EMAIL: simone.redighieri@studio.unibo.it

MATRICOLA: 0000989873

TRACCIA 2

ARCHITETTURA CLIENT-SERVER UDP PER TRASFERIMENTO FILE

SCOPO DEL PROGETTO:

Lo scopo del progetto è quello di realizzare un'applicazione Client-Server per il trasferimento di file che impieghi il servizio di rete senza connessione, ovvero UDP come protocollo di strato di trasporto.

Operazioni:

- Visualizzazione sul client dei file disponibili sul server
- Il download di un file dal server
- L'upload di un file sul server

DESCRIZIONE DELL'ARCHITETTURA DEL SISTEMA:

L'architettura utilizzata è quella del Client-Server implementata attraverso il protocollo UDP (User Datagram Protocol), un servizio di trasporto datagram-oriented (non affidabile) e non orientato alla connessione.

Un sistema Client-Server, come suggerito dal nome, è un'architettura costituita da due moduli distinti:

- Il client, che invia un messaggio di richiesta verso il server, al quale richiede un determinato servizio
- Il server, che riceve il messaggio di richiesta e risponde attraverso l'indirizzo e la porta del client

SERVER

Il server è un componente informatico di elaborazione e gestione del traffico di informazioni che fornisce servizi ad altre componenti, tipicamente clients, che ne fanno richiesta attraverso una rete.

Il processo server è ospitato su un computer, detto host, che è in ascolto su una determinata porta in attesa di ricevere messaggi. Ogni servizio offerto è accessibile dai client mediante la connessione al socket del server, un'interfaccia progettata per la trasmissione e la ricezione di dati attraverso una rete ed è identificata univocamente dall'indirizzo IP dell'host e dalla porta.

- **Funzionalità del Server:**

- Inviare il messaggio di risposta al comando ' list ' al client richiedente
- Inviare il messaggio di risposta contenente la file list, ovvero la lista dei nomi dei file disponibili per la condivisione
- Inviare il messaggio di risposta al comando ' get ' contenente il file richiesto o un opportuno messaggio d'errore se questo non è presente
- Ricevere un messaggio ' put ' contenente il file da caricare sul server e inviare un messaggio di risposta con l'esito

- **Implementazione del Server:**

Il server è capace di gestire più client contemporaneamente ed è composto da due moduli: UDPServer.py e UDPServerMultiClient.py

La classe UDPServer è estesa dalla classe UDPServerMultiClient, la quale effettua override del metodo *wait_for_client()* al fine di non far chiudere il socket dopo aver gestito la richiesta di un solo client ma di rimanere aperto utilizzando un loop infinito per poter gestire le richieste di più client attraverso l'utilizzo dei threads.

UDPServer

- Crea la classe UDPServer e assegna l'indirizzo IP e la porta al server

```
class UDPServer:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        self.sock = None
```

- *configure_server()* configura il server creando il socket UDP e lo assegna all'indirizzo IP e alla porta assegnatagli in precedenza

```
def configure_server(self):
    self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    print('\n\r starting up on %s port %s' % (self.host, self.port))
    self.sock.bind((self.host, self.port))
```

- *list_files()* scorre in tutta la directory corrente i file disponibili (eccetto UDPServer.py e UDPServerMultiClient.py) e aggiorna di volta in volta la *filelist*. Infine manda la *filelist* al client e ritorna il numero di byte inviati

```
def list_files(self, address):
    filelist = ''
    for filename in os.listdir(os.getcwd()):
        if filename != 'UDPserver.py' and filename != 'UDPServerMultiClient.py':
            filelist = filelist + filename + '\n'
    sent = self.sock.sendto(filelist.encode(), address)
    return sent
```

- *get_file(file, address)*:
  - Controlla che il file richiesto esista (*os.path.isfile()*)
  - Se esiste, lo comunica al client e legge 4096 byte (BUFF_SIZE) alla volta mandandoli al client di volta in volta
  - Se non esiste, manda un messaggio d'errore al client

```
def get_file(self, file, address):
    if os.path.isfile(file):
        sent = self.sock.sendto('File exist'.encode(), address)
        f = open(file, 'rb')
        bytes_read = f.read(BUFF_SIZE)
        while (bytes_read):
            check = self.sock.sendto(bytes_read, address)
            if(check):
                sent = sent + check
                bytes_read = f.read(BUFF_SIZE)
        f.close()
    else:
        sent = self.sock.sendto('\nError: file not exist!\n'.encode(), address)
    return sent
```

- *put_file(file)*:
  - Aspetta di ricevere un messaggio dal client per verificare che il file passatogli esista
  - Se esiste crea un file con lo stesso nome (o apre il file se questo è già presente), aspetta di ricevere i dati del file dal client e li scrive man mano all'interno del nuovo file. Alla fine invia un messaggio al client comunicando che il file è stato caricato con successo.

```
def put_file(self, file):
        accept, address = self.sock.recvfrom(BUFF_SIZE)
        if accept.decode('utf8') == 'Exist':
            f = open(file, 'wb')
            while True:
                read = select.select([self.sock], [], [], 0.1)
                if read[0]:
                    bytes_read, address = self.sock.recvfrom(BUFF_SIZE)
                    f.write(bytes_read)
                else:
                    f.close()
                    response = '\nfile uploaded successfully!\n'
                    break
        else:
            response = '\nError: impossible to upload the file!\n'
        sent = self.sock.sendto(response.encode(), address)
        return sent
```

(Viene utilizzato il metodo select.select() dal modulo select per determinare quando il server è pronto a ricevere i dati)

- Se non esiste manda un messaggio d'errore al client
- Infine ritorna il numero di byte inviati

- *handle_request(data, address)* gestisce la richiesta del client richiamando i metodi descritti in precedenza oppure, se il servizio richiesto non è disponibile, manda un messaggio d'errore al client

```
def handle_request(self, data, address):
    message = data.decode('utf8')
    print('\n\r received "%s" from %s' % (message, address))
    if message[0:4] == 'list':
        sent = self.list_files(address)
    elif message[0:3] == 'get':
        sent = self.get_file(message[4:], address)
    elif message[0:3] == 'put':
        sent = self.put_file(message[4:])
    else:
        sent = self.sock.sendto('\nError: command not available!\n'.encode(), address)

    print('sent %s bytes back to %s' % (sent, address))
```

- *wait_for_client()* aspetta di ricevere un messaggio da un client e richiama il metodo *handle_request()* (*try…except* utilizzato per gestire le eccezioni che si possono verificare)

```
def wait_for_client(self):
    try:
        print('\n\r waiting to receive messagge...')
        data, address = self.sock.recvfrom(BUFF_SIZE)
        self.handle_request(data, address)
    except Exception as err:
        print(err)
```

- *shutdown_server()* chiude il server socket

```
def shutdown_server(self):
    print('\nShutting down server...')
    self.sock.close()
```

UDPServerMultiClient

- Crea la classe UDPServerMultiClient richiamando il costruttore della superclasse

```
class UDPServerMultiClient(UDPServer.UDPServer):
    def __init__(self, host, port):
        super().__init__(host, port)
```

- *(Override) wait_for_client()* gestisce più client contemporaneamente usando un loop infinito per mantenere aperto il socket
  - Fa partire un loop infinito
  - Aspetta finché non riceve un messaggio da un client

```
def wait_for_client(self):
    try:
        while True:
            try:
                print('\n\r waiting to receive messagge...')
                data, address = self.sock.recvfrom(BUFF_SIZE)

                thread = threading.Thread(target = self.handle_request,
                                          args= (data, address))
                thread.daemon = True
                thread.start()

                if data.decode('utf8')[0:3] == 'put':
                    time.sleep(1)

            except OSError as err:
                print(err)
    except KeyboardInterrupt:
        self.shutdown_server()
```

- Crea un thread per ogni client ed esegue il metodo *handle_request()* della superclasse
- *try…*except più annidato gestisce le eccezioni generate riguardanti il sistema operativo, quello più esterno gestisce le eccezioni generate dall'utente e richiama il metodo della superclasse *shutdown_server()*

CLIENT

Un client è un componente che accede ai servizi di un server per effettuare determinate operazioni. Il processo client, ospitato su un host, deve essere eseguito soltanto dopo il processo server. In un secondo momento, viene mostrato un elenco di operazioni rese disponibili dal server che permetteranno al client di scegliere l'opzione desiderata. Il messaggio contenente il servizio richiesto dal client verrà inviato al server attraverso il socket.

- **Funzionalità del Client**

- Inviare il messaggio ' list ' per richiedere la lista dei nomi dei file disponibili
- Inviare il messaggio ' get ' per ottenere un file
- Ricevere un file richiesto tramite il messaggio di ' get ' o un eventuale messaggio d'errore
- Inviare il messaggio ' put ' per effettuare l'upload di un file sul server e ricevere il messaggio di risposta con l'esito dell'operazione

Implementazione del Client

- Crea la classe UDPClient e il socket UDP del client con cui interagisce con il server

```
class UDPClient:
    def __init__(self):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

- *get_file(file, server_address)*:
  - Il client attende di ricevere una risposta dal server che gli dica se il file richiesto esiste
  - Se esiste crea un nuovo file con lo stesso nome (o apre il file se questo già esiste), aspetta di ricevere i dati del file dal server e li scrive all'interno del nuovo file.
    (È stato utilizzato il metodo select.select() dal modulo select, come nel server, per controllare quando il client è pronto a ricevere i dati)
  - Se invece non esiste stampa il messaggio d'errore inviatogli dal server

```
def get_file(self, file, server_address):
    response, serve_address = self.sock.recvfrom(BUFF_SIZE)
    if response.decode('utf8') == 'File exist':
        print('\nwaiting to receive the file...\n')
        f = open(file, 'wb')
        while True:
            read = select.select([self.sock], [], [], 0.1)
            if read[0]:
                bytes_read, address = self.sock.recvfrom(BUFF_SIZE)
                f.write(bytes_read)
            else:
                f.close()
                break
        print('file received successfully!\n')
    else:
        print(response.decode('utf8'))
```

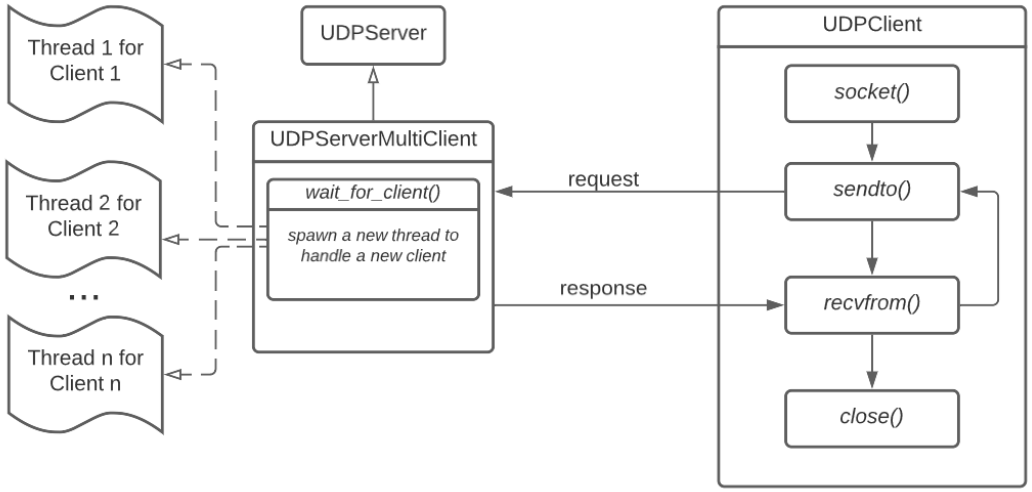
- *put_file(file, server_address)*:
  - Verifica, in primo luogo, se il file da caricare sul server esiste
  - Se esiste invia un messaggio al server dicendogli che il file esiste così si prepara a riceverlo. Successivamente legge il file a gruppi di 4096 byte (BUFF_SIZE) e li manda al server di volta in volta
  - Se non esiste manda un messaggio al server dicendogli che il file non esiste
  - Infine, aspetta di ricevere l'esito dell'operazione

```
def put_file(self, file, server_address):
    if os.path.isfile(file):
        self.sock.sendto('Exist'.encode(), server_address)
        print('\r\nwaiting to send the file...')
        f = open(file, 'rb')
        bytes_read = f.read(BUFF_SIZE)
        while (bytes_read):
            if (self.sock.sendto(bytes_read, server_address)):
                bytes_read = f.read(BUFF_SIZE)
        f.close()
    else:
        self.sock.sendto('File not exist'.encode(), server_address)
    data, server = self.sock.recvfrom(BUFF_SIZE)
    print(data.decode('utf8'))
```

```python
def interact_with_server(self, server_address):
    try:
        message = input(options)

        print('\nsent: %s' % message)
        self.sock.sendto(message.encode(), server_address)
        command = message[0:4]

        if command == 'get ':
            self.get_file(message[4:], server_address)
        elif command == 'put ':
            self.put_file(message[4:], server_address)
        else:
            data, server = self.sock.recvfrom(BUFF_SIZE)
            print('\r\nReceved:\n %s' % data.decode('utf8'))

    except OSError as err:
        print(err)
    finally:
        print('closing socket')
        self.sock.close()
```

- *interact_with_server(server_address)* mostra prima di tutto un elenco dei servizi messi a disposizione dal server e salva nella variabile *message* il servizio che si vuole richiedere e lo invia al server. Se il servizio richiesto è un *get* o *put* richiama i metodi descritti in precedenza, altrimenti aspetta di ricevere un messaggio dal server (che può essere l'elenco dei file disponibili se il servizio richiesto è un *list* oppure un messaggio d'errore se si richiede un servizio non disponibile).
- *Try…except…finally* gestisce le eccezioni generate dal sistema operativo e chiude il client socket

SCHEMA GENERALE DEI THREADS



Il processo server rimane sempre in esecuzione per poter offrire i servizi a tutti i client che li richiedono. Per ogni client che effettua una richiesta (possono effettuarla anche contemporaneamente) il server crea un nuovo thread che si occuperà di gestire la richiesta del client e si rimette in attesa di ricevere una nuova richiesta da un altro client.

ESEGUIRE IL CODICE

Server -> >> python UDPServerMultiClient.py

Client -> >> python UDPClient.py