کاراپشن

داستان از کجا شروع شد؟

سالیان سال است که فدراسیون بین‌المللی فوتبال (FIFA) مانند خدایانی ظالم در تمام اتفاقات دنیای فوتبال دخالت می‌کنند. این فرایند در دهه اخیر به طرز فاجعه باری از کنترل خارج شد به طوری که تیمی بدون افتخار و هویت، موفق شد دارای 5 بار قهرمانی لیگ قهرمانان اروپا (Champions League) شود و حتی توپ طلا (ballon d'or) که ملاکی برای نشان دادن فوتبالیست‌های افسانه‌ای بود 8 بار در دهه اخیر به سرقت رسید.

اولین شک و شبهه‌ها بر وجود دستی بالاتر در این بازی‌ها پس از نیمه‌نهایی 2009 لیگ قهرمانان شکل گرفت. بازی‌ای که داور آن حتی سال‌ها پس از گذشت آن بازی مورد تهدید هواداران تیم بازنده قرار داشت و بخاطر خیانتی که به دنیای فوتبال کرد، شب‌ها خواب راحت نداشت.

توضیح تصویر

سال‌ها می‌گذرد و بازیکنان و باشگاه‌های فوتبالی، ناچار با شرایط کنار می‌آیند و این خفت و خواری را به جان می‌خرند که شاید سالی برسد که خدایان فوتبال، آنها را به‌عنوان قهرمان انتخاب کند.

در سال 2024 بازیکنی جوان و تازه نفس به این میدان جنگ وارد می‌شود و از بدو ورود خود در معرض بی‌عدالتی‌ها و نابرابری‌های بی‌شماری قرار می‌گیرد.

اون هنوز جوانی خام است و نمی‌تواند آرام بگیرد و با شرایط کنار بیاید. روزها می‌گذرد و هر هفته شاهد مورد ظلم قرار گرفتن تیمش در زمین می‌شود. روزی از روزها پس از 4 باخت متوالی که همه آنها توسط فیفا برنامه‌ریزی شده بود دیگر تصمیم می‌گیرد که دربرابر این ظلم و بی‌عدالتی قیام کند. حتی در بازی آخرش موفق به زدن 3 گل برای تیمش شد و هتریکی باشکوه به نمایش گذاشت، اما همچنان نتوانست رقیب script فوتبال شود.

او می‌خواهد به سرور های فیفا نفوذ کرده و اسرار مخفی فوتبال را پیدا کند. برای اینکار لازم است پسورد رئیس فیفا را پیدا کند. او N نفر را مأمور می‌کند که با استفاده از الگوریتم هش SHA-256 رمز عبور را از روی هش آن پیدا کنند. آن‌ها یک لیست از رمزهای احتمالی دارند و می‌خواهند با بالاترین کارایی ممکن آن را جستجو کنند. رمزهای احتمالی در یک فایل با نام `passwords.txt` ذخیره شده است و هر رمزی که مقدار هش آن با هش داده شده برابر باشد، رمز صحیح می‌باشد.

ذخیره رمزهای کاربران به صورت Plain text در پایگاه‌های داده بسیار ناامن است. به همین منظور، از عملیاتی به نام Hashing(هش کردن) برای احراز هویت کاربران استفاده می‌کنیم.

هش کردن (Hashing) فرایندی است که طی آن یک آرایه به طول دلخواه را به یک آرایه با طول ثابت تبدیل می‌شود. این رشته خروجی، هش نامیده می‌شود. از تابع هش برای مقاصد مختلفی مانند رمزنگاری استفاده می‌شود. این فرایند باید برگشت‌ناپذیر باشد تا از روی خروجی آن نتوان به رشته اصلی دست پیدا کرد.

در سامانه‌های احراز هویت، نام کاربری در کنار هش رمز عبور (به جای خود رمز عبور) ذخیره می‌شود، به این شکل هنگامی که سیستم بخواهد رمز کاربر را تایید کند، هش آن را محاسبه می‌کند و با هش‌ای که هنگام ثبت نام کاربر محاسبه کرده بود، مقایسه می‌کند. با انجام این فرایند حتی در صورت پخش شدن اطلاعات پایگاه داده، رمزها به صورت مستقیم فاش نخواهند شد.

الگوریتم‌های هش کردن زیادی وجود دارد که در این سوال به الگوریتم SHA-2 (به طور خاص تر، SHA-256) می‌پردازیم. برای آشنایی بیشتر با توابع هش رمزنگاری، میتوانید به این لینک و با خانواده SHA-2 به این لینک مراجعه کنید. در صورت علاقه مندی بیشتر، در اینجا می‌توانید با نحوه کار الگوریتم SHA-2 بیشتر آشنا شوید.

برای استفاده از تابع هش SHA-256، میتوانید از کلاس زیر در جاوا استفاده کنید:

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.nio.charset.StandardCharsets;

public class CryptoHash {
    public static String hashString(String text) {
        try {
```

```
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] encodedhash = digest.digest(text.getBytes(StandardChar
                StringBuilder hexString = new StringBuilder(2 * encodedha
            for (byte b : encodedhash) {
                String hex = String.format("%02x", b);
                hexString.append(hex);
            }

            return hexString.toString();

        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

ورودی

در خط اول ورودی به شما یک هش ۳۲ بایتی با نمایش Hexadecimal داده می شود. همچنین در خط دوم ورودی به شما عدد N داده می‌شود. همچنین فایل `passwords.txt` را در انتهای سوال می‌توانید دریافت کنید. تضمین می‌شود هش داده شده از N رمز عبور آخر نیست.

▼ *برای آشنایی بیشتر*

برای تمرین، می‌توانید تمام رمزهای فایل را کنار یکدیگر **بدون وایت اسپیس** گذاشته و هش آن را بررسی کنید که با مقدار زیر یکی شده است! (طبیعتا این بخش الزامی نیست و نمره‌ای ندارد و هدف آن صرفاً آشنایی بیشتر دانشجو با عملیات هش‌کردن است).

```
609355ff03275dd9d9c1144b585b03b25526d188a16d85d83154df93d2f0e049
```

نحوه اجرا

با توجه به ماهیت سوال، **باید تمام شرایط زیر رعایت شود** تا جواب یکتا به دست بیاید.

- به تعداد N ترد ثابت داریم که هم‌زمان اجرا می‌شوند.
- در هر سری، باید **کار هر N ترد تمام شود** تا سراغ N ترد بعدی برویم.
- لیست رمزهای عبور به صورت صف (queue) است. (یعنی به ترتیب از اول لیست رمزهای عبور تست شوند)
- هر ترد پس از تمام شدن بررسی یک رمز عبور، به‌صورت خودکار رمز عبور بعدی را از صف می‌گیرد.
- عملیات تا زمانی ادامه دارد که یا رمز عبور صحیح پیدا شود، یا همه رمزهای عبور بررسی شده باشند.

پیشنهاد : در مورد Executors و AtomicReference تحقیق کنید.

تعریف ویژگی خاص

در طول بررسی رمزهای عبور، تردها هش SHA-256 هر رمز را محاسبه می‌کنند. اگر حداقل یکی از ۳ رقم اول (از سمت چپ) هش در بازه 0 تا 6 باشد، آن پسورد ویژگی خاص دارد. تعداد کل پسوردهای با ویژگی خاص با در نظر گرفتن و حل **race condition** ها باید به ما برگردانده شود.

توابع مورد نیاز در کلاس PasswordBreaker

تابع	توضیح
void setTargetHash(String hash)	هش رمز هدف را تعیین می‌کند.
void loadPasswords(File file)	لیست رمزهای احتمالی را از فایل بارگذاری می‌کند.
void startCracking(int numThreads)	عملیات کرک را با N ترد موازی آغاز می‌کند.
String getFoundPassword	در صورت یافتن رمز صحیح، آن را برمی‌گرداند. در غیر این صورت null بر میگرداند.
int getSpecialHashCount	تعداد پسوردهایی که ویژگی خاص دارند.

خروجی

در ابتدا، تعداد رمزهای عبور با ویژگی خاص باید چاپ شود. سپس، در صورتی که رمز عبور صحیح پیدا نشد، جمله `NOT FOUND` چاپ می شود. در غیر این صورت، رمز عبور صحیح باید چاپ شود. تضمین می‌شود که کاراکتر اسپیس در کلمات موجود در فایل وجود ندارد.

مثال

اگر ورودی زیر به شما داده شود:

```
00f7ef114a768fce3089a2204624984c6f24646a0fd00acc5799854d628aabc1
5
```

خروجی باید به شکل زیر باشد:

```
12
k#iW*ipOletab^Le%@
```

نکته: اگر به صورت Single Thread بررسی را انجام می‌دادیم، هنگامی که به رمز مدنظر می‌رسیم تنها ۱۰ رمز با ویژگی خاص پیدا می‌کنیم اما از آن‌جایی که قرار شد به صورت دسته‌های ۵ تایی رمزها را بررسی کنیم، دو رمز دیگر بعد از رمز شماره ۱۳ هم رمز عبور صحیح خواهند بود که با استفاده از دسته‌های ۵ تایی پیدا می‌شوند.

ساختار پروژه و نحوهٔ داوری

ساختار پروژهٔ شما باید به صورت زیر باشد:

```
Main.java
PasswordBreaker.java
CryptoHash.java
passwords.txt
```

فایل passwords.txt را برای تست خودتان باید در کنار پروژهٔ خود قرار دهید: **دریافت فایل**

توجه: نیازی به قرار دادن به فایل passwords.txt داخل فایل زیپ ارسالی نیست. در سامانهٔ داوری به‌طور خودکار این فایل کنار پروژه قرار می‌گیرد.

پیشنهاد: می‌توانید یک کد python/bash بنویسید که به ازای یک hash ثابت، و تعداد ترد (N) متغیر، نمودار زمان را رسم کرده یا مقادیر زمان را چاپ کند؛ و برآورد کنید که مقدار N بهینه برای ترکیب این مسئله و سیستم شما چه عددی است.

دانجر

پس از اتفاقات اخیر و حملات پهبادی به خانه نقی معمولی، بیش از پیش زیر نظر وزارت گرفته شد. تمام فعالیت‌ها و حرکات او اعم از ردپاهای سایبری‌اش تحت نظر قرار گرفت. او برای اینکه به فعالیت‌های مرموز خود ادامه دهد و ارتباط خود را با دشمن حفظ کند، از خواستگار دخترش (ارشاد) درخواست می‌کند برای او یک سیستم مخفی برای اشتراک‌گذاری اسناد محرمانه‌اش بسازد جوری که ردپای سایبری‌اش به حداقل برسد. ارشاد تصمیم می‌گیرد از **تورنت** استفاده کند. از آنجا که این آخر هفته، ارشاد با سارا مشغول است، از شما خواسته یک سیستم مدیریت تورنت مانند BitTorrent برای نقی پیاده کنید.

بیت تورنت

تورنت، یک پروتکل ارتباطی است که برای اشتراک گذاری فایل به صورت P2P استفاده می‌شود، جهت کسب اطلاعات بیشتر می‌توانید از این لینک استفاده کنید.

ابتدا دو مفهوم اصلی در بیت تورنت را بررسی می‌کنیم و سپس به شرح سوال می‌پردازیم.

همتا (Peer): به هر دستگاه که به این شبکه متصل می‌شود و شروع به دانلود و اشتراک گذاری فایل‌ها می‌کند، همتا گفته می‌شود.

ردیاب (Tracker): ردیاب در بیت‌تورنت یک سرور ویژه است که به ارتباط بین کاربران (یا همان همتاها) کمک می‌کند تا بتوانند فایل‌ها را به صورت همتا به همتا (P2P) به اشتراک بگذارند. ردیاب خودش فایل‌ها را نگه نمی‌دارد، بلکه وظیفه‌اش این است که فهرستی از کاربران فعال در یک تورنت خاص را داشته باشد و به کلاینت‌های بیت‌تورنت کمک کند تا یکدیگر را پیدا کنند.

شما می‌بایست در این سوال یک نسخه بسیار ساده از پروتکل بیت تورنت را پیاده‌سازی کنید، که در ادامه جزئیات آن را ذکر می‌کنیم.

در پروتکل بیت‌تورنت، یک همتا لزوما همه قسمت‌های یک فایل را ندارد و واحدهای ارسالی توسط کاربران، قطعات کوچک‌تری از یک فایل می‌باشند، اما در این تمرین نیازی به رعایت این موضوع نیست و همتاها فایل‌ها را به صورت کامل دارند.

همچنین در پروتکل بیت‌تورنت، لزومی بر حضور دو همتا در یک شبکه وجود ندارد اما در این تمرین پیاده‌سازی برای شرایطی که تمامی همتاها در شبکه داخلی باشند کافی است.

در این تمرین شما می‌بایست دو ماژول را پیاده‌سازی کنید: **ماژول Tracker** (یا همان سرور) و **ماژول Peer** (یا همان کلاینت). **شما در این تمرین لازم نیست این دو ماژول را از صفر پیاده‌سازی کنید؛ بلکه برای راحتی کار شما یک پروژهٔ اولیه آماده شده است و شما صرفاً باید آن را کامل کنید. برای اطلاع از ساختار پروژه و همچنین دریافت فایل پروژهٔ اولیه به انتهای صورت سوال مراجعه کنید.**

ردیاب (Tracker)

روند کار به این صورت است که ابتدا Tracker (که entrypoint آن، فایل `TrackerMain.java` می‌باشد) روی یک پورت مشخص که در `args[0]` به آن داده خواهد شد، روی کامپیوتر سرور به اجرا در می‌آید. برای مثال اگر بخواهیم Tracker را روی پورت ۸۰۸۰ راه‌اندازی کنیم، دستور زیر را می‌زنیم (دستورات دقیق‌تر در انتهای صورت سوال):

```
java tracker.TrackerMain 8080
```

همتا (Peer)

روی کامپیوترهای دیگر، **Peer** راه‌اندازی می‌شود (entrypoint آن، فایل `PeerMain.java` باید باشد)؛ توجه کنید که هر **Peer** نقش دوگانه‌ای در معماری Client/Server دارد؛ یعنی در ارتباط با Tracker، کلاینت محسوب می‌شود، ولی در ارتباط با **Peer**های دیگر می‌تواند نقش Client یا Server داشته باشد (با توجه به اینکه آیا درخواست فایل کرده است یا از آن درخواست فایل شده است). بنابراین، برای اتصال به سرور Tracker باید آی‌پی

و پورت Tracker را بداند، و برای اتصال **Peer** های دیگر به آن و دانلود فایل از آن، یک پورت باید مشخص کند (که روی آن Listen کند). همچنین هنگام راه‌اندازی باید مسیر مخزنِ فایل‌هایی که داریم را مشخص کنیم. در نتیجه هر Peer به‌صورت زیر اجرا می‌شود:

```
java peer.PeerMain <peer-ip>:<peer-port> <tracker-ip>:<tracker-port> <folder>
```

در دستور بالا، `args[0]` شامل یک عبارت مانند `192.168.1.10:8081` است که شامل آی‌پی همین سیستمی که در حال اجرای Peer روی آن هستیم، به‌همراه مشخص کردن یک پورت، جهت اتصال Peer های دیگر است. همچنین `args[1]` شامل یک عبارت مانند `192.168.1.2:8080` است که شامل آی‌پی و پورت Tracker می‌باشد. و `args[2]` نیز مسیر نسبی یا مطلق فولدری است که می‌خواهیم فایل‌های درون آن را با بقیهٔ Peer ها به اشتراک بگذاریم. (دقت کنید که تنها فایل‌های با عمق یک -یعنی فایل‌هایی که داخل subfolder دیگری نیستند- را کار خواهیم داشت).

دستورات بین همتا و ردیاب

همتا در بدو اجرا شدن، با Socket به ردیاب متصل می‌شود. این سوکت، شامل یک ارتباط دو طرفه از دستورات است که در آن هر لحظه ممکن است همتا به ردیاب دستوری ارسال کند یا دستوری از ردیاب دریافت کند. (بدیهی‌است که این دستورات باید در تردی جداگانه از ترد اصلی هر یک از طرفین Handle شوند تا هم امکان اتصال Peer های همزمان وجود داشته باشد و هم این‌که طرفین بتوانند دستورات کاربر در کنسول را Handle کنند چرا که این‌کار در Thread اصلی هرکدام صورت می‌گیرد).

قالب Represenation داده‌ها در این بخش از دستورات، قالب JSON است. دقت کنید که حق استفاده از کتابخانهٔ خارجی (مانند Gson) برای شما به استفاده از فایل `.jar` آن‌ها محدود می‌شود؛ یعنی هر کتابخانه‌ای که از آن استفاده می‌کنید باید فایل `.jar` آن را در پروژهٔ خود طبق توضیحات انتهای صورت سوال قرار دهید. برای مثال برای تبدیل به JSON و برعکس، می‌توانید از کتابخانهٔ `Gson` استفاده کنید که می‌توانید فایل `.jar` آن را از این مسیر دانلود کنید.

دستور `status` - نوع: ردیاب به همتا

(پیگیری وضعیت)

```
{
    "type": "command",
    "body": {
        "command": "status"
    }
}
```

این دستور در بدو اتصال هر همتا به ردیاب، باید از سوی ردیاب به آن همتا ارسال شود (البته در مواقع دیگری نیز ردیاب این دستور را ارسال می‌کند که در بخش‌های بعدی به آن اشاره می‌شود). همتا باید در پاسخ به این دستور، آی‌پی خود به‌همراه پورتی که روی آن Listen می‌کند را برای ردیاب ارسال کند:

```
{
    "type": "response",
    "body": {
        "command": "status",
        "response": "ok",
        "peer": "192.168.1.10",
        "listen_port": 8081
     }
}
```

دستور `get_files_list` - نوع: ردیاب به همتا

(دریافت لیست فایل‌ها)

```
{
    "type": "command",
    "body": {
        "command": "get_files_list"
    }
}
```

هنگامی که این دستور از ردیاب به یک همتا ارسال می شود (که در بخش‌های بعدی توضیح می‌دهیم در چه مواقعی این اتفاق می‌افتد)، همتا باید در جواب لیست فایل‌هایی که در پوشهٔ مشخص شده وجود دارد (که گفتیم صرفاً فایلهای داخل ریشهٔ این پوشه مد نظرمان است و فایل‌های داخل subfolder ها مد نظر نیست) را به همراه MD5 Hash آن‌ها به ردیاب برگرداند؛ تا ردیاب مطلع شود که این همتا چه فایل‌هایی را دارد:

```
{
    "type": "response",
    "body": {
        "command": "get_files_list",
        "response": "ok",
        "files": {
            "<file1>" : "<md5_of_file1>",
            "<file2>" : "<md5_of_file2>",
            ...
        }
    }
}
```

دستور get_sends - نوع: ردیاب به همتا

(دریافت لیست ارسال‌ها)

```
{
    "type": "command",
    "body": {
        "command": "get_sends"
    }
}
```

همتا می‌بایست با دریافت این دستور از سمت ردیاب، لیست فایل‌هایی که برای همتاهای دیگر ارسال کرده است را در پاسخ برگرداند. لازم به ذکر است که نیازی به ذخیره این لیست روی دیسک نیست و با هربار اجرای برنامه این لیست خالی می‌شود. نمونه پاسخ:

```
{
    "type": "response",
    "body": {
        "command": "get_sends",
        "response": "ok",
        "sent_files": {
            "<ip1:port1>": [
            "<file1> <hash1>",
            "<file2> <hash2>",
            ...
            ],
            "<ip2:port2>": [
                "<file1> <hash1>",
                "<file2> <hash2>",
                ...
            ],
            ...
        }
    }
}
```

دستور `get_receives` - نوع: ردیاب به همتا

(دریافت لیست دریافت‌ها)

```
{
    "type": "command",
    "body": {
        "command": "get_receives"
    }
}
```

همتا می‌بایست با دریافت این دستور از سمت ردیاب، لیست فایل‌هایی که از همتاهای دیگر دریافت کرده است را به‌عنوان پاسخ برگرداند. لازم به ذکر است که نیازی به ذخیره این لیست روی دیسک نیست و با هربار اجرای برنامه این لیست خالی می‌شود. نمونه پاسخ همتا:

```
{
    "type": "response",
    "body": {
        "response": "ok",
        "command": "get_receives",
        "received_files": {
            "<sender1-ip:port>" : [
                "breaking_bad.mkv <md5_of_file>",
                ...
            ],
            "<sender2-ip:port>" : [
                "rick_and_morty_season1.zip <md5_of_file>",
                ...
            ],
            ...
        }
    }
}
```

حال سراغ تنها دستوری می‌رویم که همتا می‌تواند برای ردیاب ارسال کند:

دستور `file_request` - نوع: همتا به ردیاب

(درخواست یک فایل به ردیاب برای اطلاع از Peer هایی که آن فایل را دارند.)

```
{
    "type": "file_request",
    "body": {
        "name": "<file_name>"
    }
}
```

ردیاب باید با دریافت این دستور، لیست همتایانی که این فایل را دارند را در نظر بگیرد و یک مورد را به‌صورت تصادفی در پاسخ برگرداند. نمونه پاسخ در حالت عادی:

```
{
    "type": "response",
    "body": {
        "response": "peer_found",
        "md5": "<md5_hash>",
        "peer_have": "<peer-ip>",
        "peer_port": <peer-port>
    }
}
```

همچنین نیازمند دو نوع Error Handling در این بخش هستیم؛ یکی در حالتی که همتایی وجود نداشت که فایل مربوطه را داشته باشد (File not found)، و یکی نیز در حالتی که همتایانی وجود دارند که فایل را دارند اما حداقل دوتا از آن‌ها MD5 Hash متفاوتی دارند. (Hash Conflict) در حالت خطای Not found:

```
{
    "type": "response",
    "body": {
        "response": "error",
        "error": "not_found"
    }
}
```

در حالت خطای Hash Conflict:

```
{
    "type": "response",
    "body": {
        "response": "error",
        "error": "multiple_hash"
    }
}
```

دستورات بین دو همتا

هر همتا می‌بایست در یک ترد جداگانه از ترد اصلی برنامه و نیز تردی که وظیفه پاسخ به دستورات ردیاب را دارد، منتظر پیغام‌های همتایان دیگر باشد. در واقع این همان بخشی است که لازم است روی پورتی که مشخص شده، در نقش Server عمل کند و Listen کند. تنها دستوری که از یک همتا می‌تواند به یک همتای دیگر ارسال شود، دستور **دانلود فایل** است که همتای درخواست کننده، به‌عنوان Client، به همتای دارای فایل (که از طریق ردیاب با او آشنا شده است) متصل می‌شود و درخواست زیر را می‌فرستد:

```
{
    "type": "download_request",
    "body": {
        "name": "<file_name>",
        "md5": "<md5_of_file>",
        "receiver_ip": "<my-ip>",
        "receiver_port": <my-port>
    }
}
```

از این لحظه هرگونهٔ دادهٔ دریافتی از سمت همتای دارای فایل، به‌عنوان دادهٔ فایل تلقی می‌شود. همتای دارای فایل (همتای در نقش سرور)، هنگامی که تمام داده‌های فایل را ارسال کرد، سوکت را می‌بندد. یعنی بستن سوکت از سمت همتای سرور، به منزلهٔ EOF تلقی می‌شود. همچنین، همتای سرور، در صورتی که درخواست اشتباه آمده باشد (یعنی واقعاً فایل را ندارد و همتای درخواست کننده بدون توجه به گزارش ردیاب، به او درخواست زده است)، درجا سوکت را می‌بندد. یعنی انگار یک فایل با حجم ۰ بایت ارسال می‌کند.

همتای دریافت کننده با دریافت فایل و قرار دادن آن در پوشهٔ مخصوص به خود (همان که در `args[2]` داده شده بود)، وظیفه دارد MD5 Hash محتوای دریافت شده را محاسبه کند و با چیزی که در درخواست خود زده بود، مطابقت دهد. در صورت مطابقت، فایل داخل پوشه حفظ می‌شود. در غیر این‌صورت، با File Corruption روبرو شده‌ایم و فایل باید حذف شود. (یا کلاً ایجاد نشود).

دستورات کنسول سمت همتا

پردازش دستورات کنسول می‌تواند در ترد اصلی برنامه انجام شود.

دانلود فایل - `download`

کاربر می‌تواند با وارد کردن دستور زیر، یک فایل را دانلود کند.

```
download rick_and_morty_season1.zip
```

روند کار: دستور بالا، باعث می‌شود که دستور `file_request` به ردیاب ارسال شود و ردیاب پاسخ دهد. در صورتی که به دو خطای File Not Found و یا Hash Conflict نخوریم، باید به همتایی که ردیاب معرفی کرده است `download_request` بفرستیم. سپس طبق توضیحات فوق فایل را دریافت می‌کنیم؛ و طبق توضیحات، بعد از دریافت کامل فایل، MD5 آن را محاسبه کرده و مطابقت می‌دهیم و در صورتی که به خطای File Corruption نخوریم، عملیات دانلود با موفقیت انجام شده است و پیغام زیر در خروجی چاپ می‌شود:

```
File downloaded successfully: rick_and_morty_season1.zip
```

اگر فایل از قبل در مخزن خودمان موجود بود:

```
You already have the file!
```

اگر با خطای File not found روبرو شدیم:

```
No peer has the file!
```

اگر با خطای Hash Conflict روبرو شدیم:

```
Multiple hashes found!
```

و اگر دچار File Corruption شده بودیم:

```
The file has been downloaded from peer but is corrupted!
```

لیست فایل‌ها - `list`

با زدن این دستور در کنسول، باید لیست فایل‌های داخل پوشهٔ مخزن به همراه md5 hash آن‌ها نمایش داده شود:

```
list
```

نمونهٔ خروجی (لیست خروجی بر اساس اسم فایل و به ترتیب حروف الفبا مرتب‌سازی شود) :

```
aa.zip 54062885c8a7faf3ccc54ba35fb6c801
salam.txt 826e8142e6baabe8af779f5f490cf5f5
```

اگر هیچ فایلی وجود نداشت:

```
Repository is empty.
```

دستور `exit` :

```
exit
```

با زدن این دستور، تمام روند کاری سمت همتا متوقف می‌شود (ترد Listen کننده، ترد اتصال با ردیاب، و تردهای ارسال فایل در صورت وجود) و از برنامه خارج می‌شویم. دقت کنید که حتماً سوکت اتصال با ردیاب باید close شود.

دستورات کنسول و روند کاری سمت ردیاب

در سمت ردیاب، پذیرش اتصالات همتایان و پردازش دستورات مربوط به `file_request` و ... از سمت همتایان، در تردهای موازی جداگانه باید صورت گیرد؛ یعنی یک ترد به‌ازای هر همتا در حال پردازش است. البته در واقعیت به دلایل مختلفی اعم از تعداد زیاد همتا و موارد امنیتی از این حالت استفاده نمی‌شود و از سیستمی موسوم به Thread Pool استفاده می‌گردد ولی در این تمرین برای ساده‌سازی، شما کافی است به ازای هر Peer متصل، یک

ترد جداگانه ایجاد کنید. واضح است که با توجه به اینکه می‌خواهیم پردازش دستورات کنسول در ترد اصلی برنامه انجام شود، باید یک ترد جداگانه نیز برای Accept سوکت سیستم‌های همتا داشته باشیم.

لیست همتایان و فایل‌ها

سرور می‌بایست در یک لیست، همتایانِ فعال (که در حال حاضر روشن هستند و ارتباط با آن‌ها در حال حاضر به‌وسیلهٔ سوکت برقرار است) را به همراه نام+هش فایل‌هایی که داخل مخزن آن‌ها وجود دارد و نیز شیء Socket ارتباط با آن‌ها را نگه‌دارد. *(لیستی از `PeerConnectionThread` ها)*

طبق اشاراتی که در بخش‌های قبلی داشتیم، هر لحظه که یک همتا با سوکت به ردیاب متصل می‌شود، ردیاب دستور `status` را برای آن ارسال می‌کند تا در پاسخ، IP و پورت همتا را دریافت کرده و در این حالت این همتا به لیست همتایان اضافه می‌شود. (دقت کنید که همتاها به‌وسیلهٔ ترکیب IP:PORT یکتا می‌شوند؛ نه هرکدام به‌صورت جدا). حال برای این‌که فایل‌هایی که آن همتا دارد را نیز بداند، باید دستور `get_files_list` را به همتا ارسال کند و فایل‌های آن‌ها را به آن‌ها نسبت دهد.

پس ردیاب در بدو Accept کردن سوکت از هر همتا، این دو دستور را به‌ترتیب می‌فرستد و در این لحظه داده‌ساختار تکمیل و صحیح است.

حال ممکن است دو سناریو در ادامه رخ دهد:

۱. **مشکلات اتصال با همتا**

۲. **تغییر در فایل‌های مخزن یک همتا**

دربارهٔ مورد اول، شما باید Exception های مربوط به مشکلات IO و Network را طوری Handle نمایید که در صورت قطع سوکت یا عدم توانایی در ارسال هر بسته (Excpetion خوردن در توابع send و receive در حالتی که به مشکلات اتصال مرتبط است)، همتای مربوطه را از لیست همتایان حذف کنید. (و Socket آن را در صورتی که close نشده است close کنید و حتماً تردِ مربوط به پردازش آن را نیز stop کنید وگرنه پایداری برنامهٔ شما زیر سوال می‌رود.)

علاوه بر این، دو دستور در ترمینال سرور درنظر گرفته شده است تا عملیات به‌روز نگه‌داشتن این داده‌ساختار را انجام دهند:

دستور refresh_files :

```
refresh_files
```

این دستور، به ازای تمامِ همتاهای متصل که در لیست همتایان وجود دارند، دستور `get_files_list` را برای آن‌ها ارسال می‌کند و طبق پاسخ‌های دریافتی، نام+هش فایل‌های این همتا را در داده‌ساختار مربوطه تغییر می‌دهد و به‌روز می‌کند. (لیست قبلی فایل‌ها را پاک می‌کند و لیست جدید را جایگزین می‌کند).

دستور reset_connections :

```
reset_connections
```

این دستور، داده‌ساختار مربوط به لیست همتایانِ متصل را پاک می‌کند و از نو، به ازای تمام Socket های برنامه، (یا به نوعی می‌توان گفت به‌ازای همهٔ Thread های پردازشگر سوکت‌ها) دستور `status` را به کلاینت ارسال می‌کند (گویی تمام این سوکت‌ها از نو متصل شده‌اند) و اگر پاسخ هر کدام موفقیت آمیز بود، با IP:PORT اعلامی، به داده‌ساختار وارد می‌کند؛ و اگر موفقیت‌آمیز نبود، سوکت باید بسته شود و ترد مربوطه نیز متوقف شود. (تعریف موفقیت‌آمیز نبودن ارتباط در پایین‌تر اشاره شده است). در صورت موفقیت‌آمیز بودن ارتباط، باید لیست فایل‌های هر همتا نیز مجدداً در لیست تازه‌ساخت وارد شود. (کاری معادل دستور `refresh-files` انجام شود)

دستور list_peers :

```
list_peers
```

با اجرای دستور فوق، لیست تمام همتایانِ متصل (آن‌هایی که در لیست‌مان ذخیره داریم) به صورت زیر چاپ می‌شوند:

```
<peer1-ip>:<port>
<peer2-ip>:<port>
```

```
...
```

اگر هیچ همتایی متصل نشده بود:

```
No peers connected.
```

دستور `list_files` :

```
list_files <IP:PORT>
```

با اجرای دستور فوق، ادمین ردیاب می‌تواند لیست فایل‌های همتای IP:PORT را که در داده‌ساختار خود نگه‌داشته است مشاهده کند (نیازی به ارسال دستور `get_files_list` به همتا نیست). فرمت خروجی این دستور مانند دستور `list` در سمت کنسول همتا می‌باشد.

دستور `get_sends` :

```
get_sends <IP:PORT>
```

با اجرای دستور فوق، ردیاب درخواست `get_sends` (که در بخش‌های قبلی توضیح داده شد) را به همتا ارسال کرده و منتظر پاسخ آن می‌ماند و پس از دریافت پاسخ، آن را نمایش می‌دهد (برای یکتایی پاسخ، خروجی باید بر اساس ترتیب نام فایل به‌صورت الفبا و سپس آی‌پی به‌صورت صعودی باشد). فرمت نمایش هر خط به‌صورت زیر است:

```
<filename> <md5> - <receiver-ip:port>
```

اگر لیست خالی بود باید خط زیر را چاپ کنید:

```
No files sent by <IP>:<PORT>
```

دستور get_receives :

```
get_receives <IP:PORT>
```

با اجرای دستور فوق، ردیاب درخواست get_receives (که در بخش‌های قبلی توضیح داده شد) را به همتا ارسال کرده و منتظر پاسخ آن می‌ماند و پس از دریافت پاسخ، آن را نمایش می‌دهد (برای یکتایی پاسخ، خروجی باید بر اساس ترتیب نام فایل به‌صورت الفبا و سپس آی‌پی به‌صورت صعودی باشد). فرمت نمایش هر خط به‌صورت زیر است:

```
<filename> <md5> - <sender-ip:port>
```

اگر لیست خالی بود باید خط زیر را چاپ کنید:

```
No files received by <IP>:<PORT>
```

###توجه : در تمام دستورهای فوق باید زمانی که هیچ همتای متصل با IP و پورت داده شده وجود نداشت، خطای زیر را چاپ کنید:

```
Peer not found.
```

دستور exit :

```
exit
```

با زدن این دستور، تمام روند کاری ردیاب متوقف می‌شود (ترد Listen کننده و نیز تمام تردهای متصل به همتایان متوقف می‌شوند) و از برنامه خارج می‌شویم.

توجه:

- **نکته ۱ :** در بخش‌های مختلفی، لازم است که یک عنصر به عنصر دیگری درخواستی و منتظر دریافت پاسخ آن بماند. با توجه به اینکه می‌دانیم بطور مثال، هم ارسال درخواست ما به سرور، هم ارسال پاسخ سرور توسط ما، هر دو از طریق یک Stream انجام می‌شود، پس ممکن است در تشخیص بسته‌های مختلف دچار چالش شویم. همچنین چالش دیگر، زمانی است که ما باید یک درخواست را بفرستیم و منتظر پاسخ آن بمانیم (مانند اغلب دستورات سمت کنسول ردیاب). این مسائل با تکمیل پیاده‌سازی ساختارهای داده شده با پروژه اولیه رفع خواهد شد ولی دقت کنید که برای دستورات یک Timeout برابر با **۵۰۰ میلی‌ثانیه** در نظر بگیرید که هم برای خود سوکت و هم برای دریافت پاسخ درخواست‌ها، باید لحاظ شود**(تعریف موفقیت‌آمیز نبودن ارتباط)**؛ در صورتی که یک درخواست Timeout شود باید در *System.err* عبارت زیر را چاپ کنید:

```
Request Timed out.
```

راهنمایی: از تابع `setSoTimeout` استفاده کرده و هنگام خواندن از جریان داده ورودی باید `SocketTimeoutException` را Handle کنید.

- **نکته ۲ :** برای پیاده‌سازی تمیز و اصولی‌تر، کلاس `Message` و برخی ساختارهای مربوط به تردها در سمت همتا و ردیاب (مانند کلاس‌های `ConnectionThread` و کلاس‌های وابسته و همچنین کلاسهای دیگر) در پروژه قرار گرفته و بعضاً به‌صورت نیمه‌کاره پیاده‌سازی شده که باید آن‌ها را کامل کرده و در پیاده‌سازی بخش‌های دستورات تحت شبکه، از آن‌ها استفاده کنید. همچنین یک کلاس MD5Hash وجود دارد که شما باید پیاده‌سازی آن را کامل کنید و برای بدست آوردن Hash یک فایل، از آن استفاده کنید. **توجه کنید که با وجود اینکه بخش قابل توجهی از فایل‌های پروژه دارای پیاده‌سازی اولیه هستند، ولی شما لازم است که به تمام کد پروژه هر دو سمت ردیاب و همتا و نیز کلاسهای مشترکِ از قبل پیاده شده نیز تسلط کامل داشته باشید و این موضوع هنگام تحویل بررسی خواهد شد.**
- **نکته ۳:** از آنجا که تشکیل‌دادن یک شبکه با اعضای مختلف و ipهای متفاوت از یکدیگر، از اهداف این تمرین نیست، **در تست‌های خودتان، ip تمام اعضای شبکه، اعم از همتا و ردیاب، آدرس localhost یا 127.0.0.1 خواهد بود؛** بنابراین در این تست‌ها محل تمایز تمام اعضای شبکه، پورتی است که به واسطه پاسخ سایر اعضا را خواهند داد. به طور دقیق‌تر، هر عضو از شبکه، یک پورت باز جهت پاسخ‌دهی به درخواست‌های سایر اعضای شبکه دارد که آن پورت به عنوان شناسه آن عضو در نظر گرفته می‌شود

(هر عضو از شبکه پورت‌های باز دیگری دارد اما این پورت‌ها به عنوان کلاینت به سایر اعضای شبکه متصل هستند.). **البته پروژه شما باید بتواند روی شبکه واقعی و IP های مختلف کار کند که این موضوع در داوری خودکار بررسی می‌شود.**

ساختار پروژه و داوری

با توجه به سیستم داوری، پروژهٔ شما باید شامل ساختار زیر باشد:

```
common/
    ├── models
    │   ├── <Model-1>.java
    │   ...
    │   └── <Model-n>.java
    ├── ...
    └── <Some java files>
tracker/
    ├── <files/packages>
    ...
    └──  TrackerMain.java
peer/
    ├── <files/packages>
    ...
    └──  PeerMain.java
lib/
    ├── <library-1>.jar
    ...
    └── <library-n>.jar
```

که `common` برای تعریف کلاس‌های مشترک بین Tracker و Peer به‌کار می‌رود؛ و پکیج کلاس‌های زیر مجموعهٔ آن نیز با `common` شروع می‌شود. پکیجی که بالای `TrackerMain.java` نوشته می‌شود نیز برابر `tracker` و پکیجی که بالای `PeerMain.java` نوشته می‌شود نیز برابر `peer` می‌باشد. همچنین باید فایل `.jar` کتابخانه‌هایی که در کدتان از آن‌ها استفاده کرده‌اید را در پوشهٔ `lib` قرار دهید.

با توجه به این ساختار، با دستور زیر در فولدر اصلی پروژه، کل پروژه شامل Peer و Tracker کامپایل می‌شود (دستور روی Linux یا WSL اجرا می‌شود. دقت داشته باشید که نسخهٔ جاوای مناسب روی آن نصب داشته باشید).

```
javac -cp "lib/*" -d out $(find -name '*.java')
```

(اگر لازم دارید دستور بالا را روی خط فرمان ویندوز اجرا کنید، بجای بخش `$(find -name '*.java')` ، لیست آدرس نسبی تمام فایل‌های `.java` پروژه را با فاصله بین آن‌ها قرار دهید.)

و در نتیجه، دستورات اجرای Tracker و Peer که در ابتدای صورت سوال به حالت `java PeerMain ....` آمده بود، به‌صورت زیر کامل خواهد بود:

```
java -cp "lib/*:out" peer.PeerMain <peer-ip:port> <tracker-ip:port> <fold
```

```
java -cp "lib/*:out" tracker.TrackerMain <listen-port>
```

شما می‌توانید فایل پروژهٔ اولیه را از اینجا دانلود کنید. لازم به ذکر است که در صورتی که از لینوکس و یا WSL استفاده می‌کنید، می‌توانید جهت کامپایل و ایجاد فایل اجرایی jar از همتا و ردیاب، از `build_tracker.sh` و `build_peer.sh` (که در پروژه اولیه قرار دارند) استفاده کنید. برای اجرای فایل‌های jar نیز می‌توانید از دستور `java -jar x.jar` استفاده کنید.

شما می‌توانید ویدئوی کارگاه آموزشی مربوط به این تمرین و مبحث شبکه را از این لینک مشاهده کنید. همچنین شما می‌توانید سناریوهای تست‌کیس‌ها را از این سند مشاهده کنید.