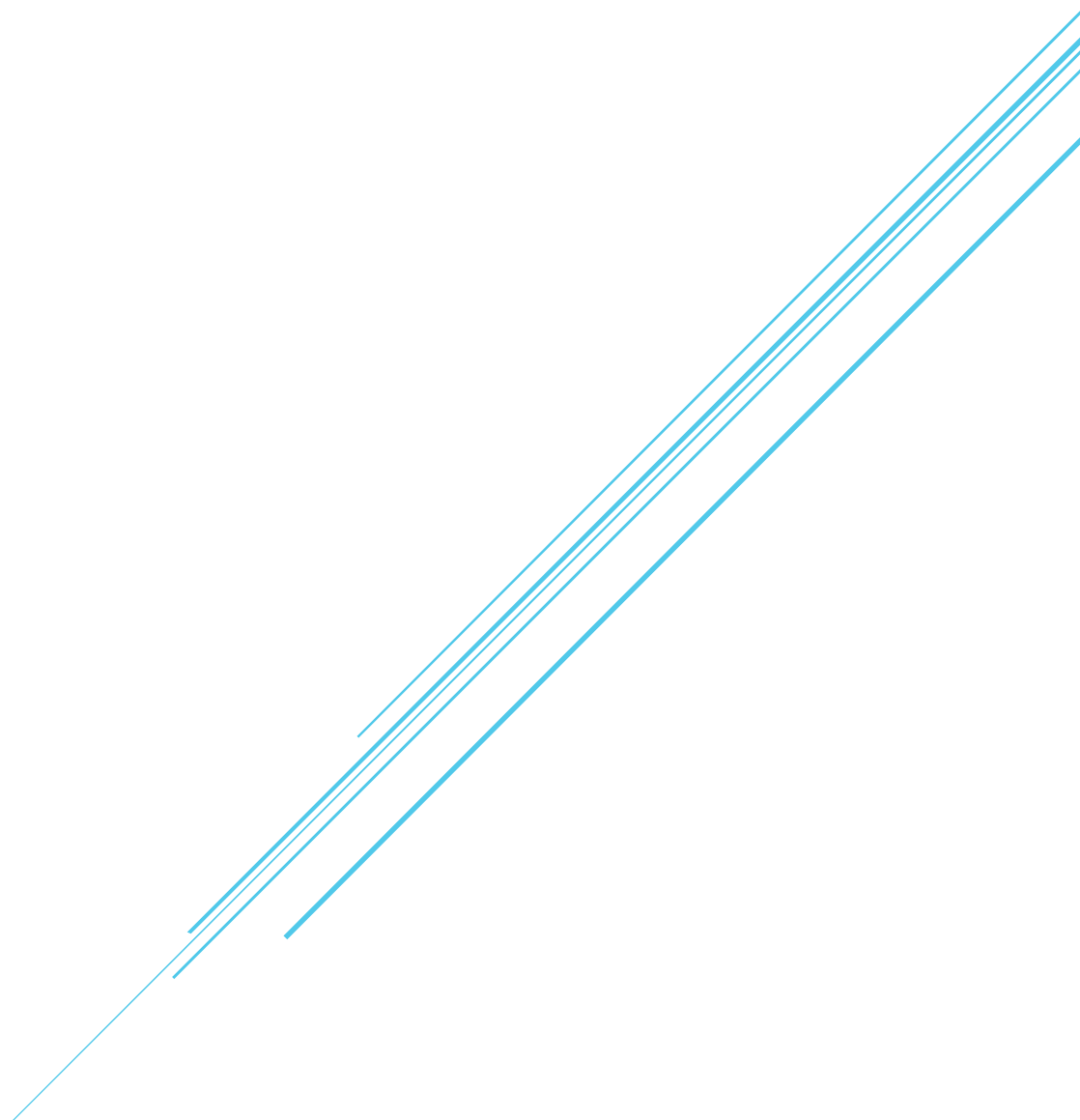


گزارش

اسم درس

۱۴۰۲/۸/۸



• نام و نام خانوادگی :

• نام استاد :

• گروه :

گزارش کار سوال ۱

خلاصه ی کار:

ابتدا نیاز برای این که یک فایل حجیم را ارسال کنیم، آن را به قطعه های کوچکتر تقسیم کنیم. به همین منظور باید تصمیمی برای تقسیم بندی ارسال و دریافت فایل انجام میگیرفتم. به همین منظور متغیر `FILE_PART` را تعریف کردم که در هر دو برنامه ی `Client` و `Server` باید عدد یکسانی داشته باشند. برای سادگی کار آن را در تعریف کلاس آورده ام وگرنه میتوانستیم برای اجرای بهتر برنامه، ابتدا کلاینت به سرور متصل شده و مقدار این عدد را به سرور اعلام کند.

پس از تعیین این عدد، باید فایل ها را به همین تعداد تقسیم میکردم. به طور پیش فرض عدد ۸ که در متن سوال نیز ذکر شده بود را انتخاب کردم. در این صورت فایل ها به ۸ قسمت مختلف تقسیم بندی میشوند و هر بار یک بخش از فایل خوانده میشود (پارت های قبلی دور ریخته میشوند و پارت جدید خوانده میشود) و یک `thread` جدید ایجاد شده و تابع `send_file` را صدا میزند. این تابع مقدار مورد نظر از فایل را خوانده و سپس یک `TCP` جدید به سرور باز میکند و فایل را برای آن ارسال میکند. در این صورت ۸ ترد همزمان با هم قسمت های متخلف آن فایل را برای سرور ارسال میکنند.

در سمت `Server` نیز، همان عدد `FILE_PART` تعریف شده است و این بار برای دریافت فایل از آن استفاده میشود. به این صورت که ۸ ترد جدید باز کرده و هر کدام وظیفه ی یک بخش از فایل را خواهند داشت. این ترد ها به تابع `receive_file` متصل شده و بعد از دریافت هر بخش از فایل، آن را در فایلی به صورت `temp` ذخیره میکنند. سپس کل اتصال را میبندند.

پس از اینکه کار کلاینت تمام شد، با تابع `end_file`، اعلام میکند که کار خود را انجام داده است و دستوری را ارسال میکند که سرور با دریافت آن متوجه میشود که ارسال فایل ها تمام شده است و میتواند فایل های `temp` را با هم `merge` کند و فایل اصلی را باز بسازد.

بعد از اجرای اولیه و رفع مشکل های سطحی، به مشکلی تازه برخوردم. این مشکل که فایل های ارسال شده به ترتیب ارسال، دریافت نمیشوند. و بعد از این که فایل های `merge` میشدند، فایل های `corrupt` شده بودند.

گزارش کار

به همین دلیل یک فایل متنی با متن مشخص را فرستادم تا ببینم فایل خروجی نهایی به صورت ارسال هست یا نه. این فایل از اعداد ۱ تا ۱۰۰ را در هر خط وارد کرده بود.

پس از بررسی این فایل متوجه شدم که ترتیب دریافت فایل ها میتواند در حین ارسال و دریافت تغییر کند. به همین دلیل، برای ارسال هر فایل، یک رشته ی کوچک از عدد index قسمتی که در حال ارسال نیز اضافه کردم.

```
sock.connect((sock.host, sock.port))  
sock.sendall(bytes(str(index) + ""), encoding='utf-8') + data)
```

در این صورت وقتی که میخواستم در سرور merge را انجام بدهم، میتوانستم بفهمم که هر فایل باید در کدام بخش قرار بگیرد.

بعد از اضافه کردن این بخش و بررسی دیگر بخش ها و رفع ارور ها، توانستم که کار را به پایان برسانم.

به صورت پیش فرض یک فایل file.mkv را در دایکتوری جاری در نظر گرفته ام، و پورت آن را نیز ۱۲۳۴۵ قرار داده ام. چون سرور روی همین کامپیوتر قرار دارد، هاست آن را نیز localhost در نظر گرفته ام.

همچنین بخشی را اضافه کردم که بتوان برای قسمت debug از آن استفاده کرد. به این صورت که فایل های temp را نگه داری میکند تا به محتوای آن ها دسترسی داشته باشیم. در صورتی که KEEP_TEMP_FILE را false کنیم، بعد از merge فایل ها، آن ها را پاک میکند.

برای بهینه سازی میتوان در همان بخشی که فایل ها دریافت میشوند، چک کرد که کدام فایل دریافت شده و آن را در همان فایل temp خود ذخیره کرد و در آخر نیز فایل های temp را به ترتیب خوانده و به یک فایل اصلی (که در خود سرور اسم آن را تعریف کرده ام) اضافه کنیم.

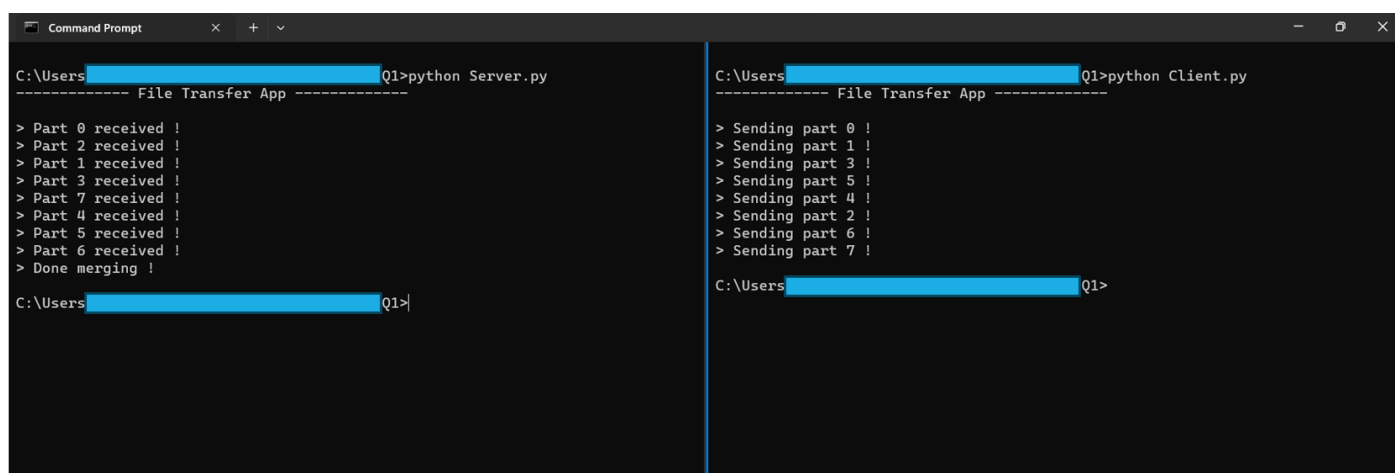
گزارش کار

نحوه ی اجرای برنامه:

ابتدا در cmd یا هر نوع ترمینال دیگری، فایل سرور را با python باز میکنیم، سپس فایل client را با python باز میکنیم.

خروجی برای ما نمایش داده میشود و پس از انجام همه ی موارد، هر دو برنامه بسته میشوند.

عکس از اجرای برنامه در CMD:



```
C:\Users\Q1>python Server.py
----- File Transfer App -----
> Part 0 received !
> Part 2 received !
> Part 1 received !
> Part 3 received !
> Part 7 received !
> Part 4 received !
> Part 5 received !
> Part 6 received !
> Done merging !
C:\Users\Q1>

C:\Users\Q1>python Client.py
----- File Transfer App -----
> Sending part 0 !
> Sending part 1 !
> Sending part 3 !
> Sending part 5 !
> Sending part 4 !
> Sending part 2 !
> Sending part 6 !
> Sending part 7 !
C:\Users\Q1>
```

عکس از فایل های ایجاد شده (همراه با temp ها):

0.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
1.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
2.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
3.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
4.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
5.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
6.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
7.temp	2023-10-30 7:47 PM	TEMP File	28,786 KB
Client.py	2023-10-27 9:43 PM	JetBrains PyCharm	3 KB
film.mkv	2023-10-24 10:22 PM	MKV Video File (V...	230,282 KB
film_final.mkv	2023-10-30 7:47 PM	MKV Video File (V...	230,282 KB
Server.py	2023-10-30 7:47 PM	JetBrains PyCharm	4 KB

گزارش کار سوال ۲

خلاصه ی کار:

برای این برنامه، ابتدا باید تصمیم می‌گرفتم که برنامه به کدام روش نوشته شود. یک روش این بود که سرور بین هر دو کلاینت قرار بگیرد و هر پیامی که رد و بدل میشود از طریق سرور انجام شود. در این صورت وقتی که برنامه ی سرور بسته میشد پیام های ارسالی عملاً از بین میرفت و دیگر دو کلاینت نمیتوانستند با هم ارتباط برقرار کنند و حتی وقتی یکی از کلاینت ها آفلاین میشد، حتماً باید سرور چک میکرد این آفلاین شدن را و خب وقتی از پروتکل UDP استفاده میکنیم، این کار ممکن نیست. برای همین، از سرور صرفاً فقط برای اتصال دو تا کلاینت استفاده میکنم و بعد از آن نیازی به سرور نخواهد بود و سرور میتواند از دسترس خارج شود.

در صورتی که یکی از کلاینت ها آفلاین شود، نیاز است که حتماً به سرور متصل شود و پورت های ارسال و دریافت را عیناً وارد کند که به کلاینت دیگر دوباره وصل بشود.

چالش دیگری که داشتم این بود که چطور از یک کامپیوتر به کامپیوتر دیگر چندین کانکشن مختلف داشته باشم به صورتی که در یکدیگر تداخلی نداشته باشند. یک راه حل این بود که پورت ها را جوری تنظیم کنم که پورت ها تکراری نباشند، در این صورت میتوانیم از یک کامپیوتر به کامپیوتر دیگر چندین کانکشن متخلف بزنیم و چت های خاص خود را داشته باشیم.

چالش دیگری که با آن مواجه شدم، آفلاین شدن و تضمین دریافت پیام در کلاینت دیگر بود. برای این کار یک تابع با نام `check_online` نوشتم و این تابع هر بار یک کانکشن TCP به کلاینت دیگر زده و اگر قبول میشد (تابع `send_online`) به این معنی بود که کلاینت دیگر هنوز آنلاین هست و میتواند پیام را دریافت کند.

در صورتی که کلاینت آفلاین بود و پیام را دریافت نمیکرد، پیام ها در یک لیست ذخیره میشود و در اولین زمانی که کلاینت دیگر باز آنلاین شود، پیام ها را به ترتیب برای آن کلاینت ارسال میکند.

بعد از اجرا کردن کلاینت ها، یک پورت برای دریافت و یک پورت برای ارسال در نظر گرفته میشود. همچنین تابعی در نظر گرفته شده است که آپی هر کلاینت را نشان دهد.

پس از اجرا، هر کلاینت سعی میکند که آدرس سروری که وارد برنامه میکنیم وصل شود و در صورتی که کلاینت دیگر نیز با همان پورت ها به سرور متصل شود، سرور هر دو کلاینت را بهم وصل میکند و از چرخه ی آن ها خارج میشود. سپس هر کلاینت منتظر دریافت یا ارسال پیام از/به کلاینت دیگر میشود.

برای امنیت بیشتر تضمین پیام های ارسالی، هر بار که برنامه بسته میشود، لیست پیام های ارسال نشده داخل یک فایل ذخیره میشود و در صورتی که کلاینت را دوباره باز کنیم، لیست فایل ها دوباره load میشود. در فایل server، یک پورت ثابت برای ارتباط با سرور در نظر گرفته شده است که میتوان آن را تغییر داد یا پیش فرض آن را استفاده کرد.

بعد از اجرای سرور، سرور منتظر دریافت هر کانکشنی از طرف کلاینت ها میشود، در صورتی که یک کلاینت متصل شود، اطلاعات ready را برای آن میفرستد و منتظر دیگر کلاینت با همان اطلاعات و پورت ها میشود و در صورتی که کلاینت دیگر متصل شود و تعداد کلاینت هایی که پورت های آن ها با هم یکسان است از ۱ بیشتر شود، به آن ها اطلاع داده و اطلاعاتشان را برای همدیگر میفرستد.

و اطلاعات آن ها را نیز نگه داری میکند که در صورتی که کلاینتی قطع شد و خواست دوباره به کلاینت خود وصل شود، بتواند از آن اطلاعات استفاده کند و دوباره وصل شود.

همچنین توابع دیگری دارد که در قسمت client از آن ها نام برده شد.

نحوه ی اجرای برنامه:

برای اجرا بهتر است موارد زیر رعایت شود که مشکلی برای اجرای برنامه پیش نیاید.

۱. کاملاً ضروری است که دو کلاینت از یک کامپیوتر نباشند، اگر کامپیوتر دیگری در دسترس نیست از شبیه ساز یا Linux subsystem یا هر روش دیگری باید استفاده کرد که آپی کلاینت ها با هم یکسان نباشند.

۲. بهتر است سرور روی لینوکس اجرا شود یا روی Linux subsystem اجرا شود. اگر یک کلاینت نیز کنار سرور اجرا شود، مشکلی ایجاد نمیکند

گزارش کار

۳. در صورتی که پورت های پیش فرض قابل استفاده نباشند، برنامه با خطا مواجه میشود. بهتر است از پورت های خودتان استفاده کنید.

۴. برای استفاده از برنامه، چون از آیدی پابلیک استفاده میکند، بهتر است اینترنت متصل باشد ولی اگر همه ی سرور و کلاینت ها روی یک سیستم اجرا میشوند، نیازی به داشتن اینترنت نیست.

پس از رعایت همه ی حالات بالا،

ابتدا سرور را در Linux subsystem اجرا میکنیم.

```
(~)
[ ( on master ★) → python3 Server.py
----- Connector Server -----

* Server Information> 172.18.146.168:44444
```

پس از اجرای آن، میتوان دید که سرور اجرا میشود و آدرس خود را به ما میدهد، این آدرس را به صورت کامل کپی کرده و در کلاینت وارد میکنیم. سپس موارد زیر را کلاینت به ما نشان میدهد:

```
(~)
[ ( on master ★) → python3 Client.py
----- P2P Messaging App (ME:@172.18.146.168) -----

Enter Server [IP:PORT] > 172.18.146.168:44444

+ Connecting to the server ...
+ Connected To 172.18.146.168:44444
+ Waiting for a peer ...
```

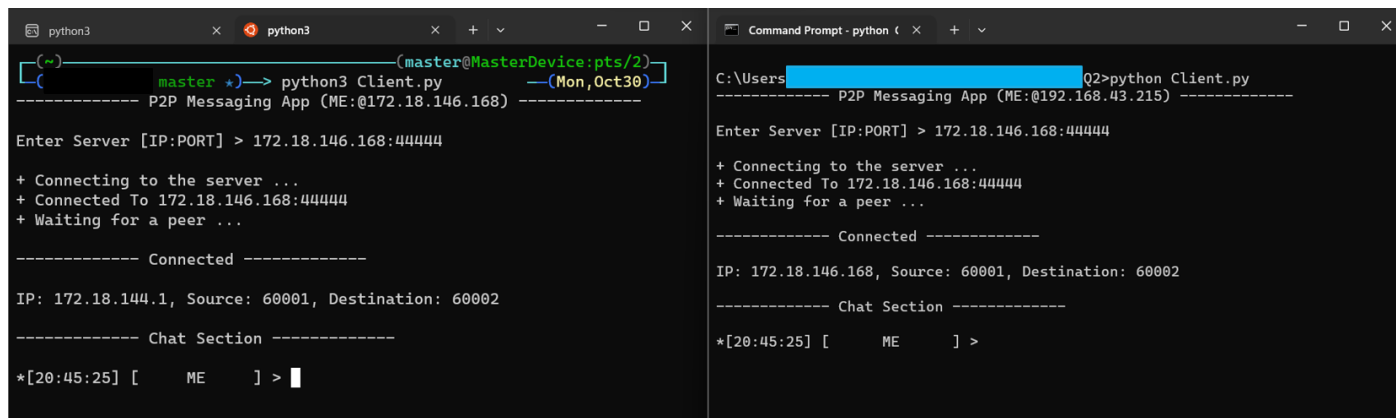
در همین زمان، سرور تغییر کرده و موارد زیر را نشان میدهد:

```
(~)
[ ( on master ★) → python3 Server.py
----- Connector Server -----

* Server Information> 172.18.146.168:44444
+ Connection accepted from > ('172.18.146.168', 60001)
```

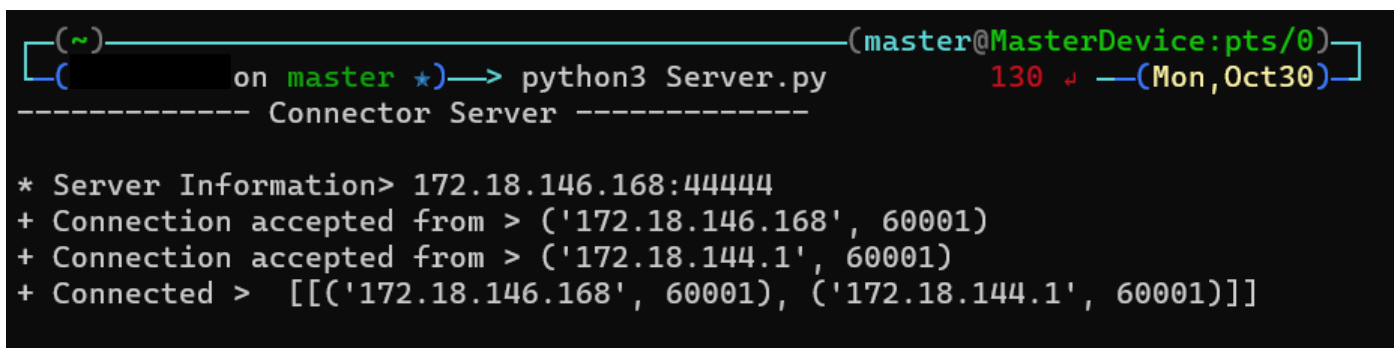
گزارش کار

در ادامه، سرور و کلاینت اول منتظر اتصال کلاینت دوم هستند. برای این کار من از CMD ویندوز استفاده کرده و به سرور متصل میشوم:



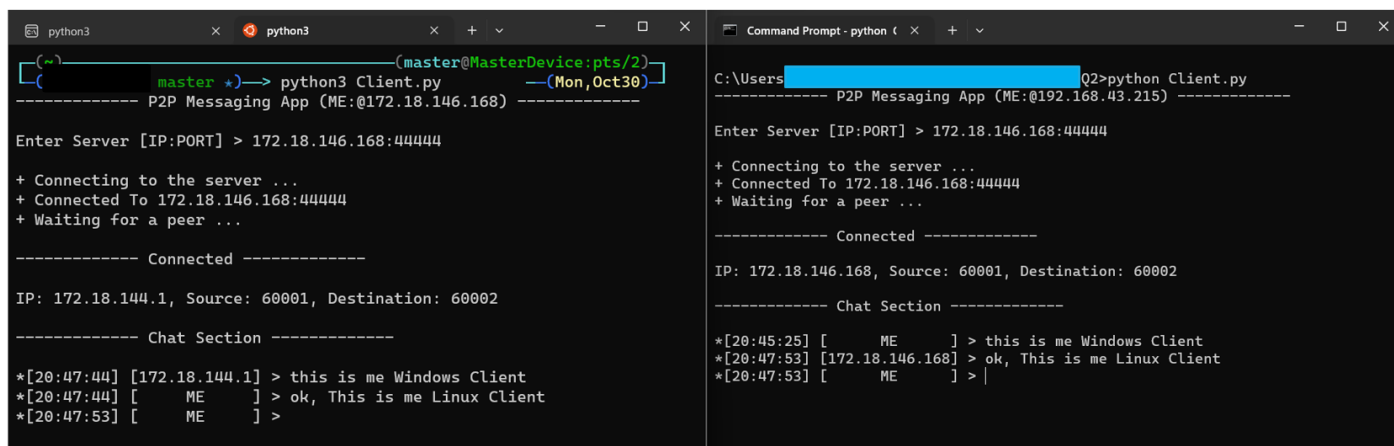
The image shows two terminal windows side-by-side. The left window is a Windows Command Prompt titled 'python3' with a user prompt '(master@MasterDevice:pts/2)'. It shows the execution of 'python3 Client.py', which connects to the server at 172.18.146.168:4444. The right window is a Linux terminal titled 'Command Prompt - python' with a user prompt 'C:\Users\...Q2>python Client.py'. It also shows the execution of 'python Client.py', which connects to the server at 172.18.146.168:4444. Both windows show the 'P2P Messaging App' interface with a 'Chat Section'.

در این حالت میتوانید مشاهده کنید که هر دو کلاینت یکدیگر متصل میشوند، همچنین پیغام های سرور به شکل زیر خواهد بود:



The image shows a terminal window titled 'python3' with a user prompt '(master@MasterDevice:pts/0)'. It shows the execution of 'python3 Server.py', which displays 'Connector Server' information. The output shows two connections: one from 172.18.146.168:4444 and another from 172.18.144.1:60001. The output also shows the connected state for both connections.

سپس میتوان در کلاینت ها پیام دریافت یا ارسال کرد.



The image shows two terminal windows side-by-side. The left window is a Windows Command Prompt titled 'python3' with a user prompt '(master@MasterDevice:pts/2)'. It shows the execution of 'python3 Client.py', which connects to the server at 172.18.146.168:4444. The right window is a Linux terminal titled 'Command Prompt - python' with a user prompt 'C:\Users\...Q2>python Client.py'. It also shows the execution of 'python Client.py', which connects to the server at 172.18.146.168:4444. Both windows show the 'P2P Messaging App' interface with a 'Chat Section'. The chat section shows messages from both clients: 'this is me Windows Client' and 'ok, This is me Linux Client'.

علامت * برای نشان دادن آنلاین بودن کلاینت مورد استفاده قرار میگیرد. در صورتی که کلاینت قطع شود، دیگر * را نمایش نمیدهد:

گزارش کار

```
python3 python3
(master@MasterDevice:pts/2)
on master *) -> python3 Client.py (Mon, Oct30)
----- P2P Messaging App (ME:@172.18.146.168) -----

Enter Server [IP:PORT] > 172.18.146.168:44444

+ Connecting to the server ...
+ Connected To 172.18.146.168:44444
+ Waiting for a peer ...

----- Connected -----

IP: 172.18.144.1, Source: 60001, Destination: 60002

----- Chat Section -----

*[20:47:44] [172.18.144.1] > this is me Windows Client
*[20:47:44] [ ME ] > ok, This is me Linux Client
*[20:47:53] [ ME ] > ok
[20:48:55] [ ME ] >

C:\Users\ [redacted] Q2>python Client.py
----- P2P Messaging App (ME:@192.168.43.215) -----

Enter Server [IP:PORT] > 172.18.146.168:44444

+ Connecting to the server ...
+ Connected To 172.18.146.168:44444
+ Waiting for a peer ...

----- Connected -----

IP: 172.18.146.168, Source: 60001, Destination: 60002

----- Chat Section -----

*[20:45:25] [ ME ] > this is me Windows Client
*[20:47:53] [172.18.146.168] > ok, This is me Linux Client
*[20:47:53] [ ME ] >

----- Exiting P2P Messaging App -----
```

برای اتصال دوباره، کلاینت را به سرور متصل میکنیم. چون پورت ها را عوض نکردیم و از پورت های پیش فرض استفاده میکنند، بهم دیگر متصل میشوند و بعد از اتصال، پیام های ارسال نشده به ترتیب برای کلاینت قطع شده ارسال میشوند:

```
----- Connected -----

IP: 172.18.144.1, Source: 60001, Destination: 60002

----- Chat Section -----

*[20:47:44] [172.18.144.1] > this is me Windows Client
*[20:47:44] [ ME ] > ok, This is me Linux Client
*[20:47:53] [ ME ] > ok
[20:48:55] [ ME ] > not ok
[20:49:46] [ ME ] > where are you?
[20:49:50] [ ME ] > hello???
*[20:50:12] [ ME ] >

^C
C:\Users\ [redacted] Q2>python Client.py
----- P2P Messaging App (ME:@192.168.43.215) -----

Enter Server [IP:PORT] > 172.18.146.168:44444

+ Connecting to the server ...
+ Connected To 172.18.146.168:44444
+ Waiting for a peer ...

----- Connected -----

IP: 172.18.146.168, Source: 60001, Destination: 60002

----- Chat Section -----

*[20:50:12] [172.18.146.168] > ok
*[20:50:12] [172.18.146.168] > not ok
*[20:50:12] [172.18.146.168] > where are you?
*[20:50:12] [172.18.146.168] > hello???
*[20:50:12] [ ME ] > |
```

سرور نیز همه ی اتفاقات را رصد میکند و پیغام آن را نشان میدهد:

```
(~) (master@MasterDevice:pts/0)
on master *) -> python3 Server.py 130 (Mon, Oct30)
----- Connector Server -----

* Server Information> 172.18.146.168:44444
+ Connection accepted from > ('172.18.146.168', 60001)
+ Connection accepted from > ('172.18.144.1', 60001)
+ Connected > [[('172.18.146.168', 60001), ('172.18.144.1', 60001)]]
+ Connection accepted from > ('172.18.144.1', 60001)
+ Reconnecting ('172.18.144.1', 60001) - ('172.18.146.168', 60001) ...
```