‌کند. Len=90 یعنی ۹۰ بایت از بدنهٔ فایل `code.c` در این سگمنت آمده. این همان متن فایل است که به صورت بایت خام روی TCP منتقل می‌شود.

No. 20245
57884 → 5000
پروکسی دریافت بدنهٔ فایل را تأیید می‌کند. Ack=94 یعنی از دید پروکسی تا بایت شماره ۹۳ (جمعاً ۹۰ بایت بعد از Seq=4) دریافت شده و بایت بعدی مورد انتظار ۹۴ است.

No. 20246
4000 → 57883
پروکسی همان ۹۰ بایت محتوای فایل را برای کلاینت فوروارد می‌کند. یعنی داده‌ای که از سرور روی اتصال Proxy↔Server آمده، روی اتصال Client↔Proxy به کلاینت ارسال می‌شود.

No. 20247
57883 → 4000
کلاینت دریافت محتوای فایل را تأیید می‌کند. با توجه به اینکه سایز فایل کوچک بوده، همین یک سگمنت برای بدنه کافی بوده و دانلود کامل می‌شود (کلاینت همان تعداد بایت را می‌خواند و در فایل ذخیره می‌کند).

بررسی پکت ها زمانی که کلاینت جدیدی را همزمان با قبلی ران می‌کنیم:

25840	6456.635981	127.0.0.1	127.0.0.1	TCP	56 57975 → 4000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
25841	6456.636009	127.0.0.1	127.0.0.1	TCP	56 4000 → 57975 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
25842	6456.636026	127.0.0.1	127.0.0.1	TCP	44 57975 → 4000 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
25843	6456.637325	127.0.0.1	127.0.0.1	TCP	56 57976 → 5000 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
25844	6456.637353	127.0.0.1	127.0.0.1	TCP	56 5000 → 57976 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
25845	6456.637365	127.0.0.1	127.0.0.1	TCP	44 57976 → 5000 [ACK] Seq=1 Ack=1 Win=2619648 Len=0

همانطور که مشاهده می‌شود، دو کانکشن TCP جدید ایجاد می‌شود و این بار Proxy_Port_New مقدار 57976 را دارد.

۷) لینک های مفید

- برای دریافت فایل های پروژه به گیت هاب مراجعه نمایید.
- برای دریافت نرم افزار wireshark به سایت soft98 مراجعه نمایید.
- برای دریافت ادیتور کدها به سایت carbon مراجعه نمایید.