به نام کیمیاگر عالم

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

سیستم های توزیع شده

عنوان

تمرین اول – RMI

مدرس

دکتر امیر کلباسی

دانشجو

امیرحسین بابائیان

۴۰۱۱۳۱۰۰۲

ترم پاییز ۰۲-۰۱

گروه معماری کامپیوتر و شبکه های کامپیوتری

دانشکده مهندسی کامپیوتر، دانشگاه صنعتی امیرکبیر (پلی تکنیک تهران)

فهرست

شرح مسئله توضیحات اولیه

در این مسئله قصد داریم تا کتابخانه ای کوچک برای انجام امور مربوط به RPC یا همان فراخوان از راه دور پیاده سازی کنیم و سپس از کتابخانه پیاده سازی شده که ما نام rpcClass را برایش برگزیدیم استفاده نموده و سرور و کلاینتی پیاده سازی کرده و از آن استفاده نماییم.

بخش مربوط به قابلیت ارسال کلاس برای محاسبات که بخشی از خواسته ی این مسئله بود نیز پیاده سازی شده است، بدین معنی که برای فراخوانی از راه دور، نه صرفا محتوای لازم که یک کلاس را به صورت کامل ارسال می کنیم و آن کلاس در سرور ساخته شده و فراخوانی های مربوطه اجرا شده و سپس حاصل بر می گردد.

برای پیاده سازی این پروژه مفاهیم مربوط به RMI در زبان برنامه نویسی جاوا مورد بررسی دقیق تری قرار گرفت و موارد مشابه با استفاده از کتابخانه های موجود جهت ایجاد دید بهتر آزمایش شد و سپس با ایده ای مشابه کتابخانه های موجود و مورد اقبال عموم پیاده سازی مربوط به کتابخانه rpcClass در زبان python انجام شد.

در این پروژه از Visual Studio Code 1.57.1 به عنوان ویرایشگر کد و محیط توسعه استفاده است و از git به صورت local به عنوان ورژن کنترل بهره گرفته ایم.

این گزارش نیز با استفاده از Microsoft Office Word 2019 نوشته شده است.

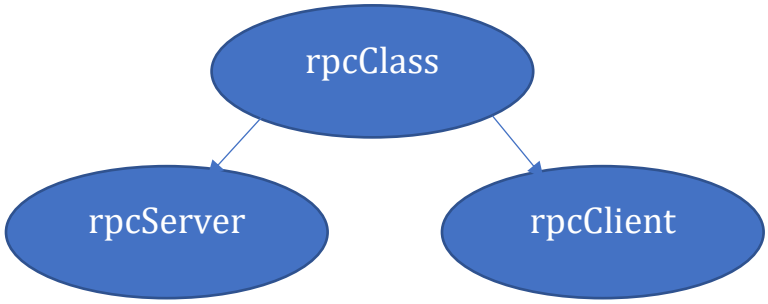
پیاده سازی کلاس rpcClass

ماژول rpcClass.py به عنوان فایل اصلی کتابخانه پیاده سازی شده است که برای استفاده از کتابخانه نیاز است تا این فایل را در کنار کد خود داشته باشیم، این کتابخانه قابلیت استفاده از امکان فرخوانی از راه دور را به ما میدهد که در فرایند توضیح کلاس های داخل این فایل به جزئیات مربوط به آن می پردازیم.

```
import socket, json, time, array, typing
```

در این فایل از کتابخانه های socket، json، time، array و typing استفاده شده است.

فرم کلی کلاس های داخل این فایل به شرح ذیل می باشد.



کلاس rpcClass

کلاس اصلی این کتابخانه که دو کلاس دیگر rpcClient و rpcServer از آن ارث بری میکنند.

```
class rpcClass(object):
```

این کلاس از object ارث بری میکند تا ویژگی های object را داشته باشد.

```
    #constructor
    def __init__(self, mode, headSize, buferSize):
        self.rpcSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.mode = mode
        self.headsize = headSize
        self.buferSize = buferSize
        self.CLOSE_CONNECTION = -1
        self.CLOSE_REMOTE = -2
```

متد سازنده ی این کلاس ابتدا یک سوکت به صورت AF_INET به صورت TCP میسازد، سپس ویژگی های مختلفی که در فرایند هایی همچون اتصال و قطع آن و دریافت و ارسال داده کاربردی است را مقدار دهی میکند، بخشی از مقدار دهی بر اساس ورودی های سازنده انجام می شود.

```
    def dataToByte(self, data) -> bytes:
        if (self.mode is None):
            return data

        if (self.mode == 'STR'):
            return data.encode()

        if (self.mode == 'JSON'):
            return bytes(json.dumps(data), 'utf-8')

        if (self.mode == "ARRAY"):
            return bytes(data)
```

متد dataToBytes یکی از متدهای اصلی در فرایند انتقال اطلاعات است که وظیفه تبدیل داده های ورودی به بایت را به عهده دارد، این متد بر اساس نوع mode که می تواند سه حالت STR، JSON و یا ARRAY باشد محتوای ورودی را به بایت تبدیل می کند.

```
    def byteToData(self, byteStream: bytes):
        if (self.mode is None):
            return byteStream

        if (self.mode == 'STR'):
            return byteStream.decode('utf-8')

        if (self.mode == 'JSON'):
            return json.loads(byteStream.decode('utf-8'))

        if (self.mode == "ARRAY"):
            d = array.array('d')
            d.frombytes(byteStream)
            return d
```

متد byteToData دیگر متدی است که برعکس متد dataToByte عمل می کند و بایت ها را بر اساس نوع تعریف شده به حالت اولیه بر می گرداند تا مورد استفاده مجدد قرار گیرد، برای مثال در حالت JSON ابتدا بایت اسریم ورودی متد بر اساس یونیکد UTF-8 تبدیل به یک استرینگ شده و سپس توسط json.loads تبدیل به دیتا می شود و برگشت داده می شود.

```
    def logging(self, log: typing.Tuple[str]):
        t = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
        print(' - '.join((t, *log)))
```

متد logging برای ثبت لاگ ها مورد استفاده قرار می گیرد که یک مجموعه داده رشته ای شکل را به همراه زمان در آن لحظه ثبت می نماید.

```
2022-12-08 12:10:59 - sendMesssge - packetlen 118
2022-12-08 12:10:59 - reciveMessage - packetlen 21
2022-12-08 12:10:59 - close
```

تصویر فوق بخشی از اجرای برنامه است که متد logging برای سه فعالیت send، receive و close مورد استفاده قرار گرفته است.

```
    def sendMesssge(self, rpcSocket, data):
        data = self.dataToByte(data)
        header = len(data).to_bytes(self.headsize, byteorder='little')
        packet = header + data
        rpcSocket.sendall(packet)
        self.logging(('sendMesssge', f'packetlen {len(packet)}'))
```

متد sendMessage وظیفه ی ارسال دیتا را به سوکت داده شده دارد، این متد ابتدا داده ها را با استفاده از متد داخلی dataToByte تبدیل به بایت میکند، سپس مقدار هدر را محاسبه نموده و به ابتدای فایل اضافه می کند و آن را به عنوان packet ذخیره می کند، سپس با استفاده از کتابخانه socket و شی ساخته شده از آن packet را به ادرسی که در ابتدا هماهنگ شده ارسال می کند، در نهایت با استفاده از متد logging یک log ذخیره می کند.

```
    def reciveMessage(self, rpcSocket, buferSize):
        packet = rpcSocket.recv(buferSize)
        if (not packet):
            return self.CLOSE_CONNECTION, None
        header = packet[:self.headsize]
        data = packet[self.headsize:]

        decoded_header = int.from_bytes(header, 'little', signed=True)
        if decoded_header in [self.CLOSE_CONNECTION, self. CLOSE_REMOTE]:
            return decoded_header, None
        while (len(data) != int.from_bytes(header, 'little')):
            data += rpcSocket.recv(buferSize)
        packetLen = self.headsize + len(data)
        self.logging(('reciveMessage', f'packetlen {packetLen}'))
        return header, self.byteToData(data)
```

متد reciveMessage نیز به اندازه مقدار بافر از سوکت ساخته شده قرار است برایمان دیتا بگیرد، در ابتدا با socket.recv از محتوا میخواند و packet را می سازد سپس اگر packet وجود نداشت، درخواست پایان ارتباط را به ما نشان می دهد که بازگشت دادن مقدار -۱ می باشد، در ادامه اگر packet وجود داشت، هدر و دیتا را براساس اندازه هدر که قبلا مشخص کرده ایم انجام می دهد، در گام بعدی هدر را از بایت به مقدار صحیح بر می گرداند، در این بخش باید در نظر داشته باشیم که حتما محاسبه مقدار به صورت علامتدار انجام شود که در مقایسه آتی به مشکل بر نخوریم و بتوانیم با مقادیر پیشفرض -۱ و -۲ مقایسه انجام دهیم، در ادامه نیز مادامی که مقدار اندازه data برابر با مقدار هدر نیست باید محتوا را با استفاده از recv بخوانیم و به دیتا اضافه کنیم، در پایان نیز با استفاده از متدlogging گزارش استفاده از تابع و انجام صحیح reciveMessage را ثبت می نماییم.

کلاس rpcClient

این کلاس که از کلاس rpcClass ارث بری میکند، علاوه بر متد سازنده دارای ۵ متد دیگر می باشد که به صورت مشروح در ادامه توضیح داده خواهد شد.

از این کلاس برای راه اندازی بخش client استفاده میکنیم.

```
class rpcClinet(rpcClass):
```

کلاس rpcClient از کلاس rpcClass ارث بری میکند.

```
    def __init__(self, address, port, **kwargs):
        super().__init__(**kwargs)
        self.rpcSocket.connect((address, port))
        self.logging(('connect', f'to {address}:{str(port)}'))
```

متد سازنده ابتدا با استفاده از آرگومان های kwargs** که به صورت یک دیکشنری است سازنده پدرش یعنی rpcClass را فراخوانی میکند و سپس با کمک کتابخانه سوکت، ارتباط با سرور را که آدرس و پورت داده شده به سازنده را در آن می گذارد ایجاد میکند، با انجام درست این مرحله یک گزارش با استفاده از متد logging درج می گردد.

```
    def sendMesssgeClient(self, data):
        self.sendMesssge(self.rpcSocket, data)
```

متد sendmessageClient صرفا متد پیاده سازی شده در پدر که در بخش قبلی توضیح دادیم را فراخوانی میکند.

```
    def reciveMessageClient(self):
        return self.reciveMessage(self.rpcSocket, self.buferSize)
```

متد reciveMessageClient نیز همانند متد ارسال پیام در کلاینت صرفا به فراخوانی و برگرداندن مقدار reciveMessage که در rpcClass پیاده سازی شده است بسنده می کند.

```
    def close(self):
        self.rpcSocket.close()
        self.logging(('close', ))
```

متد close وظیفه بستن ارتباط ایجاد شده توسط سوکت را به عهده دارد و یک گزارش نیز ثبت می نماید.

```
    def closeConnection(self):
        packet = self.CLOSE_CONNECTION.to_bytes(self.headsize, byteorder='little'
, signed=True)
        self.rpcSocket.sendall(packet)
        self.close()
```

متد closeConnection وظیفه دارد بستن کانکشن را به اطلاع برساند و سپس اقدام به فراخوانی متد close که توضیح دادیم می کند.

```
    def closeRemote(self):
        packet = self.CLOSE_REMOTE.to_bytes(self.headsize, byteorder='little', si
gned=True)
        self.rpcSocket.sendall(packet)
        self.logging(('close remote ...',))
```

متد CloseRemote نیز وظیفه دارد تا بستن ریموت را مشخص میکند، این بدین معنی است که با فراخوانی این متد، یک پکت ایجاد شده که صرفا حاوی مقدار -۲ به صورت بایت است و ارسال می گردد که نشان دهنده پایان ارتباط می باشد.

کلاس rpcServer

این کلاس که از کلاس rpcClass ارث بری میکند، علاوه بر متد سازنده دارای ۴ متد دیگر می باشد که به صورت مشروح در ادامه توضیح داده خواهد شد.

از این کلاس برای راه اندازی بخش server استفاده میکنیم.

```
class rpcServer(rpcClass):
```

کلاس rpcServer از کلاس rpcClass ارث بری میکند.

```
    def __init__(self, port, **kwargs):
        super(rpcServer, self).__init__(**kwargs)
        self.rpcSocket.bind(('localhost',port))
        self.rpcSocket.listen(5)
        self.logging(("listening ...",))
```

سازنده ی این کلاس هم سازنده پدر را فرخوانی میکند و همچنین محیط جهت گوش کردن به داده ها را می سازند و منتظر می ماند تا داده ها را بگیرد، با استفاده از متد logging که پیش از این بارها از آن استفاده کرده بودیم این بخش نیز گزارش را ثبت می کنیم.

```
    def sendMesssgeServer(self, rpcSocket, data):
        self.sendMesssge(rpcSocket, data)
```

این متد همانند متد در rpcClient است.

```
    def reciveMessageServer(self, rpcSocket):
        return self.reciveMessage(rpcSocket, self.buferSize)
```

این متد نیز همانند متد در rpcServer می باشد.

```
    def close(self, rpcSocket):
        rpcSocket.close()
```

همچنین متد close نیز به همانصورت که در rpcClient تعریف شده است در این بخش نیز تعریف شده است با این تفاوت که ثبت log نداریم.

در ابتدا قرار بود که همه مراحل ثبت log داشته باشد و همچنین در فایل نیز ذخیره سازی انجام شود که با توجه به کمبود وقت این اتفاق نیفتاد، در آینده و زمانی که در github قرارداده شود بخش هایی به این توابع اضافه می گردد.

```
    def loop(self, callBack: callable):
        while (1):
            connection, address = self.rpcSocket.accept()
            self.logging(('connect', f'to {address[0]}:{str(address[1])}'))
            while (1):
                header, data = self.reciveMessageServer(connection)
                if (header == self.CLOSE_CONNECTION):
                    self.close(connection)
                    self.logging(('close', f'to {address[0]}:{str(address[1])}'))
                    break
```

```
                elif (header == self.CLOSE_REMOTE):
                    self.logging(('exit...',))
                    return
                else:
                    self.sendMesssgeServer(connection, callBack(data))
```

اما مهم ترین متد rpcServer بدون شک متد loop است که وظیفه دارد تا دائما درخواست های ارسالی را بپذیرد و بر اساس آنچه به او میدهیم درخواست ها را اجرا کند و توابعی را فراخوانی نماید، این متد در یک حلقه بی نهایت ارتباط را ایجاد میکند و بر اساس هدر دریافتی تصمیم میگیرد که چه فعالیتی را باید انجام دهد، اگر دو حالت CLOSE_CONNECTION یا CLOSE_REMOTE باشد اقدامات خاص مربوط به پایان را انجام میدهد و در غیر این صورت ارتباط را به همراه فرخوانی تابع گفته شده در ورودی loop انجام میدهد.

در این پروژه پیاده سازی شده فرایند جابجایی داده زمان بر است و اگر کلاس ها بزرگ باشند و دیتا ها زیاد باشد قطعا در فرایند به مشکل خواهیم خورد، اما مزیتی که وجود دارد امکان اتصال همزمان به سرور می باشد که همزمان امکان پذیر است.

توضیحات پروژه پیاده سازی شده

پروژه پیاده سازی شده بسیار ساده است، یک کلاس است با نام human که دارای ویژگی های بسیار ساده ای است و صرفا محاسبه شاخص توده بدنی یا BMI را انجام می دهد.

```
class human:

    def __init__(self,name, family, weight, height, salary):
        self.name = name
        self.family = family
        self.weight = weight
        self.height = height
        self.salary = salary

    def calcuteBMI(self):
        pass
```

این کلاس به دو فرمت مختلف در exClient و exServer تعریف شده است به این صورت که تابع calcuteBMI متفاوت بازنویسی شده است که در ادامه توضیحات را آورده ایم.

پیاده سازی پروژه

پیاده سازی پروژه در دو فایل exClient.py و exServer.py انجام شده است که تعریف پروژه در هر دو بخش آورده شده است.

پیاده سازی exClient

```
    def calcuteBMI(self):
        rt = -1
        try :
            client = rpcClinet(ADDRESS, PORT,
       mode = 'JSON', buferSize = BUFERSIZE,
       headSize=HEADSIZE)
            x = dict()
            x['FUNC'] = 'calcuteBMI'
            x['weight'] = self.weight
            x['height'] = self.height
            client.sendMesssgeClient(x) #without error
            _, rt = client.reciveMessageClient()
            client.closeConnection()
            client.closeRemote()
        except Exception as e:
            print("------ Error -----------")

        return rt
```

پیاده سازی calcuteBMI در کلاینت به این صورت است که ارتباط را با استفاده از rpcClient ایجاد میکنند، سپس محتوا را در قالب دیکشنری ذخیره سازی میکند و سپس با فراخوانی sendMessageClient محتوا را ارسال می کند تا در سرور محاسبات انجام شود و سپس برگشت محتوا انجام شده و در rt ذخیره سازی شود، سپس کانکشن بسته می شود و مقدار برگشت داده می شود.

```
ahb = human(
    name="Amirhossein",
    family="Babaeayan",
    weight=94,
    height=187,
    salary=2000
)
```

تعریف یک نمونه برای تست انجام شده است.

```
bmiResult = ahb.calcuteBMI()
print(f"BMI result: {bmiResult}")
```

فراخوانی انجام شده و سپس مقدار نمایش داده می شود.

تابع ارسال کلاس

```
def calcuteBMIwithsendHumanClass(h : human):
    rt = -1
    try:
        client = rpcClinet(ADDRESS, PORT, mode = 'JSON', buferSize = BUFERSIZE, h
eadSize=HEADSIZE)
        h_dict = dict()
        h_dict['FUNC'] = 'HumanClass'
        h_dict['name'] = h.name
        h_dict['family'] = h.family
        h_dict['weight'] = h.weight
        h_dict['height'] = h.height
        h_dict['salary'] = h.salary
        client.sendMesssgeClient(h_dict)
        _, rt = client.reciveMessageClient()
        client.closeConnection()
        client.closeRemote()
    except Exception as e:
        print('-------------Error-----------')
    return rt
```

این تابع یک شی از کلاس human را دریافت کپسوله و ارسال می نماید و مقدار محسابه شده را برمی گرداند.

پیاده سازی exServer

```
    def calcuteBMI(self):
        return self.weight/((0.01*self.height)**2)
```

متد calcuteBMI در کلاس human داخل exServer به فرم بالا تعریف شده است.

```
server = rpcServer(PORT, mode = 'JSON', buferSize = BUFERSIZE, headSize=HEADSIZE)
print("xxxxx")
server.loop(callBack=fff)
```

سرور راه اندازی می شود و سپس loop فراخوانی میشود تا دائما پاسخگویی انجام شود، callback تابع fff را فرخوانی میکند که در ادامه میبینیم:

```
def fff(data):
    if data['FUNC']=='calcuteBMI':
        print('we have a function call from client:')
        print(data)
```

```
        bmi = int(data['weight'])/((0.01*int(data['height']))**2)
    return bmi
```

این تابع اطلاعات را میگیرد و سپس bmi را محاسبه نموده و برمیگرداند.

تابع دریافت کلاس

در حالت دیگری که کلاس را به صورت کامل ارسال کرده باشیم به فرم ذیل پیاده سازی شده است.

```
def fff(data):
    if data['FUNC'] == 'HumanClass':
        print('we have a human class from client :')
        print(data)
        hp = human(
            name=data['name'],
            family=data['family'],
            weight=data['weight'],
            height=data['height'],
            salary=data['salary']
        )
        bmi = hp.calcuteBMI()
    return bmi
```

بدین صورت اقدام میکند که ابتدا شی را با استفاده از داده های استخراج شده میسازیم و پس با فراخوانی calculateBMI به عنوان متد کلاس human آن را محاسبه نموده و سپس بر می گردانیم.

یک نمونه اجرا با داده ها فوق

کلاینت

```
(base) D:\AUT\Learning\4011\DistributedSystems\Assignments\DS4011_Assignment01\src>python exClient.py
Calcute BMI with send Argument :
2022-12-08 14:20:34 - connect - to 127.0.0.1:7731
in sendmessage rpc:
b'3\x00\x00\x00{"FUNC": "calcuteBMI", "weight": 94, "height": 187}'
2022-12-08 14:20:34 - sendMesssge - packetlen 55
2022-12-08 14:20:34 - reciveMessage - packetlen 21
2022-12-08 14:20:34 - close
BMI result: 26.88095170007721
Calcute BMI with send Class :
2022-12-08 14:20:39 - connect - to 127.0.0.1:7731
in sendmessage rpc:
b'q\x00\x00\x00{"FUNC": "HumanClass", "name": "Amirhossein", "family": "Babaeayan", "weight": 94, "height": 187, "salary": 2000}'
2022-12-08 14:20:39 - sendMesssge - packetlen 117
2022-12-08 14:20:39 - reciveMessage - packetlen 21
2022-12-08 14:20:39 - close
BMI Class result: 26.88095170007721
```

سرور

```
(base) D:\AUT\Learning\4011\DistributedSystems\Assignments\DS4011_Assignment01\src>python exServer.py
2022-12-08 14:23:50 - listening ...
xxxxx
2022-12-08 14:23:54 - connect - to 127.0.0.1:12224
2022-12-08 14:23:54 - reciveMessage - packetlen 55
from rpcClassServer:  b'3\x00\x00\x00' {'FUNC': 'calcuteBMI', 'weight': 94, 'height': 187}
============fff==============
{'FUNC': 'calcuteBMI', 'weight': 94, 'height': 187}
in sendmessage rpc:
b'\x11\x00\x00\x0026.88095170007721'
2022-12-08 14:23:54 - sendMesssge - packetlen 21
from rpcClassServer:  -1 None
2022-12-08 14:23:54 - close - to 127.0.0.1:12224
2022-12-08 14:23:59 - connect - to 127.0.0.1:12227
2022-12-08 14:23:59 - reciveMessage - packetlen 117
from rpcClassServer:  b'q\x00\x00\x00' {'FUNC': 'HumanClass', 'name': 'Amirhossein', 'family': 'Babaeayan', 'weight': 94, 'height': 187, 'salary': 2000}
============fff==============
{'FUNC': 'HumanClass', 'name': 'Amirhossein', 'family': 'Babaeayan', 'weight': 94, 'height': 187, 'salary': 2000}
in sendmessage rpc:
b'\x11\x00\x00\x0026.88095170007721'
2022-12-08 14:23:59 - sendMesssge - packetlen 21
from rpcClassServer:  -1 None
2022-12-08 14:23:59 - close - to 127.0.0.1:12227
```

بایند کردن کلاینت ها و سرورها

برای این عملیات ما از یک ایده استفاده میکنیم آن هم این است که از یک سرور همیشه فعال استفاده میکنیم که اطلاعات آن را به همه کلاینت ها از قبل داده ایم و می توانند از آن سرور کمک گرفته و سرور مد نظر خودشان را درخواست دهند و سپس exServerFinder به آنها سرور مد نظرشان را برگرداند، در این پروژه چون ما در به صورت محلی کار را پیش میبریم از پورت های ۱۱۱۱۱ برای ServerFinder استفاده کرده ایم که درخواست به این پورت ارسال شده و این پورت، پورت سرور اصلی که عملیات را انجام میدهد بر می گرداند، این سرور به این صورت عمل میکند که همیشه فعال است و با دریافت درخواست مطلوب که نوع آن getServer است مقدار مربوط به پورت ۷۷۳۱ که پورت اساسی ما است را بر می گرداند، ما می توانستیم اگر روی پورت های مختلفی در حال اجرای سرورهای یکسانمان هستیم یکی را بر اساس انتخاب تصادفی یا چنین موردی برگردانیم، پیاده سازی سرور یابنده ی سرور در ادامه توضیح داده خواهد شد.

پیاده سازی exServerFinder

```
from rpcClass.rpcClass import rpcServer

import json
```

```
def sendServerForClient(data):
    if data['FUNC']=='GetServer':
        SERVERPORT = 7731
        return SERVERPORT
```

در این بخش میبینید که یک تابع تعریف شده است که پورت را به ما برمیگرداند.

```
serverF =rpcServer(PORT, mode = 'JSON', buferSize = BUFERSIZE, headSize=HEADSIZE)
serverF.loop(callBack=sendServerForClient)
```

این بخش نیز راه اندازی سرور می باشد که از همان rpcServer استفاده کرده ایم که خود یک فراخوان از راه دور میگیرد و مقدار port مطلوب را بر می گرداند، توضیحات کامل تر نیز در ابتدای این بخش آورده شده است.