فهرست

فهرست مطالب

- 5.4. کدنویسی سرور
- 5.5. کدنویسی کلاینت
- 5.6. اجرا و تست

۶. مباحث پیشرفته

- 6.1. مدیریت قطع ارتباط ناگهانی
- 6.2. پیام خصوصی (Private Messaging)
- 6.3. لاگ‌گیری (Logging)
- 6.4. امنیت پایه (SSL/TLS)

مقدمه

1. مقدمه

در دنیای امروز که نرم‌افزارها روزبه‌روز پیچیده‌تر، تعاملی‌تر و متصل‌تر می‌شوند، دو مهارت کلیدی برای هر برنامه‌نویس سیستم یا برنامه‌نویس بک‌اند اهمیت ویژه‌ای پیدا کرده‌اند: **برنامه‌نویسی هم‌زمان (Multithreading)** و **برنامه‌نویسی سوکت (Socket Programming)**.

چرا Multithreading و Socket Programming مهم هستند؟

- تصور کنید در حال ساخت یک برنامه هستید که باید به چند کاربر به‌صورت هم‌زمان پاسخ دهد (مانند یک سرور چت).
- یا برنامه‌ای که همزمان باید فایل‌هایی را از اینترنت دانلود کند، وضعیت کاربران را ذخیره کند و رابط کاربری را نیز فعال نگه دارد.
- یا شاید شما یک بازی آنلاین یا نرم‌افزار کنترل تجهیزات شبکه‌ای توسعه می‌دهید.

در تمام این موارد، نیاز به اجرای هم‌زمان چندین عملیات داریم — بدون اینکه برنامه هنگ کند یا به کندی واکنش نشان دهد.

تعریف مفاهیم پایه:

خب Multithreading چیست ؟

درواقع Multithreading به معنای اجرای هم‌زمان چند **رشته (Thread)** از کد در یک فرایند (Process) است. در زبان ساده:

> تردها مسیرهای اجرایی جداگانه‌ای هستند که می‌توانند هم‌زمان (یا شبه‌هم‌زمان) اجرا شوند.

پایتون از طریق ماژول `threading` امکان ساخت و مدیریت این تردها را فراهم می‌کند. این قابلیت به‌ویژه برای کارهای **I/O-bound** (مثل کار با فایل یا شبکه) بسیار مؤثر است.

> توجه: پایتون دارای محدودیتی به نام **GIL (Global Interpreter Lock)** است که اجازه نمی‌دهد چند ترد واقعاً به‌صورت هم‌زمان روی چند هسته اجرا شوند — مگر در صورت انجام عملیات I/O یا استفاده از multiprocessing.

خب Socket Programming چیست؟

درواقع Socket Programming روشی برای برقراری ارتباط بین دستگاه‌ها (یا پردازش‌ها) از طریق شبکه است. این نوع برنامه‌نویسی، پایه و اساس اینترنت و شبکه‌های توزیع‌شده است.

پایتون با استفاده از ماژول `socket` امکان ساخت کلاینت و سرور را با چند خط کد فراهم می‌کند. با سوکت‌ها می‌توان داده‌ها، پیام‌ها، فایل‌ها و حتی ویدیوها را بین دو یا چند سیستم منتقل کرد.

ساختار این درسنامه

در این درسنامه گام به گام با مفاهیم و ابزارهای مربوط به Multithreading و Socket Programming آشنا می‌شوید و سپس با کمک آن‌ها یک پروژه عملی یعنی **اتاق گفت‌وگوی چندکاربره (Chatroom)** طراحی و پیاده‌سازی می‌کنیم.

مراحل یادگیری به ترتیب زیر است:

۱. **یادگیری Multithreading**:
   - ساخت Threadها
   - همگام‌سازی و اشتراک منابع
   - مثال‌های عملی
۲. **یادگیری Socket Programming**:
   - TCP و UDP
   - ساخت سرور و کلاینت
   - ارسال و دریافت پیام
۳. **پروژه عملی Chatroom**:
   - معماری پروژه

- مدیریت چند کلاینت با Threadها
- ارسال گروهی پیام‌ها
- پیاده‌سازی امنیت و ویژگی‌های تکمیلی

نکته‌ی فنی درباره‌ی GIL

> پایتون به دلیل وجود **Global Interpreter Lock (GIL)** فقط اجازه می‌دهد یک Thread در لحظه درون مفسر پایتون اجرا شود. اما: برای عملیات I/O (مثلاً کار با شبکه یا دیسک) این محدودیت معمولاً مشکلی ایجاد نمی‌کند. برای عملیات CPU-bound بهتر است از ماژول `multiprocessing` استفاده کنید.

پیش نیاز ها

2. پیش نیاز ها

برای آنکه مثال‌های این درسنامه را بدون مشکل اجرا کنید، ابتدا موارد زیر را آماده کنید:

۱. **نصب Python 3.8+**

- ویندوز: نصب‌کننده را از وب‌سایت رسمی Python دانلود و اجرا کنید.
- macOS:

```
brew install python
```

لینوکس (مثلاً Ubuntu/Debian):

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

۱. پس از نصب، با اجرای `python3 --version` یا `python --version` مطمئن شوید که نسخه‌ی موردنظر فعال است.

۲. **ایجاد و فعال‌سازی محیط مجازی (Virtual Environment)** کار در محیط مجازی، تضمین می‌کند که بسته‌ها و نسخه‌های موردنیاز هر پروژه از هم جدا بمانند.

```
python3 -m venv venv          # ساخت محیط مجازی با نام venv
source venv/bin/activate      # فعال‌سازی در macOS/Linux
venv\Scripts\activate         # فعال‌سازی در Windows
```

۱. پس از فعال کردن، در prompt ترمینال نام محیط (مثلاً `(venv)`) نمایش داده می‌شود.

۲. **نصب وابستگی‌ها** اکثر مثال‌ها صرفاً از ماژول‌های استاندارد `threading` و `socket` استفاده می‌کنند. در صورت نیاز به امکانات اضافی:

- **SSL/TLS**

```
pip install pyOpenSSL
```

لاگ‌گیری پیشرفته (اختیاری)

```
pip install loguru
```

برای مدیریت قالب‌بندی کد (اختیاری):

```
pip install black
```

۱. **ویرایشگر و ابزار توسعه** برای نوشتن و خواندن آسان‌تر کد پیشنهاد می‌شود از یک IDE یا ویرایشگر پیشرفته استفاده کنید:
   - VS Code (با افزونه‌های Python و Pylance)
   - PyCharm
   - یا هر ویرایشگر دیگری که از هایلایت سینتکس و تکمیل خودکار پشتیبانی کند.

۲. **آشنایی پایه با پایتون** قبل از شروع این درسنامه لازم است:
   - سینتکس پایه (تعریف تابع، حلقه‌ها، شرط‌ها، مدیریت استثناء)
   - ساختار داده‌های اصلی (لیست، دیکشنری، مجموعه)
   - کار با ماژول‌های استاندارد و نصب بسته‌ها با `pip`

۳. **مفاهیم ابتدایی شبکه** درک کلی از موارد زیر به فهم بهتر بخش سوکت کمک می‌کند:
   - آدرس IP و پورت
   - تفاوت TCP و UDP
   - مدل OSI یا TCP/IP و جایگاه سوکت در آن

> **نکته عملی:** پس از فعال کردن محیط مجازی، با اجرای:

```
python --version
pip list
```

> بررسی کنید که نسخه‌ی پایتون و بسته‌های نصب‌شده مطابق آنچه می‌خواهید باشند.

مباحث Multithreading در پایتون

3. مباحث Multithreading در پایتون

در این بخش با ابزارها و الگوهای پایه برای کار با چندرشته‌ای (Multithreading) در پایتون آشنا می‌شویم. تمرکز ما بر روی ماژول استاندارد `threading` است که امکانات زیر را در اختیار ما قرار می‌دهد:

۱. ایجاد و مدیریت Thread
۲. همگام‌سازی (Synchronization) برای جلوگیری از شرایط رقابتی
۳. مثال‌های عملی برای درک بهتر

3.1. ماژول `threading`

ماژول `threading` در پایتون کلاس‌ها و توابع زیر را ارائه می‌کند:

- **`Thread`** نماینده یک رشته‌ی اجرا (Thread) است.
- **`Lock`** و **`RLock`** برای محافظت از بخش‌های بحرانیِ برنامه (Critical Sections)
- **`Event`** برای هماهنگ‌سازی با علامت‌دهی ساده (Flag-based synchronization)
- **`Condition`** برای هماهنگ‌سازی پیچیده‌تر با امکان انتظار (wait) و اطلاع‌رسانی (notify)
- **`Semaphore`** برای کنترل دسترسی همزمان چند Thread به منابع محدود

```
import threading

print("CPU cores:", threading.active_count())
```

3.2. ایجاد و اجرای Thread

دو روش اصلی برای تعریف یک Thread وجود دارد:

۱. **زبان تابعی (Functional API):** با فراخوانی `threading.Thread(target=…, args=…)`
۲. **زبان شی‌گرا (Subclassing):** ارث‌بری از کلاس `Thread` و override کردن متد `run()`

3.2.1. روش تابعی

```python
import threading
import time

def worker(name, delay):
    for i in range(3):
        print(f"[{name}] iteration {i+1}")
        time.sleep(delay)

t1 = threading.Thread(target=worker, args=("Thread-A", 1))
t2 = threading.Thread(target=worker, args=("Thread-B", 1.5))

t1.start()
t2.start()

t1.join()
t2.join()
print("All threads finished.")
```

3.2.2. روش ارث‌بری

```python
import threading
import time

class WorkerThread(threading.Thread):
    def __init__(self, name, delay):
        super().__init__(name=name)
        self.delay = delay

    def run(self):
        for i in range(3):
            print(f"[{self.name}] iteration {i+1}")
            time.sleep(self.delay)

t = WorkerThread(name="MyWorker", delay=1)


```

```
t.start()
t.join()
```

3.3. همگام‌سازی (Synchronization)

وقتی چند Thread به یک منبع مشترک (مثل یک متغیر یا فایل) دسترسی دارند، اگر کنترل درستی روی این دسترسی‌ها وجود نداشته باشد، *شرایط رقابتی* (Race Conditions) رخ می‌دهد. برای جلوگیری از این مشکل از ابزارهای زیر استفاده می‌کنیم:

۱. **`Lock`** ساده‌ترین قفل؛ فقط یک Thread می‌تواند وارد بخش بحرانی شود.

```
lock = threading.Lock()
shared_counter = 0

def safe_increment():
    global shared_counter
    with lock:
        temp = shared_counter
        temp += 1
        shared_counter = temp
```

۲. **`RLock`** قفل بازگشتی که اجازه می‌دهد همان Thread چندبار قفل را بگیرد.

```
rlock = threading.RLock()
```

۳. **`Event`** برای اطلاع‌رسانی ساده بین Threadها؛ یک Flag دارد که با `set()` و `clear()` کنترل می‌شود.

```
ready = threading.Event()

def waiter():
    print("Waiting for event...")
    ready.wait()
    print("Event is set, proceeding!")
```

```
threading.Thread(target=waiter).start()
ready.set()
```

۴. **`Condition`** شبیه به Event اما با امکان صف‌بندی منتظرها و اطلاع‌رسانی انتخابی (`notify()` / `notify_all()`).

```
cond = threading.Condition()
items = []

def producer():
    with cond:
        items.append(1)
        cond.notify()

def consumer():
    with cond:
        while not items:
            cond.wait()
        item = items.pop(0)
        print("Consumed", item)
```

۵. **`Semaphore`** برای کنترل مدل «N تا Thread همزمان می‌توانند وارد بخش بحرانی شوند».

```
sem = threading.Semaphore(3)

def access_resource(id):
    with sem:
        print(f"Thread {id} accessing resource")
        time.sleep(2)
        print(f"Thread {id} done")
```

3.4. مثال عملی: شمارش موازی

در این مثال می‌خواهیم عدد 1 تا 1_000_000 را دو نیمه کنیم و هر نیمه را با یک Thread جداگانه جمع بزنیم، سپس نتایج را با هم ترکیب کنیم:

```python
import threading

def partial_sum(start, end, result, index, lock):
    total = sum(range(start, end))
    with lock:
        result[index] = total

if __name__ == "__main__":
    lock = threading.Lock()
    result = [0, 0]

    ranges = [(1, 500_001, 0), (500_001, 1_000_001, 1)]
    threads = []

    for start, end, idx in ranges:
        t = threading.Thread(
            target=partial_sum,
            args=(start, end, result, idx, lock)
        )
        t.start()
        threads.append(t)

    for t in threads:
        t.join()

    print("Total sum:", result[0] + result[1])
```

در این کد:

- دو Thread ایجاد می‌شود که هر کدام نیمی از بازه را محاسبه می‌کند.
- با استفاده از `lock` نتیجه‌ی هر Thread در لیست مشترک `result` ثبت می‌شود.
- در نهایت پس از اتمام هر دو، مجموع کاملاً محاسبه و نمایش داده می‌شود.

مباحث Socket Programming در پایتون

4. مباحث Socket Programming در پایتون

در این بخش با نحوه‌ی برقراری ارتباط شبکه‌ای در پایتون آشنا می‌شویم. ابتدا تفاوت‌های اصلی پروتکل‌های TCP و UDP را بررسی می‌کنیم، سپس با ماژول استاندارد `socket` پل می‌زنیم و سرور و کلاینت ساده‌ای می‌سازیم. در پایان، یک مثال عملی «Echo Server» را پیاده‌سازی می‌کنیم.

4.1. مفاهیم TCP و UDP

- **TCP (Transmission Control Protocol)**
  - اتصال‌گرا (Connection-oriented): یک «ارتباط» (Connection) قبل از ارسال داده، بین سرور و کلاینت برقرار می‌شود.
  - تضمین تحویل: بسته‌ها به ترتیب و بدون خطا تحویل می‌شوند.
  - مناسب برای برنامه‌هایی که به صحت و ترتیب داده اهمیت می‌دهند (وب‌سرورها، فایل‌منتقل‌کن‌ها).
- **UDP (User Datagram Protocol)**
  - بدون اتصال (Connectionless): هر بسته (Datagram) مستقل است و پیش از ارسال نیازی به برقراری ارتباط نیست.
  - کم‌هزینه و سریع، اما تضمینی برای تحویل یا ترتیب وجود ندارد.
  - مناسب برای کاربردهای real-time شبکه‌ای مثل ویدیوکنفرانس یا بازی‌های آنلاین.

4.2. ماژول `socket`

ماژول `socket` رابط اصلی پایتون برای کار با سوکت‌های شبکه است. دو خانواده‌ی اصلی:

- `AF_INET` برای IPv4
- `AF_INET6` برای IPv6

و دو نوع اصلی سوکت:

- `SOCK_STREAM` برای TCP
- `SOCK_DGRAM` برای UDP

```python
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

4.3. ساخت یک سرور ساده (TCP)

در این مثال یک سرور TCP می‌سازیم که به یک کلاینت گوش می‌دهد و داده‌ای را که دریافت می‌کند دوباره ارسال می‌کند:

```python
import socket

HOST = '0.0.0.0'
PORT = 12345

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
    server.bind((HOST, PORT))
    server.listen()
    print(f"Server listening on {HOST}:{PORT}")

    conn, addr = server.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
    print("Connection closed.")
```

نکات کلیدی:

- `bind()` آدرس و پورت سرور را مشخص می‌کند.

- `listen()` سوکت را به حالت «گوش دادن» می‌برد.
- `accept()` تا زمان برقراری اتصال منتظر می‌ماند و یک جفت `(conn, addr)` برمی‌گرداند.
- در حلقه‌ی `recv()` ، تا زمانی که کلاینت داده ارسال نکند یا اتصال قطع شود ادامه می‌دهیم.

4.4. ساخت یک کلاینت ساده (TCP)

در کلاینت، به سرور متصل می‌شویم، پیامی ارسال می‌کنیم و پاسخ را دریافت می‌کنیم:

```
import socket

HOST = '127.0.0.1'  # Local Host
PORT = 12345

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as client:
    client.connect((HOST, PORT))
    message = "Hello, Server!"
    client.sendall(message.encode('utf-8'))
    data = client.recv(1024)
    print("Received from server:", data.decode('utf-8'))
```

نکات کلیدی:

- `connect()` برای برقراری اتصال به سرور استفاده می‌شود.
- پس از ارسال `sendall()` ، تابع `recv()` منتظر رسیدن داده می‌ماند.
- در پایان با خروج از بلوک `with` سوکت خودکار بسته می‌شود.

4.5. مثال عملی: Echo Server تک‌رشته‌ای

با ترکیب کدهای بالا، می‌توان یک سرور Echo ساخت که پیام هر کلاینت را به خود آن برمی‌گرداند:

```
# server.py
import socket

HOST = '0.0.0.0'
PORT = 5000
```

```

def run_server():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as srv:
        srv.bind((HOST, PORT))
        srv.listen()
        print(f"Echo server running on {HOST}:{PORT}")
        conn, addr = srv.accept()
        with conn:
            print(f"Connected by {addr}")
            while True:
                data = conn.recv(1024)
                if not data:
                    break
                conn.sendall(data)

if __name__ == "__main__":
    run_server()
```

```
# client.py
import socket

HOST = '127.0.0.1'
PORT = 5000

def run_client():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cli:
        cli.connect((HOST, PORT))
        while True:
            msg = input("Enter message (or 'exit'): ")
            if msg.lower() == 'exit':
                break
            cli.sendall(msg.encode('utf-8'))
            reply = cli.recv(1024).decode('utf-8')
            print("Server replied:", reply)

if __name__ == "__main__":
    run_client()
```

نحوه اجرا:

۱. در ترمینال اول:

```
python server.py
```

۲. در ترمینال دوم (یا بیشتر):

```
python client.py
```

۳. پیام بنویسید و مشاهده کنید که سرور همان پیام را برمی‌گرداند.

پروژهٔ Chatroom در پایتون

۵. پروژهٔ Chatroom آموزشی

5.1. نمای کلی پروژه

- **هدف:** سرور TCP که بتواند هم‌زمان به چند کلاینت پاسخ دهد و پیام‌های دریافتی را به همهٔ دیگر کلاینت‌ها پخش کند.
- **ابزارها:**
  - `socket` برای ارتباط شبکه‌ای
  - `threading` برای مدیریت هر کلاینت در یک Thread مجزا
  - (اختیاری) `logging` یا `loguru` برای ثبت رویدادها

5.2. تحلیل معماری

۱. **ساختار داده‌ها**

```
clients: List[socket.socket] = []
clients_lock = threading.Lock()
```

۲. **جریان ارتباط**

- سرور یک سوکت می‌سازد، bind و listen می‌کند.
- در حلقهٔ اصلی با `accept()` هر اتصال جدید را می‌پذیرد.
- هر سوکت جدید را در لیست `clients` قرار می‌دهد و یک Thread جدید به `handle_client` تخصیص می‌دهد.

۳. **مدیریت هر کلاینت (`handle_client`)**

- خواندن مداوم پیام با `recv()`
- فراخوانی `broadcast(msg, conn)` برای ارسال پیام به دیگران
- در صورت قطع اتصال یا خطا، حذف سوکت از `clients` و بستن آن

۴. **پخش پیام‌ها (`broadcast`)**

- پیمایش لیست `clients` در یک بلوک اتمیک (با قفل)
- ارسال پیام به همه جز فرستنده
- مدیریت استثناءها و حذف کلاینت‌های معیوب

5.3. ساختار دایرکتوری

```
chatroom/
├── server.py
├── client.py
└── README.md
```

5.4. `server.py` (تمپلیت)

```python
import socket
import threading

HOST = '0.0.0.0'
PORT = 5000

# shared list of all connected client sockets
clients = []
clients_lock = threading.Lock()

def handle_client(conn, addr):
    """
    - Loop: receive data = conn.recv(buffer_size)
    - If data:
        * optionally decode or prepend username
        * broadcast(data, conn)
    - On empty data or exception:
        * remove conn from clients (inside clients_lock)
        * close conn
    """
    # TODO: implement receive loop
    pass
```

```
23
24  def broadcast(message: bytes, source_conn):
25      """
26      - Acquire clients_lock
27      - For each client in clients:
28          if client is not source_conn:
29              try:
30                  client.send(message)
31              except:
32                  # on failure remove client and close socket
33                  clients.remove(client)
34                  client.close()
35      """
36      # TODO: implement broadcast logic
37      pass
38
39  def start_server():
40      """
41      1. Create a TCP socket
42      2. Bind to (HOST, PORT)
43      3. Listen for connections
44      4. Loop forever:
45           a. conn, addr = accept()
46           b. with clients_lock: add conn to clients
47           c. spawn a daemon Thread(target=handle_client, args=(conn, addr)
48      """
49      # TODO: implement server setup and accept loop
50      pass
51
52  if __name__ == '__main__':
53      start_server()
```

5.5. `client.py` (تمپلیت)

```
1  import socket
2  import threading
3
4  HOST = '127.0.0.1'
5
```

```python
PORT = 5000

def receive_loop(sock):
    """
    - Loop: msg = sock.recv(buffer_size)
    - If msg:
        * decode and print to console
    - On empty msg or exception:
        * break and exit loop
    """
    # TODO: implement receiving logic
    pass

def start_client():
    """
    1. Create TCP socket and connect to (HOST, PORT)
    2. (Optional) send username or greeting
    3. Start a daemon Thread(target=receive_loop, args=(sock,))
    4. Loop: read input from user
         a. if input is exit command: break
         b. else: sock.send(input.encode())
    5. Close socket on exit
    """
    # TODO: implement client setup, threading, and send loop
    pass

if __name__ == '__main__':
    start_client()
```

5.6. README.md (نمونه)

```
# Chatroom in Python

## Requirements
- Python 3.8+
- No external dependencies (uses stdlib `socket` & `threading`)

## Setup & Run
```

1. Start the server:
   ```bash
   python server.py

### اجرای کلاینت‌ها

برای اجرای یک یا چند کلاینت، در ترمینال دستور زیر را وارد کنید:

```
python client.py
```

سپس در هر کنسول کلاینت می‌توانید پیام تایپ کنید و ارسال نمایید.

برای قطع اتصال از دستور زیر استفاده کنید:

```
/exit
```

قابلیت‌های قابل پیاده‌سازی (TODO)

- درخواست **نام کاربری** از کاربر هنگام برقراری اتصال
- اضافه کردن دستور `/list` برای نمایش فهرست کاربران متصل
- اعمال **حداکثر طول** برای پیام‌ها
- مدیریت **خطاهای شبکه** و تلاش مجدد (retries)
- افزودن **ثبت لاگ** ساده در یک فایل

ساختار پروژه

```
chatroom/
├── server.py
├── client.py
└── README.md
```
```

ادامه پروژه Chatroom در پایتون

5.4. تکمیل `server.py`

5.4.1. پیاده‌سازی `start_server()`

در این تابع باید:

۱. سوکت TCP بسازید و آن را به آدرس و پورت موردنظر متصل کنید

۲. با `listen()` سوکت را در حالت «گوش دادن» قرار دهید

۳. در یک حلقه‌ی نامتناهی:

- با `accept()` یک اتصال جدید بپذیرید
- با قفل (`clients_lock`) سوکت را در لیست `clients` اضافه کنید
- یک ترد دیمون بسازید که `handle_client(conn, addr)` را اجرا کند

Skeleton:

```
def start_server():
    # 1. create socket
    server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # 2. bind and listen
    server_sock.bind((HOST, PORT))
    server_sock.listen()
    print(f"Server listening on {HOST}:{PORT}")

    # 3. accept loop
    while True:
        conn, addr = server_sock.accept()
        print(f"New connection: {addr}")

        # add to clients list
        with clients_lock:
            clients.append(conn)

```

```
        # start a daemon thread to handle this client
        thread = threading.Thread(
            target=handle_client,
            args=(conn, addr),
            daemon=True
        )
        thread.start()
```

> **TODO:**
>
> - در صورت بروز خطا (مثلاً پورت در دسترس نبود)، با یک بلوک `try/except` آن را هندل کنید و پیام مناسبی چاپ کنید.

5.4.2. پیاده‌سازی `handle_client(conn, addr)`

در این تابع باید:

۱. در یک حلقه‌ی `while True` :

- داده را با `conn.recv(buffer_size)` بخوانید
- اگر داده خالی (`b''`) برگشت، قطع اتصال رخ داده → `break`
- در غیر این صورت، مستقیماً `broadcast(data, conn)` را فراخوانی کنید

۲. پس از خروج از حلقه یا خطا:

- با قفل، `conn` را از `clients` حذف کنید
- سوکت را با `conn.close()` ببندید
- (اختیاری) یک پیام لاگ یا پرینت درباره‌ی قطع ارتباط چاپ کنید

Skeleton:

```
def handle_client(conn, addr):
    try:
        while True:
            data = conn.recv(1024)  # you can adjust buffer size
            if not data:
```

```
                # client disconnected
                break

            # optional: decode & prepend username, timestamp, etc.

            # broadcast to others
            broadcast(data, conn)
    except Exception as e:
        print(f"Error handling {addr}: {e}")
    finally:
        # cleanup
        with clients_lock:
            if conn in clients:
                clients.remove(conn)
        conn.close()
        print(f"Connection closed: {addr}")
```

TODO:

- اگر می‌خواهید پیام‌ها را همراه با نام کاربری یا زمان ارسال کنید، در همین تابع `decode` کنید و قالب‌بندی نمایید.
- برای تست قبل از `broadcast()` ، یک پرینت ساده از پیام دریافتی بگذارید.

5.4.3. پیاده‌سازی `broadcast(message, source_conn)`

این تابع وظیفه ارسال یک پیام بایت‌دار را به همه‌ی کلاینت‌های دیگر دارد:

۱. با قفل (`clients_lock`) به‌صورت اتمیک لیست `clients` را پیمایش کنید

۲. هر سوکت به جز `source_conn` را `send(message)` بزنید

۳. اگر حین ارسال خطا رخ داد (مثلاً اتصال قطع شده)، آن سوکت را حذف و ببندید

Skeleton:

```
def broadcast(message: bytes, source_conn):
    with clients_lock:
        for client in clients.copy():  # copy to avoid modification durir
```

```
            if client is not source_conn:
                try:
                    client.send(message)
                except Exception:
                    # remove faulty client
                    clients.remove(client)
                    client.close()
```

TODO:

- می‌توانید یک پیشنهاد کوتاه قبل از ارسال پیغام به چند کلاینت گیرنده پیام بدهید.
- اگر می‌خواهید ارسال را به شکل UDP یا پخش برودکست (در شبکه محلی) انجام دهید، اینجا تغییر دهید.

5.5. تکمیل client.py

5.5.1. پیاده‌سازی receive_loop(sock)

این تابع به‌صورت یک Thread دیمون اجرا می‌شود و پیام‌های دریافتی از سرور را نمایش می‌دهد:

```
def receive_loop(sock):
    """
    - Loop forever:
        * try to recv data = sock.recv(buffer_size)
        * if not data: server disconnected → break
        * else: decode data and print to console
    - On exception: print error (optional) and exit loop
    """
    while True:
        try:
            data = sock.recv(1024)  # buffer size
            if not data:
                # server closed connection
                print("Server disconnected.")
                break
            # decode and display
            print(data.decode('utf-8'))
```

```
        except Exception as e:
            # optional: log or print the exception
            print(f"Receive error: {e}")
            break
```

TODO:

- اگر پیام‌ها شامل نام کاربری یا `timestamp` هستند، اینجا آنها را بررسی و قالب‌بندی کنید.
- می‌توانید حداکثر زمان `timeout` برای `recv` تنظیم کنید تا همیشه منتظر نماند.

5.5.2. پیاده‌سازی `start_client()`

این تابع مسئول راه‌اندازی کلاینت، ایجاد Thread دریافت و حلقهٔ ارسال پیام است:

```
def start_client():
    """
    1. Create TCP socket and connect to (HOST, PORT)
    2. (Optional) prompt user for username and send to server
    3. Start a daemon Thread(target=receive_loop, args=(sock,))
    4. Loop:
         a. read input from user (e.g., input("> "))
         b. if input.lower() == '/exit': break
         c. otherwise send input.encode() to server
    5. Close socket on exit
    """
    # 1. create and connect
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((HOST, PORT))
    print(f"Connected to server at {HOST}:{PORT}")

    # 2. optional username
    # username = input("Enter your username: ")
    # sock.send(username.encode('utf-8'))

    # 3. start receive thread
    recv_thread = threading.Thread(
        target=receive_loop,
        args=(sock,),
```

```
        daemon=True
    )
    recv_thread.start()

    # 4. send loop
    while True:
        msg = input()
        if msg.lower() == '/exit':
            break
        # optional: enforce max length, strip whitespace, handle commands
        sock.send(msg.encode('utf-8'))

    # 5. cleanup
    sock.close()
    print("Disconnected from server.")
```

TODO:

- برای دستورات خاص (مثل `/list`)، قبل از ارسال به سرور آنها را پردازش محلی کنید یا به سرور بفرستید.
- اگر می‌خواهید قابلیت تلاش مجدد روی ارسال (`retries` یا `timeout` داشته باشید، از `socket.settimeout()` و حلقهٔ `try/except` استفاده کنید .

5.6. گام‌های بعدی و تست

۱. **اجرای سرور:** در ترمینال پروژه:

```
python server.py
```

۲. **اجرای کلاینت:** در یک یا چند ترمینال جدید:

```
python client.py
```

۳. **تست عملکرد:**

- در هر کلاینت پیام بنویسید و ببینید در سایر کلاینت‌ها نمایش داده می‌شود.
- با وارد کردن `/exit` کلاینت به‌درستی قطع شود.

5.7. توسعه‌های پیشنهادی (TODOs)

- **درخواست نام کاربری:** کلاینت یک نام کاربری بپرسد و سرور آن را ثبت و همراه پیام‌ها منتشر کند.
- **دستور `/list` :** کلاینت این دستور را به سرور ارسال کند و سرور فهرست کاربران را برگرداند.
- **محدودیت طول پیام:** قبل از ارسال، طول ورودی کاربر را بررسی و از ارسال پیام‌های خیلی طولانی جلوگیری شود.
- **مدیریت خطا و تلاش مجدد:** در تابع ارسال کلاینت و ارسال پیام broadcast در سرور، اگر خطا رخ داد چند بار تلاش کنید.
- **لاگ‌گیری:** با ماژول `logging` یا `loguru` لاگ‌های ورود/خروج و پیام‌ها را در فایل ثبت کنید.

مباحث پیشرفته (Advanced Topics)

6.1. مدیریت قطع ارتباط ناگهانی

وقتی یک کلاینت بدون ارسال `/exit` قطع اتصال می‌کند، ممکن است سوکت در لیست باقی بماند و باعث خطا شود. برای بهبود:

```
def handle_client(conn, addr):
    try:
        while True:
            data = conn.recv(1024)
            if not data:
                # TODO: detect abrupt disconnect
                break
            broadcast(data, conn)
    except ConnectionResetError:
        # TODO: log abrupt disconnect
        pass
    finally:
        # remove and close
        with clients_lock:
            if conn in clients:
                clients.remove(conn)
        conn.close()
```

- **TODO:**
  - ثبت لاگ هنگام `ConnectionResetError` یا `BrokenPipeError`.

6.2. پیام خصوصی (Private Messaging)

امکان ارسال پیام فقط به یک کاربر خاص:

۱. تعریف یک دستور در کلاینت مثل `/pm <username> <message>`

۲. ارسال پیام با فرمت ویژه به سرور

۳. در سرور، تشخیص پیام‌های خصوصی و ارسال فقط به سوکت کاربر مقصد.

```
def handle_client(conn, addr):
    # ... recv loop ...
    if message.startswith(b'/pm '):
        # TODO: parse target username and private message
        # TODO: lookup target_conn by username
        # target_conn.send(private_message)
    else:
        broadcast(message, conn)
```

6.3. لاگ‌گیری (Logging)

به‌جای استفاده از `print` برای دیباگ، از ماژول `logging` استاندارد یا `loguru` بهره ببرید:

```
import logging

logging.basicConfig(
    filename='chatroom.log',
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s'
)

logging.info(f"New connection from {addr}")
logging.error(f"Error handling {addr}: {e}")
```

- **TODO:**
  - تعیین سطوح مناسب (INFO, WARNING, ERROR).
  - دو فایل لاگ مجزا برای سرور و کلاینت.

6.4. امنیت پایه (SSL/TLS)

برای رمزنگاری داده‌ها از ماژول `ssl` استفاده کنید:

```python
import ssl

# Server side
context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
context.load_cert_chain(certfile='server.crt', keyfile='server.key')
bind_sock = socket.socket(...)
ssl_server_sock = context.wrap_socket(bind_sock, server_side=True)

# Client side
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.load_verify_locations('ca.crt')
ssl_client_sock = context.wrap_socket(
    socket.socket(),
    server_hostname=HOST
)
ssl_client_sock.connect((HOST, PORT))
```

منابع :

- threading
- socket
- ssl