گزارش پروژه‌ی نهایی شبکه‌های کامپیوتری

دکتر مقصود عباس‌پور

آرمان امینیان ندوشن - ۹۶۲۴۳۰۰۹

محمدحسین زارعی - ۹۶۲۴۳۰۳۱

فهرست

بخش اول - مقدمه و معرفی

پروژه‌ی معرفی‌شده طراحی یک شبکه‌ی P2P مشابه BitTorrent است. در این شبکه Nodeها پس از ورود می‌توانند به عنوان فرستنده ویا گیرنده عمل کنند. Nodeها با توجه به نوع خود در هنگام ورود تمامی اطلاعات لازم را از Tracker دریافت می‌کنند و نسبت به آن عملیات مورد نظر را انجام می‌دهند. اطلاعات مکمل برای پروژه در صورت‌پروژه و قسمت بعدی گزارش آمده است.

بخش دوم - نحوه‌ی عملکرد کد به صورت کلی

همان‌طور که اشاره شد، سیستم متشکل از چندین Node و یک Tracker است. در ابتدا لازم است که کد Tracker اجرا شود تا ماژول متناسب فعال شود. سپس به تعداد دلخواه از ماژول Node اجرا می‌کنیم و هر کدام از Nodeها می‌توانند به عنوان فرستنده ویا گیرنده‌ی یک فایل عمل کنند. دقت داشته باشید که کد به صورت Blocking عمل نمی‌کند و هر Node می‌تواند همزمان مسئول آپلود چند فایل ویا در حال دانلود چند فایل باشد و از آنجایی که هر عملیات مربوط به هر فایل با ایجاد یک Thread انجام می‌شود، مشکلی در این قسمت به وجود نمی‌آید. نکته‌ی دیگر این است که فایل‌های مختص به هر Node در پوشه‌ی جداگانه‌ی مربوط به همان Node قرار داده شده، مشابه دنیای واقعی.

در صورتی که Node انتخاب کند که می‌خواهد یک فایل را Upload کند، ابتدا باید به Tracker اعلام کند که فایل مورد نظر را دارد و قصد آپلود آن را دارد تا به لیست دارندگان فایل اضافه شود و سپس در یک Thread جداگانه منتظر درخواست گرفتن فایل بماند.

در صورتی که Node انتخاب کند که می‌خواهد یک فایل را از دارندگان آن Download کند، ابتدا لیست دارندگان را از Tracker دریافت کرده و سپس سایز فایل را از یکی از دارندگان می‌پرسد تا بتواند تقسیم‌بندی درست فایل را انجام دهد و بعد از آن به ازای هر دارنده یک Thread ایجاد کرده و قسمت‌های مختلف داده را از آنها می‌گیرد. در این قسمت لازم است که منتظر بمانیم تا کار تمامی Threadها تمام شود. پس از اتمام کار، قسمت‌های فایل را مرتب کرد و به یکدیگر می‌چسبانیم تا فایل دریافتی کامل شود.

در ادامه به بررسی دقیق‌تر این دو ماژول و پیام‌های بین آنها می‌پردازیم.

بخش سوم - معرفی توابع کمکی

تمامی توابع و متغیرهای کمکی در فایل utils.py قرار داده شده‌اند. در ابتدا به معرفی متغیرهای استفاده‌شده در این فایل می‌پردازیم که در جدول زیر توضیح داده شده‌اند:

متغیر	توضیحات
MAX_DATA_SIZE = 65,507	بیانگر حد داده‌ایست که در قسمت dataی یک datagram می‌توانیم قرار دهیم.
BUFFER_SIZE = 65536	میزان بافر پورت udp که برابر با ماکزیمم میزان بسته‌ی udp است.
TRACKER_ADDR = ('localhost', 12340)	آدرس واحد Tracker.
open_ports = (1024, 49151)	مقادیری که می‌توانیم به عنوان پورت به کاربر سوکت‌ها بدهیم باید در این محدوده باشد تا در سیستم مشکلی نباشد.
occupied_ports	آرایه‌ای که پورت‌هایی که تا به حال داده شده‌اند را نگه می‌دارد تا از دادن پورت تکراری جلوگیری شود.

حال به بررسی توابعی که در این فایل هستند می‌پردازیم:

تابع split_file:

این تابع وظیفه‌ی تقسیم‌بندی بایت‌های یک فایل به اندازه‌های مشخص را دارد و بایت‌های آن را در یک آرایه خروجی می‌دهد. به عنوان ورودی می‌بایست مسیر فایل، بازه‌ای از فایل که باید تقسیم کنیم و حجم بایتی هر قسمت است (که در این شبکه باید برابر با MAX_DATA_SIZE منهای یک مقدار مشخص باشد). در ابتدا بررسی می‌شود که حجم بایتی هر قسمت بیشتر از 0 باشد و سپس با باز کردن فایل، محدوده‌ای که باید تقسیم کنیم را انتخاب می‌کنیم با کمک کتابخانه‌ی mmap. پس از برداشتن محدوده با یک حلقه‌ی ساده محدوده را به حجم‌های یکسان تقسیم می‌کنیم. عملیات گفته‌شده در زیر آورده شده‌اند:

```
def split_file(path: str, rng: Tuple[int, int],
               chunk_size: int = MAX_DATA_SIZE - 2000) -> list:
    assert chunk_size > 0, print("The chunk size should be bigger than 0.")
```

```
    with open(path, "r+b") as f:
        # getting the specified range
        mm = mmap.mmap(f.fileno(), 0)[rng[0]: rng[1]]
        # diving the bytes into chunks
        ret = [mm[chunk: chunk + chunk_size] for chunk in
               range(0, rng[1] - rng[0], chunk_size)]
        return ret
```

تابع assemble_file:

این تابع وظیفه‌ی سرهم‌بندی بایت‌های مختلف فایل را دارد. در ورودی آرایه‌ی بایت‌ها را می‌دهیم و سپس با یک حلقه بایت‌ها را در فایل مورد نظر با نامی که در ورودی می‌گیریم می‌نویسیم.

```
def assemble_file(chunks: list, path: str) -> None:
    f = open(path, "bw+")
    for c in chunks:
        f.write(c)
    f.flush()
    f.close()
```

تابع give_port:

وظیفه‌ی اختصاص دادن یک عدد از پورت‌ها را دارد. البته بررسی می‌شود که پورت داده‌شده قبلاً گرفته نشده باشد.

```
def give_port() -> int:
    rand_port = randint(open_ports[0], open_ports[1])
    while rand_port in occupied_ports:
        rand_port = randint(open_ports[0], open_ports[1])
    return rand_port
```

تابع create_socket:

وظیفه‌ی ساختن یک سوکت به ازای پورت ورودی را دارد. سوکت ساخته‌شده از نوع UDP است و وقتی که ساخته می‌شود، پورت آن به لیست پورت‌های گرفته‌شده اضافه می‌شود.

```
def create_socket(port: int) -> socket.socket:
    global occupied_ports
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind(('localhost', port))
```

```
    occupied_ports.append(port)
    return s
```

تابع free_port:

وظیفه‌ی این تابع close کردن سوکت و حذف کردن شماره‌ی پورت از پورت‌های گرفته‌شده است.

```
def free_socket(s: socket.socket):
    global occupied_ports
    port = port_number(s)
    occupied_ports.remove(port)
    s.close()
```

تابع port_number:

این تابع صرفاً شماره‌ی پورت سوکت ورودی را می‌دهد.

```
def port_number(s: socket.socket) -> int:
    return s.getsockname()[1]
```

بخش چهارم - معرفی ماژول رمزنگاری

یک کلاس خیلی ساده در فایل cryptography_unit.py قرار دارد که دو تابع رمزنگاری و رمزگشائی را در خود نوشته است و وظیفه‌ی آن رمزنگاری و رمزگشائی datagramهای UDP است.

```
class CryptographyUnit:
  def __init__(self):
      if os.path.isfile("cryptography/key.key"):
          print("Found the key!")
          self.key = open("cryptography/key.key", "rb").read()
      else:
          print("Key not found! Making a new one.")
          self.key = Fernet.generate_key()
          with open("cryptography/key.key", "wb") as f:
              f.write(self.key)

  def encrypt(self, obj: UDPDatagram) -> bytes:
      f = Fernet(self.key)
      enc = f.encrypt(obj.encode())
      return enc

  def decrypt(self, data: bytes) -> UDPDatagram:
      f = Fernet(self.key)
      dec = f.decrypt(data)
      return UDPDatagram.decode(dec)
```

بخش پنجم - بررسی پیام‌های ارتباطی بین Tracker و Node

پیام‌های ارتباطی متفاوتی در شبکه ردوبدل می‌شوند که برای هر دسته در آنها یک کلاس تعیین شده است. تمامی کلاس‌ها از کلاس Message ارث‌بری می‌کنند که کلاس Message توابع encode و decode را در خود پیاده‌سازی کرده است. این کار برای این است که تمامی کلاس‌های پیام ابتدا به آرایه‌ای از بایت‌ها تبدیل شده و سپس انتقال یابند. نمی‌توانیم در datagram چیزی بجز bytes بفرستیم. در زیر تمامی کلاس‌های مربوط به هر پیام توضیح داده شده‌اند:

کلاس NodeToTracker:

این کلاس صرفاً دارای اسم فرستنده، نوع پیام و اسم فایل مورد نظر است. سه نوع پیام از Node به Tracker داریم:

- اولی برای اعلام کردن اینکه Node یک فایل را دارد در هنگام آپلود که در فایل به اسم have آورده شده است.
- دومی برای اینکه یک Node اعلام درخواست یک فایل را بکند برای دانلود که در فایل به اسم need آورده شده است.
- و سومی برای وقتی که یک Node قصد خروج از شبکه را دارد که در فایل به اسم exit است.

```
class NodeToTracker(Message):
  def __init__(self, name: str, mode: str, filename: str):
      """
      examples:
          {name, "need", filename}
          {name, "have", filename}
          {name, "exit", ""}
      """
      super().__init__()
      self.name = name
      self.mode = mode
      self.filename = filename
```

کلاس TrackerToNode:

زمانی که یک Node درخواست دانلود یک فایل را می‌دهد، باید لیست دارندگان آن فایل از Tracker به Node ارسال شود که این کلاس برای همین منظور است و دارای متغیرهای نام مقصد، لیست دارندگان فایل و اسم فایل مدنظر است.

```
class TrackerToNode(Message):
  def __init__(self, dest_name: str, owners: list, filename: str):
      """
      examples:
```

```
        {nameA, [(nameB, ipB,  portB), (nameC, ipC, portC)], filename}
    """
    super().__init__()
    self.dest_name = dest_name
    self.owners = owners
    self.filename = filename
```

کلاس SizeInformation:

فقط برای ارسال و دریافت سایز فایل مدنظر یک Node استفاده می‌شود و دارای متغیرهای نام Node مبدأ، نام Node مقصد، اسم فایل و سایز فایل است. در ضمن وقتی که یک Node می‌خواد سایز را **درخواست** کند درون سایز فایل باید مقدار 1- قرار داده شود که به صورت پیش‌فرض همین مقدار را نوشته‌ایم.

```
class SizeInformation(Message):
  def __init__(self, src_name: str, dest_name: str, filename: str,
               size: int = -1):
    """
    examples:
        size of the file:
            {src_name, dest_name, filename, size of the file}
    """
    super().__init__()
    self.src_name = src_name
    self.dest_name = dest_name
    self.filename = filename
    self.size = size
```

کلاس FileCommunication:

اصلی‌ترین کلاس پیام‌ها همین کلاس است که وظیفه‌ی نگهداری اطلاعات و بایت‌های فایل در حال تبادل را دارد. سایز encode شده‌ی این پیام حداکثر برابر با MAX_DATA_SIZE است. متغیرهای آن عبارتند از:

- نام فرستنده و گیرنده‌ی پیام.
- اسم فایل.
- محدوده‌ی فایل: برای مثال اگر یک فایل که حجمش 3 مگابایت باید بین سه نفر تقسیم شود پس هر نفر مسئولیت انتقال 1 مگابایت از فایل را دارند. این متغیر نشان‌گر اندیس‌های محدوده‌ی مخصوص هر Node فرستنده‌ی فایل است. برای مثال در نمونه‌ی گفته‌شده فرستنده‌ی اول از بایت صفرم تا بایتی که 1 مگابایت را معلوم می‌کند را می‌فرستد. در ضمن اگر

محدوده‌ی یک فایل از MAX_DATA_SIZE بیشتر باشد با اینکه چند پیام متفاوت ارسال می‌شوند اما متغیر محدوده ثابت می‌ماند.

- اندیس: اگر که محدوده‌ی ما بیشتر از MAX_DATA_SIZE باشد در نتیجه لازم است که در چند بسته‌ی متفاوت آن محدوده را ارسال کنیم. متغیر اندیس بیانگر شماره‌ی هر بسته است.
- داده: بایت‌های داده را در خودش نگه می‌دارد و حداکثر برابر با MAX_DATA_SIZE منهای یک مقدار مشخص است.

دو نوع خاص از این پیام در شبکه وجود دارند؛ یکی start-of-transfer و دیگری end-of-transfer. اولی برای نشان دادن اینکه تبادل فایل شروع شده است (درخواست فایل) پس از آن فرستنده باید فایل را بفرستند و دومی برای نمایش دادن اینکه تمامی قسمت‌های فایل توسط فرستنده فرستاده شده‌اند (اتمام فرستادن فایل) و از فرستنده به گیرنده فرستاده می‌شود. در این دو نوع خاص از پیام متغیر اندیس برابر با 1- و متغیر داده برابر با None است.

```
class FileCommunication(Message):
  def __init__(self, src_name: str, dest_name: str, filename: str,
               indices: Tuple[int, int], idx=-1, data: bytes = None):
      """
      example:
          range of the desired file:
          "data" contains the actual bytes of the data and it is None
          when a node tells another node that it needs the specified
          range of the file.
          idx means which part of the total files am I? -1 for the first.
          maximum number of bytes held in data is 64KB.
          {src_name, dest_name, filename, [0, 100,000), idx, [bytes of data]}
      """
      super().__init__()
      self.src_name = src_name
      self.dest_name = dest_name
      self.filename = filename
      self.range = indices
      self.idx = idx
      self.data = data
```

حال می‌دانیم که انواع پیام‌هایی که در شبکه منتقل می‌شوند به چه صورت است. الان به بررسی دقیق عملکرد Tracker می‌پردازیم.

بخش ششم - بررسی دقیق نحوه‌ی عملکرد Tracker

فایل tracker شامل کلاس Tracker میشود. هدف از Tracker نگهداری برخی اطلاعات از node های شبکه از قبیل ip و پورت آنها و همچنین فایل هایی که هر یک برای آپلود در اختیار شبکه می گذارند میشود که در ادامه نحوه ی نگهداری و بروز رسانی آنها را بررسی میکنیم.

ساختمان های داده (تابع init) :

- دیکشنری uploader_list : این دیکشنری بدین گونه تعریف میشود که کلید های آن نام فایل ها و value هرکدام شامل node هایی که در حال حاضر آن فایل را برای آپلود دارند و در شبکه موجود هستند، میشود.
- دیکشنری upload_freq_list : این دیکشنری بدین گونه تعریف میشود که کلید های آن نام هر node موجود در شبکه و value هرکدام نشانده ی تعداد فایلی که این node برای آپلود در حال حاضر دارد، میشود.
- سوکت tracker_s : این سوکت پیش فرض tracker است که همواره گوش به زنگ است.

مورد استفاده ی هرکدام را در ادامه خواهیم دید.

تابع send_datagram :

تابع handle_node :

این تابع به هنگام دریافت یک درخواست توسط سوکت tracker صدا زده میشود و با توجه به نوع (mode) آن یکی از توابع متناظر را صدا می زند و عملیات مربوط به آن mode را با توجه به مقادیر بسته ورودی انجام می دهد.

```
def handle_node(self, data, addr):
    dg = crypto_unit.decrypt(data)
    message = Message.decode(dg.data)
    message_mode = message['mode']
    if message_mode == modes.HAVE:
        self.add_uploader(message, addr)
    elif message_mode == modes.NEED:
        self.search_file(message, addr)
    elif message_mode == modes.EXIT:
        self.exit_uploader(message, addr)
```

تابع listen :

این تابع وظیفه گوش به زنگ بودن را دارد و به هنگام دریافت درخواست، آنرا به handle_node میدهد تا عملیات مربوطه را انجام دهد. تابع handle_node را هر بار در یک thread جدید اجرا میکنیم تا همواره آماده دریافت بسته جدید باشیم و منتظر انجام عملیات نمانیم.

```
def listen(self):
  while True:
    data, addr = self.tracker_s.recvfrom(BUFFER_SIZE)
    t = threading.Thread(target=self.handle_node, args=(data, addr))
    t.start()
```

تابع start :

وظیفه ی اجرا کردن تابع listen روی یک thread جدید را بر عهده دارد.

```
def start(self):
  t = threading.Thread(target=self.listen())
  t.daemon = True
  t.start()
  t.join()
```

تابع add_uploader :

در صورتی که mode بسته از نوع have(به معنی وضعیت آپلود یک node برای یک فایل) باشد، این تابع صدا زده می شود. در دیکشنری upload_freq_list، مقدار value مربوط به node فرستنده بسته را یکی زیاد می کنیم. دلیل این اضافه کردن این است که این node یک فایل بیشتر برای ارسال دارد و در آینده برای دریافت فایل آنرا در اولویت قرار می دهیم. همچنین نام، ip و پورت این node را به لیست فایل مربوطه در دیکشنری uploader_list اضافه می کنیم. بنابراین این node به لیست node های آماده برای آپلود این فایل اضافه میشود. این لیست را همواره به صورت set ذخیره میکنیم تا عضو تکراری نداشته باشد.

```
def add_uploader(self, message, addr):
  node_name = message['name']
  filename = message['filename']
  item = {
    'name': node_name,
    'ip': addr[0],
    'port': addr[1]
  }
  self.upload_freq_list[node_name] = self.upload_freq_list[node_name] + 1
```

```
    self.uploader_list[filename].append(json.dumps(item))
    self.uploader_list[filename] = list(set(self.uploader_list[filename]))
    self.print_db()
```

تابع search_file :

در صورتی که mode بسته از نوع need(به معنی وضعیت جست و جو یک node برای یک فایل) باشد، این تابع صدا زده می شود. برای پیدا کردن node هایی که در حال حاضر در شبکه موجود هستند و فایل مورد نظر را در برای آپلود قرار داده اند، کافیست تا روی لیست مربوط به این فایل در دیکشنری uploader_list یک حلقه بزنیم و تمامی node های داخل آن لیست را به لیست خروجی اضافه کنیم. همچنین نیاز است تا در هر بار اضافه کردن یک node، تعداد فایلی که برای آپلود گذاشته شده را از دیکشنری upload_freq_list بخوانیم و آنرا هر به همراه اطلاعات دیگر node به لیست خروجی اضافه کنیم. بنابراین لیست نهایی مان شامل نام، ip، پورت و تعداد فایل برای آپلود هر nodeای که آن فایل را برای آپلود دارد، می باشد. در نهایت لیست نهایی را به node درخواست دهنده برمی گردانیم.

```
def search_file(self, message, addr):
    node_name = message['name']
    filename = message['filename']
    self.print_search_log(node_name, filename)
    search_result = []
    for item_json in self.uploader_list[filename]:
        item = json.loads(item_json)
        upload_freq = self.upload_freq_list[item['name']]
        search_result.append(
            (item['name'], (item['ip'], item['port']), upload_freq))
    response = TrackerToNode(node_name, search_result, filename).encode()
    self.send_datagram(response, addr)
```

تابع exit_uploader :

در صورتی که node بخواهد از شبکه خارج شود، بسته ای از نوع exit برای tracker ارسال می شود تا اطلاعات خود را بروزرسانی کند.ابتدا تعداد فایل برای آپلود آن node در دیکشنری upload_freq_list را برابر با صفر می کنیم. سپس این node را از لیست آپلود تمامی فایل ها حذف می کنیم. این عملیات حذف و بروزرسانی اطلاعات در tracker به این منظور است که اطلاعات موجود در tracker همواره وضعیت فعلی شبکه را توصیف کند.

```
def exit_uploader(self, message, addr):
    node_name = message['name']
    item = {
```

```
    'name': node_name,
    'ip': addr[0],
    'port': addr[1]
}
item_json = json.dumps(item)
self.upload_freq_list[node_name] = 0
files = self.uploader_list.copy()
for file in files:
    if item_json in self.uploader_list[file]:
        self.uploader_list[file].remove(item_json)
    if len(self.uploader_list[file]) == 0:
        self.uploader_list.pop(file)
print(f"Node {message['name']} exited the network.")
self.print_db()
```

تابع print_db :

این تابع به منظور نمایش وضعیت فعلی tracker مورد استفاده قرار میگیرد.

تابع print_search_log :

این تابع به منظور نمایش log جست و جو ی node ها است.

بخش هفتم - بررسی دقیق نحوه‌ی عملکرد Node

در این بخش به بررسی دقیق فایل node.py و کلاس Node می‌پردازیم. این بخش از دو قسمت اصلی تشکیل شده است؛ یکی برای Upload کردن و دیگری برای Download. قبل از شروع توضیحات نحوه‌ی اجرای این فایل را بررسی می‌کنیم.

نحوه‌ی اجرای فایل:

برای اجرا کردن این فایل باید دو متغیر ورودی را به صورت دستی وارد کنیم:

- نام Node که با پسوند n- مشخص می‌شود.
- پورت سوکت ارسال و دریافت Node که با پسوند p- مشخص شده‌اند. دقت داشته باشید که ترتیب پورت‌ها به صورت دریافت و ارسال است. برای مثال دستور زیر گره‌ای با نام node_A را اجرا کرده و پورت‌های 11110 و 11111 را به عنوان دریافت و ارسال آن در نظر می‌گیرد: `python3 node.py -n node_A -p 11110 11111`

حلقه‌ی اصلی اجرایی کد:

پس از دریافت آرگومان‌ها ابتدا یک شئ از کلاس Node می‌سازیم و منتظر دستور ورودی کاربر می‌مانیم. دستورات معتبر این‌ها هستند:

- برای آپلود فایل: torrent -setMode upload filename
- برای دانلود فایل: torrent -setMode download filename
- برای خروج Node از برنامه: torrent exit

پس از دریافت دستور ابتدا اسم فایل جدا می‌شود و بعد از آن برای آپلود تابع set_upload، برای دانلود هم مثل آپلود اسم فایل استخراج می‌شود و یک ترد برای دانلود کردن ایجاد می‌شود که همان تابع start_download است و ورودی تابع فقط همان نام فایل است. در مورد exit هم تابع exit فراخوانی می‌شود. در ادامه ابتدا به توضیح متغیرهای کلاس پرداخته و سپس توابع را معرفی می‌کنیم.

متغیر	توضیحات
rec_s, send_s	سوکت‌های دریافت و ارسال برای Node
name	نام Node
files	فایل‌هایی که در پوشه‌ی این Node هستند.

received_files	فایل‌هایی که توسط این Node از Nodeهای دیگر دریافت شده‌اند. به صورت یک دیکشنری است که کلید آن نام فایل و مقدار آن لیستی از فرستنده‌های همان فایل است.
has_started_uploading	از آنجایی که فقط یک بار باید در مود آپلود قرار بگیریم این متغیر را استفاده می‌کنیم که اگر یک بار گوش داده بودیم، دیگر در آن مود قرار نگیریم.

حال به تعریف و بررسی توابع در فایل می‌پردازیم.

توابع کمکی اولیه:

تابع set_filenames:

در پوشه‌ی مربوط به گره می‌گردد و لیست فایل‌هایی که گره در اختیار دارد را می‌نویسد.

```
def set_filenames(self) -> list:
  path = f"node_files/{self.name}"
  ret = []
  if os.path.isdir(path):
      _, _, ret = next(os.walk(path))
  return ret
```

تابع send_datagram:

تابع اصلی ارسال بسته‌ها بدون توجه به مقصد ویا مبدأ که در ورودی پیام (که یکی از کلاس‌های گفته‌شده است) را دریافت کرده و یک datagram می‌سازد و رمزشده‌ی آن datagram را می‌فرستد از طریق سوکتی که در ورودی به آن می‌دهیم.

```
def send_datagram(self, s, msg, addr):
  dg = UDPDatagram(port_number(s), addr[1], msg.encode())
  enc = crypto_unit.encrypt(dg)
  s.sendto(enc, addr)
  return dg
```

تابع get_full_path:

مسیر اصلی هر فایل در گره‌ی فعلی را برمی‌گرداند.

```
def get_full_path(self, filename: str):
  return f"node_files/{self.name}/{filename}
```

توابع مربوط به بخش آپلود:

تابع set_upload :

اگر درخواست command ورودی از نوع have(آپلود) باشد این تابع صدا زده می شود. این تابع وظیفه دارد تا یک بسته از نوع have بسازد و آنرا برای tracker بفرستد. بدین منظر ابتدا از اینکه خود node این فایل را نداشته باشد اطمینان حاصل می کند. همچنین تنها در اولین باری که یک فایل را برای آپلود می گذارد، یک سوکت از طریق تابع start_listening می سازدو از طریق آن گوش به زنگ برای درخواست دانلود از سمت دیگر node ها می ایستد. در دفعه های بعدی دیگر نیازی به ساختن سوکت جدید نیست و از همان سوکت اولیه منتظر می ماند.

```
def set_upload(self, filename: str):
    if filename not in self.files:
        print(f"Node {self.name} does not have {filename}.")
        return
    message = NodeToTracker(self.name, modes.HAVE, filename)
    self.send_datagram(self.rec_s, message, TRACKER_ADDR)
    if self.has_started_uploading:
        print(f"Node {self.name} is already in upload mode. Not making "
              f"a new thread but the file is added to the upload list.")
        return
    else:
        print(f"Node {self.name} is now listening for download requests.")
        self.has_started_uploading = True
    # start listening for requests in a thread.
    t = Thread(target=self.start_listening, args=())
    t.setDaemon(True)
    t.start()
```

تابع start_listening :

در این تابع روی سوکت rec_s منتظر درخواست از دیگر node ها می ماند و در صورتی که درخواست رسیده شده از نوع size باشد، تابع tell_file_size زا صدا زده و در صورتی که از نوع range باشد، تابع send_file را صدا می زنیم. جزئیات این ۲تابع را در ادامه خواهیم دید.

```
def start_listening(self):
    while True:
        data, addr = self.rec_s.recvfrom(BUFFER_SIZE)
        dg: UDPDatagram = UDPDatagram.decode(data)
```

```
        msg = Message.decode(dg.data)
        if "size" in msg.keys() and msg["size"] == -1:
            # meaning someone needs the file size
            self.tell_file_size(dg, msg)
        elif "range" in msg.keys() and msg["data"] is None:
            print(f"Node {self.name} received the start-of-transfer "
                  f"message from Node {msg['src_name']}.")
            self.send_file(msg["filename"], msg["range"], msg["src_name"],
                           dg.src_port)
```

تابع tell_file_size :

هنگامی که دیگر node ها برای دانلود یک فایل نیاز به size آن دانرد و آن درخواست را به ما node فعلی بزنند، این تابع فراخوانی می شود. از کلاس SizeInformation که پیش تر توضیح داده شد به منظور فهمیدن size فایل درخواستی استفاده می کنیم و مقدار آنرا به node درخواست دهنده برمی گردانیم.

```
def tell_file_size(self, dg: UDPDatagram, msg: dict):
    filename = msg["filename"]
    size = os.stat(self.get_full_path(filename)).st_size
    resp_message = SizeInformation(self.name, msg["src_name"],
                                   filename, size)
    temp_s = create_socket(give_port())
    self.send_datagram(temp_s, resp_message, ('localhost', dg.src_port))
    print(f"Sending the {filename}'s size to {msg['src_name']}.")
    free_socket(temp_s)
```

تابع send_file:

ابتدا فایل مورد نظر را می‌خوانیم و قسمت‌های متفاوت آن را با استفاده از تابع split_file درست می‌کنیم و به ازای هر قسمت، یک پیام از جنس FileCommunication می‌سازیم و مقدار داده و اندیس را در آن قرار داده و ارسال می‌کنیم. در آخر هم برای نشان دادن اتمام کار یک پیام از همان جنس قبلی که اندیس آن برابر با 1- است می‌فرستیم.

```
def send_file(self, filename: str, rng: Tuple[int, int], dest_name: str,
              dest_port: int):
    path = self.get_full_path(filename)
    parts = split_file(path, rng)
    temp_s = create_socket(give_port())
    for i, part in enumerate(parts):
        msg = FileCommunication(self.name, dest_name, filename, rng, i,
```

```
                        part)
        self.send_datagram(temp_s, msg, ("localhost", dest_port))
    msg = FileCommunication(self.name, dest_name, filename, rng)
    self.send_datagram(temp_s, msg, ("localhost", dest_port))
    print(f"Node {self.name} has sent the end-of-transfer message "
          f"to {dest_name}.")
    free_socket(temp_s)
```

توابع مربوط به بخش دانلود:

تابع start_download:

در ورودی تابع فقط اسم فایل مدنظر داده می‌شود. ابتدا بررسی می‌شود که فایل در مسیر فایل‌های گره قرار نداشته باشد که اگر قرار داشته باشد، عملیات دانلود کنسل شده و از کاربر درخواست می‌شود که اسم فایل را تغییر دهد. سپس با استفاده از تابع search که در ادامه توضیح داده خواهد شد لیست دارندگان فایل دریافت شده و بعد از آن، در تابع split_owners عملیات اصلی دانلود شروع می‌شود.

```
def start_download(self, filename: str):
    if os.path.isfile(self.get_full_path(filename)):
        print(f"{filename} already exists in Node {self.name}'s "
              f"directory. Please rename the existing file and try again.")
        return
    print(f"Node {self.name} is starting to download {filename}.")
    res = self.search(filename)
    owners = res["owners"]
    self.split_owners(filename, owners)
```

تابع search:

برای اینکه دارندگان فایل را دریافت کنیم ابتدا پیامی با نوع need از سوکت ارسال به Tracker می‌دهیم و پس از آن بر روی یک سوکت random منتظر ارسال لیست توسط Tracker می‌مانیم. بعد از دریافت لیست، آن را رمزگشائی کرده و پیام را برمی‌گردانیم.

```
def search(self, filename: str) -> dict:
    message = NodeToTracker(self.name, modes.NEED, filename)
    temp_s = create_socket(give_port())
    self.send_datagram(temp_s, message, TRACKER_ADDR)
    while True:
        data, addr = temp_s.recvfrom(BUFFER_SIZE)
        dg: UDPDatagram = crypto_unit.decrypt(data)
```

```
    if dg.src_port != TRACKER_ADDR[1]:
        raise ValueError(f"Someone other than the tracker with "
                         f"port:{dg.src_port} sent {self.name} "
                         f"the search datagram.")
    return Message.decode(dg.data)
```

تابع split_owners:

کار اصلی بخش دانلود در این تابع انجام می‌شود. ابتدا با مرتب‌سازی لیست دارندگان فایل تعداد مشخصی از کسانی که بیشترین فایل‌ها را ارسال کرده‌اند انتخاب کرده (با متغیر SELECT_COUNT) و آن لیست را به عنوان لیست اصلی owners نگه می‌داریم. اگر که ownerی پیدا نشد، به کاربر پیغام می‌دهیم.

```
owners = [o for o in owners if o[0] != self.name]
owners = sorted(owners, key=lambda x: x[2], reverse=True)
owners = owners[:SELECT_COUNT]
if not owners:
  print(f"Could not find any owner of {filename} for "
        f"Node {self.name}.")
  return
print(f"The top {SELECT_COUNT} owner(s) of {filename} are:\n{owners}")
```

سپس باید سایز فایل را از یکی از دارندگان بپرسیم که با تابع ask_file_size انجام می‌شود.

```
print(f"Asking the {filename}'s size from {owners[0][0]}.")
size = self.ask_file_size(filename, owners[0])
print(f"The size of {filename} is {size}.")
```

بعد از آن با استفاده از تابع split_size سایز فایل را به تعداد دارندگان فایل تقسیم کرده و در یک متغیر نگه می‌داریم.

```
ranges = self.split_size(size, len(owners))
print(f"Each owner now sends {round(size / len(owners), 0)} bytes of "
    f"the {filename}.")
```

حال باید به ازای هر دارنده‌ی فایل یک Thread بسازیم که وظیفه‌ی آن Thread دریافت فایل است. همچنین در متغیر received_files یک سطر جدید برای فایل فعلی درست می‌کنیم.

```
threads = []
self.received_files[filename] = []
print(f"Node {self.name} is making threads to receive the parts "
    f"from the owners.")
for i, o in enumerate(owners):
  t = Thread(target=self.receive_file,
             args=(filename, ranges[i], o))
  t.setDaemon(True)
```

```
    t.start()
    threads.append(t)
```

که بر روی هر ترد تابع receive_file را صدا می‌زنیم.

پس از join کردن روی تمامی Threadها حال ما تمام قسمت‌های فایل مورد نظر را داریم و نوبت آن است که قسمت‌ها را مرتب کرده (با تابع sort_received_files) و آنها را در یک آرایه قرار داده و تابع assemble_file را صدا بزنیم تا فایل را بسازد و ذخیره کند.

```
for t in threads:
    t.join()
print(f"Node {self.name} has received all the parts of {filename}. "
      f"Now going to sort them based on ranges.")
ordered_parts = self.sort_received_files(filename)
print(f"All the parts of {filename} are now sorted.")
whole_file = []
for section in ordered_parts:
    for part in section:
        whole_file.append(part["data"])
assemble_file(whole_file, self.get_full_path(filename))
print(f"{filename} is successfully saved for Node {self.name}.")
```

تابع ask_file_size:

ابتدا یک پیام از نوع SizeInformation می‌سازیم که سایز آن برابر با 1- است. و سپس با استفاده از یک سوکت رندوم پیام را فرستاده و منتظر دریافت پیام حاوی سایز می‌مانیم.

```
def ask_file_size(self, filename: str, owner: tuple) -> int:
    message = SizeInformation(self.name, owner[0], filename)
    temp_s = create_socket(give_port())
    self.send_datagram(temp_s, message, owner[1])
    while True:
        data, addr = temp_s.recvfrom(BUFFER_SIZE)
        dg: UDPDatagram = crypto_unit.decrypt(data)
        free_socket(temp_s)
        return Message.decode(dg.data)["size"]
```

تابع split_size:

با دریافت سایز و تعداد دارندگان صرفاً محدوده‌ها را مشخص کرده و برمی‌گرداند.

```
def split_size(size: int, num_parts: int):
    step = size / num_parts
```

```
    return [(round(step * i), round(step * (i + 1))) for i in
            range(num_parts)]
```

تابع receive_file:

ابتدا یک بسته‌ی FileCommunication با اندیس برابر با 1- و داده‌ی برابر با None ساخته و به عنوان start-of-transfer به گره مقصد می‌فرستیم روی یک سوکتی که رندوم ساختیم منتظر دریافت بسته‌های فایل می‌مانیم. هر بسته را decode کرده و به بسته‌های فایل مورد نظر در received_file اضافه می‌کنیم.

```
msg = FileCommunication(self.name, owner[0], filename, rng)
temp_s = create_socket(give_port())
self.send_datagram(temp_s, msg, owner[1])
print(f"Node {self.name} has sent the start-of-transfer message to "
      f"{owner[0]}.")
while True:
    data, addr = temp_s.recvfrom(BUFFER_SIZE)
    dg: UDPDatagram = crypto_unit.decrypt(data)
    msg = Message.decode(dg.data)
    if msg["filename"] != filename:
        print(f"Wanted {filename} but received {msg['range']} range "
              f"of {msg['filename']}")
        return
    if msg["idx"] == -1:
        print(f"Node {self.name} received the end-of-transfer message "
              f"from {owner[0]}.")
        free_socket(temp_s)
        return
    self.received_files[filename].append(msg)
```

تابع sort_received_files:

صرفاً قرار است که دیکشنری received_files را مرتب کند. ابتدا باید نسبت به محدوده‌ها این کار را انجام دهد و سپس بر اساس محدوده‌ها آن‌ها را group کرده و هر group را براساس اندیس مرتب کند.

```
def sort_received_files(self, filename: str):
    sort_by_range = sorted(self.received_files[filename],
                           key=itemgetter('range'))
    group_by_range = groupby(sort_by_range, key=lambda x: x["range"])
    res = []
    for k, v in group_by_range:
```

```
        vl_srt_by_idx = sorted(list(v), key=itemgetter('idx'))
        res.append(vl_srt_by_idx)
    return res
```

تابع exit :

این تابع وظیفه دارد که یک بسته از نوع exit بسازد و برای tracker بفرستد تا tracker اطلاعات خود را بروزرسانی کند.

```
def exit(self):
    print(f"Node {self.name} exited the program.")
    msg = NodeToTracker(self.name, modes.EXIT, '')
    self.send_datagram(self.rec_s, msg, TRACKER_ADDR)
    free_socket(self.rec_s)
    free_socket(self.send_s)
```

بخش هفتم - نمونه‌ی اجرای کد

در روند زیر برای اجرای برنامه ابتدا Tracker را اجرا می‌کنیم و گره‌های B و C را به عنوان دارندگان file2 معرفی می‌کنیم و از گره A درخواست دانلود فایل file2 را می‌دهیم.

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/tracker.py"
Found the key!

************************ Current Database ************************
* Upload frequency list:
defaultdict(<class 'int'>,
            {'node_B': 1})

* Files' uploader list:
defaultdict(<class 'list'>,
            {'file2': ['{"name": "node_B", "ip": "127.0.0.1", "port": 22220}']})
******************************************************************

************************ Current Database ************************
* Upload frequency list:
defaultdict(<class 'int'>,
            {'node_B': 1,
             'node_C': 1})

* Files' uploader list:
defaultdict(<class 'list'>,
            {'file2': ['{"name": "node_B", "ip": "127.0.0.1", "port": 22220}',
                       '{"name": "node_C", "ip": "127.0.0.1", "port": 33330}']})
******************************************************************
```

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/node.py" -n node_B -p 22220 22221
Found the key!
torrent -setMode upload file2
Node node_B is now listening for download requests.
```

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/node.py" -n node_C -p 33330 33331
Found the key!
torrent -setMode upload file2
Node node_C is now listening for download requests.
```

و وقتی که دستور دانلود را می‌زنیم اتفاقات زیر در خروجی خواهند افتاد:

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/node.py" -n node_A -p 11110 11111
Found the key!
torrent -setMode download file2
Node node_A is starting to download file2.
The top 2 owner(s) of file2 are:
[('node_B', ('127.0.0.1', 22220), 1), ('node_C', ('127.0.0.1', 33330), 1)]
Asking the file2's size from node_B.
Port 25306's socket is now closed.
The size of file2 is 283444.
Each owner now sends 141722.0 bytes of the file2.
Node node_A is making threads to receive the parts from the owners.
Node node_A has sent the start-of-transfer message to node_B.Node node_A has sent the start-of-transfer message to node_C.

Node node_A received the end-of-transfer message from node_C.
Port 40276's socket is now closed.
Node node_A received the end-of-transfer message from node_B.
Port 4741's socket is now closed.
Node node_A has received all the parts of file2. Now going to sort them based on ranges.
All the parts of file2 are now sorted.
file2 is successfully saved for Node node_A.
```

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/tracker.py"
Found the key!

************************ Current Database ************************
* Upload frequency list:
defaultdict(<class 'int'>,
            {'node_B': 1})

* Files' uploader list:
defaultdict(<class 'list'>,
            {'file2': ['{"name": "node_B", "ip": "127.0.0.1", "port": 22220}']})
******************************************************************

************************ Current Database ************************
* Upload frequency list:
defaultdict(<class 'int'>,
            {'node_B': 1,
             'node_C': 1})

* Files' uploader list:
defaultdict(<class 'list'>,
            {'file2': ['{"name": "node_B", "ip": "127.0.0.1", "port": 22220}',
                       '{"name": "node_C", "ip": "127.0.0.1", "port": 33330}']})
******************************************************************

************************ Search Log ************************
node_A is searching for file2...
************************************************************
```

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/node.py" -n node_B -p 22220 22221
Found the key!
torrent -setMode upload file2
Node node_B is now listening for download requests.
Sending the file2's size to node_A.
Port 29478's socket is now closed.
Node node_B received the start-of-transfer message from Node node_A.
Node node_B has sent the end-of-transfer message to node_A.
Port 1974's socket is now closed.
```

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/node.py" -n node_C -p 33330 33331
Found the key!
torrent -setMode upload file2
Node node_C is now listening for download requests.
Node node_C received the start-of-transfer message from Node node_A.
Node node_C has sent the end-of-transfer message to node_A.
Port 33817's socket is now closed.
```

اگر هم گره‌ای بخواد خارج شود تغییرات زیر در خروجی خود گره و Tracker رخ می‌دهد:

```
/usr/bin/python3.8 "/home/mh/Documents/Computer Networks/computer-networks/project/node.py" -n node_B -p 22220 22221
Found the key!
torrent -setMode upload file2
Node node_B is now listening for download requests.
Sending the file2's size to node_A.
Port 29478's socket is now closed.
Node node_B received the start-of-transfer message from Node node_A.
Node node_B has sent the end-of-transfer message to node_A.
Port 1974's socket is now closed.
torrent exit
Node node_B exited the program.
Port 22220's socket is now closed.
Port 22221's socket is now closed.

Process finished with exit code 0
```

```
Node node_B exited the network.

************************ Current Database ************************
* Upload frequency list:
defaultdict(<class 'int'>,
            {'node_B': 0,
             'node_C': 1})

* Files' uploader list:
defaultdict(<class 'list'>,
            {'file2': ['{"name": "node_C", "ip": "127.0.0.1", "port": 33330}']})
******************************************************************
```