BORANA UNIVERSITY

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Course title: Introduction to Distributed System

Course Code:CoSc4038

Project title : Client-Server Applications Using RPC

Acadamic Year:IV Semester:II

Prepared by Group Four(4)

№	Name of Student	Id №
1	GADISA ABERA	BRU/R/498/2014
2	ETAFA TARIKU	BRU/R/481/2014
3	SABONA MARARA	BRU/R/680/2014
4	MUBARIK MOHAMMED	BRU/R/654/2014
5	ABDUBA BORU	BRU/R/387/2014

Submitted to:HONEY G.(MSc)

Submission date:18/04/2025

Yabelo, Ethiopia

o

Catalog

❖ USER INTERFACE OF RPC CLIENT TO SERVER PROJECT

Implementing a simple client-server RPC application in Python involves defining remote procedures, setting up communication channels, and using serialization. Below is a step-by-step guide with code examples, followed by popular frameworks for RPC development

Steps to Create a Basic RPC Application

1. **Define the Remote Interface**
   Declare methods the server will expose (e.g., arithmetic operations).
2. **Implement the Server**
   - Use an RPC library to bind the server to a port.
   - Register the methods and handle incoming requests.
3. **Implement the Client**

- Connect to the server
- Call remote methods as if they were local.

4. **Handle Serialization**

   The framework automatically converts data to/from network-safe formats (e.g., XML, JSON).

➢ Code Examples For Server:

```
#Server code
from flask import Flask, request, jsonify
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("Flask-RPC-Server")
```

```
app = Flask(name)
```

```
@app.route('/echo', methods=['POST'])
def echo():
    """Return the same message received"""
    data = request.get_json()
    message = data.get('message', '')
    logger.info(f"Echoing message: {message}")
    return jsonify({'result': message})
```

```
@app.route('/add', methods=['POST'])
def add_numbers():
    """Add two numbers and return the result"""
    data = request.get_json()
    a = data.get('a', 0)
    b = data.get('b', 0)
    result = a + b
```

```
    logger.info(f"Adding {a} + {b} = {result}")
    return jsonify({'result': result})
```

```
@app.route('/subtract', methods=['POST'])
def subtract_numbers():
    """Subtract two numbers and return the result"""
    data = request.get_json()
    a = data.get('a', 0)
    b = data.get('b', 0)
    result = a - b
    logger.info(f"Subtracting {a} - {b} = {result}")
    return jsonify({'result': result})
```

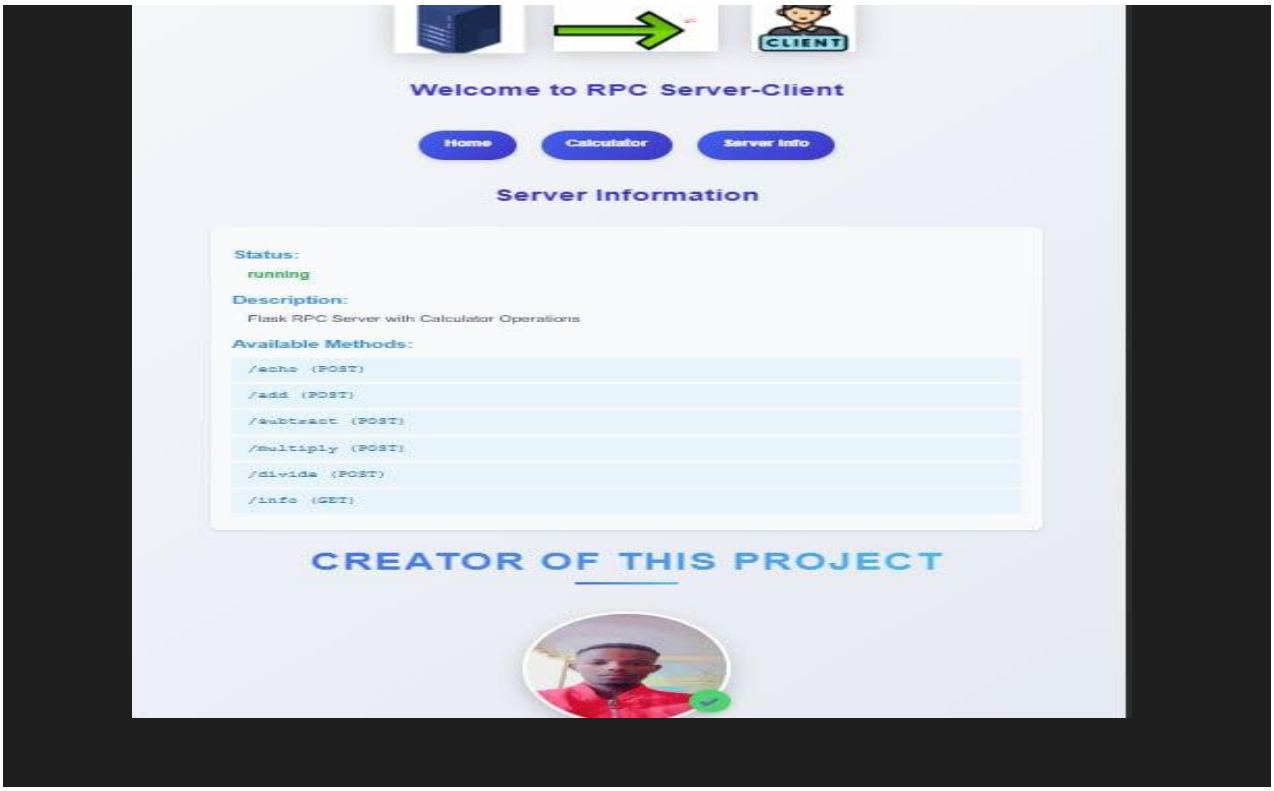
```
@app.route('/multiply', methods=['POST'])
def multiply_numbers():
    """Multiply two numbers and return the result"""
    data = request.get_json()
    a = data.get('a', 0)
    b = data.get('b', 0)
    result = a * b
    logger.info(f"Multiplying {a} * {b} = {result}")
    return jsonify({'result': result})
```

```
@app.route('/divide', methods=['POST'])
def divide_numbers():
    """Divide two numbers and return the result"""
    data = request.get_json()
    a = data.get('a', 0)
    b = data.get('b', 1)

    if b == 0:
        logger.error("Division by zero attempted")
        return jsonify({'error': 'Division by zero is not allowed'}), 400
```

```
    result = a / b
    logger.info(f"Dividing {a} / {b} = {result}")
    return jsonify({'result': result})
```

```
@app.route('/info', methods=['GET'])
def get_server_info():
    """Return server information"""
    info = {
        'status': 'running',
        'methods': [
            '/echo (POST)',
            '/add (POST)',
            '/subtract (POST)',
            '/multiply (POST)',
            '/divide (POST)',
            '/info (GET)'
        ],
        'description': 'Flask RPC Server with Calculator Operations'
    }
    logger.info("Returning server info")
    return jsonify(info)
```

```
if name == "main":
    app.run(host='0.0.0.0', port=8000, debug=True)
```



➢ Code Examples For Client:

```
from flask import Flask, render_template, request, redirect, url_for
import requests

app = Flask(name)
client = None
```

```
class FlaskRPCClient:
    def init(self, base_url="http://localhost:8000"):
        self.base_url = base_url

    def echo(self, message):
        url = f"{self.base_url}/echo"
        payload = {"message": message}
        response = requests.post(url, json=payload)
```

```
        return self._handle_response(response)

    def add(self, a, b):
        url = f"{self.base_url}/add"
        payload = {"a": a, "b": b}
        response = requests.post(url, json=payload)
        return self._handle_response(response)

    def subtract(self, a, b):
        url = f"{self.base_url}/subtract"
        payload = {"a": a, "b": b}
        response = requests.post(url, json=payload)
        return self._handle_response(response)

    def multiply(self, a, b):
        url = f"{self.base_url}/multiply"
        payload = {"a": a, "b": b}
        response = requests.post(url, json=payload)
        return self._handle_response(response)

    def divide(self, a, b):
        url = f"{self.base_url}/divide"
        payload = {"a": a, "b": b}
        response = requests.post(url, json=payload)
        return self._handle_response(response)

    def info(self):
        url = f"{self.base_url}/info"
        response = requests.get(url)
        return self._handle_response(response)

    def _handle_response(self, response):
```

```
        try:
            response.raise_for_status()
            return response.json()
        except requests.exceptions.HTTPError as err:
            try:
                return response.json()
            except:
                return {"error": str(err)}
        except requests.exceptions.RequestException as err:
            return {"error": str(err)}
```

```
@app.before_request
def initialize_client():
    global client
    client = FlaskRPCClient()
```

```
@app.route('/')
def index():
    return render_template('index.html')
```

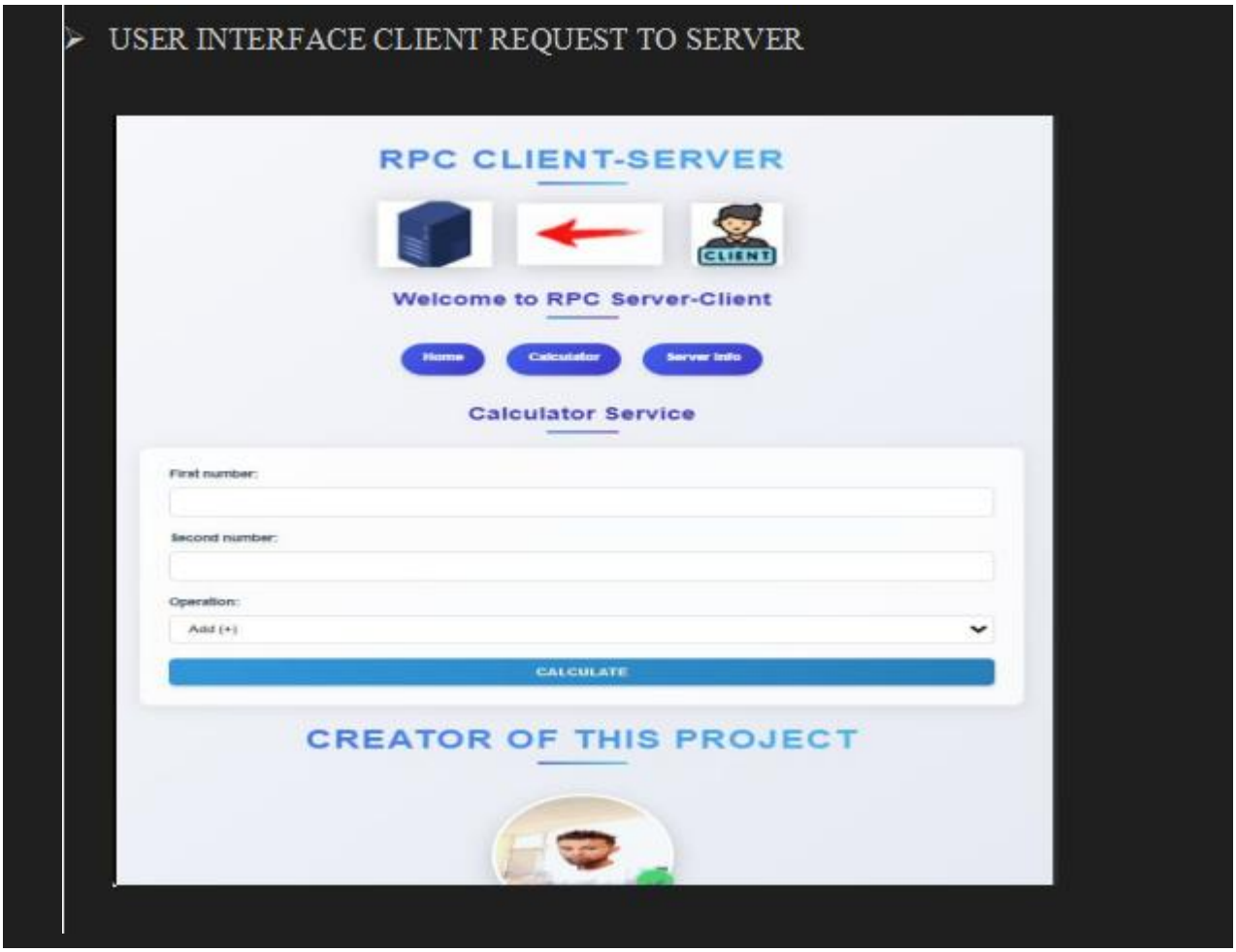
```
@app.route('/echo', methods=['GET', 'POST'])
def echo():
    if request.method == 'POST':
        message = request.form['message']
        result = client.echo(message)
        return render_template('echo.html', result=result, message=message)
    return render_template('echo.html')
```

```
@app.route('/calculator', methods=['GET', 'POST'])
def calculator():
    if request.method == 'POST':
        operation = request.form['operation']
        try:
```

```
            a = float(request.form['a'])
            b = float(request.form['b'])

            if operation == 'add':
                result = client.add(a, b)
            elif operation == 'subtract':
                result = client.subtract(a, b)
            elif operation == 'multiply':
                result = client.multiply(a, b)
            elif operation == 'divide':
                result = client.divide(a, b)

            return render_template('calculator.html',
                                   result=result.get('result', 'Error'),
                                   a=a, b=b, operation=operation)
        except ValueError:
            return render_template('calculator.html',
                                   error="Please enter valid numbers")
        except Exception as e:
            return render_template('calculator.html',
                                   error=str(e))

    return render_template('calculator.html')
```

```
@app.route('/info')
def info():
    server_info = client.info()
    return render_template('info.html', info=server_info)
```

```
if name == 'main':
    app.run(port=5000, debug=True)
```



Tools and Frameworks for RPC/RMI

RPC Frameworks

Framework	Protocol	Serialization	Use Case
gRPC	HTTP/2	Protocol Buffers	High-performance services
Apache Thrift	TCP/HTTP	Binary, JSON	Cross-language systems
XML-RPC	HTTP	XML	Legacy integrations
JSON-RPC	HTTP	JSON	Web APIs

Pyro	Custom	Pickle	Python-only systems
RPyC	TCP	Pickle	Python remote objects

RMI Frameworks

Framework	Language	Protocol	Use Case
Java RMI	Java	JRMP	Java distributed apps
CORBA	Multi	IIOP	Legacy enterprise systems
Hessian	Java/HTTP	Binary	Cross-platform RMI via HTTP

Summary

For simple RPC applications, Python's built-in `xmlrpc` or lightweight libraries like `python-jsonrpc` are ideal. For scalable systems, **gRPC** or **Apache Thrift** offer cross-language support. **Pyro** and **RPyC** excel in Python-centric environments, while **Java RMI** remains the standard for Java-based RMI.

Reference

1. https://coderspacket.com/rpc-implementation-using-python
2. https://safjan.com/how-to-use-rpc-in-python/
3. https://pypi.org/project/python-jsonrpc/