Experiment 1

Aim: The aim of this experiment is to design and implement a distributed application using Remote Procedure Call (RPC) for remote computation. Specifically, the experiment focuses on creating a client-server architecture where the client submits an integer value to the server, which then calculates the factorial of the given integer and returns the result to the client program.

Objective:

- To understand the concept of Remote Procedure Call (RPC) and its application in distributed computing.
- To design and implement a client-server architecture for remote computation using RPC.
- To develop a server program capable of computing the factorial of a given integer.
- To create a client program that interacts with the server to submit integer values for factorial computation and retrieve the results.
- To evaluate the effectiveness and efficiency of the implemented solution in performing remote computation tasks.

Theory:

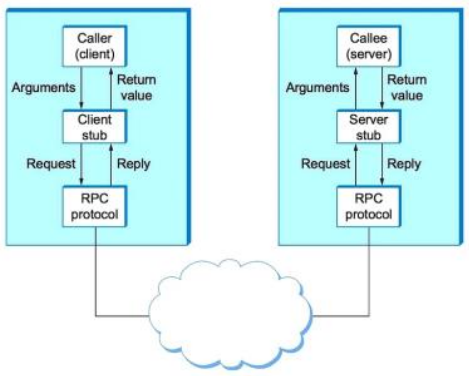
Remote Procedure Call (RPC) is a communication protocol that enables a program to execute code in another address space (typically on a remote machine) as if it were a local procedure call. It abstracts the complexity of network communication and allows developers to invoke functions or procedures on remote systems as if they were local function calls within their program.

Key Components of RPC:

- Client: The client is the program or process that initiates the RPC request. It invokes a procedure or function as if it were a local call, but the request is directed to a remote server.
- Server: The server is the program or process that receives the RPC request from the client, executes the requested procedure or function, and returns the result to the client.
- Stub: The client and server communicate through stubs, which act as proxies for the actual procedure or function being called. On the client side, the client stub marshals the parameters of the function call into a format suitable for transmission over the network. On the server side, the server stub unmarshals the parameters, executes the requested function, and marshals the result back to the client.
- Transport Layer: RPC relies on a transport layer protocol, such as TCP/IP or UDP, for transmitting data between the client and server over the network. The transport layer handles the low-level details of data transmission, ensuring reliability, ordering, and error detection.

RPC Workflow:

- Client Invocation: The client invokes a remote procedure call as if it were a local function call within its program. The parameters of the function call are marshaled into a message format suitable for transmission over the network.
- Message Transmission: The marshaled message is transmitted over the network to the server using the underlying transport layer protocol.
- Server Reception: The server receives the RPC request and passes it to the appropriate server stub, which unmarshals the parameters from the message.
- Server Execution: The server stub executes the requested procedure or function using the parameters provided. Once the computation is complete, the result is marshaled into a response message.
- Response Transmission: The response message containing the result is transmitted back to the client over the network.
- Client Reception: The client receives the response message and passes it to the client stub, which unmarshals the result.
- Client Processing: The client program processes the result returned by the server and continues execution as necessary.



Remote Procedure Call (RPC)

Applications in Distributed Computing:

RPC is commonly used in distributed computing environments where programs running on different machines need to communicate and collaborate to perform tasks. By abstracting the complexities of network communication and remote procedure invocation, RPC simplifies the development of distributed applications. In the context of the experiment described, RPC is used to implement a client-server architecture for remote computation. The client submits an integer value to the server using an RPC request, and the server calculates the factorial of the given integer. The result is then returned to the client using another RPC call. This approach enables the client and server components to communicate and collaborate effectively over a network, allowing for the distribution of computation tasks across multiple machines. Overall, RPC provides a convenient and efficient mechanism for building distributed applications,

enabling seamless interaction between remote components while abstracting away the complexities of network communication.

Outcome:

- Successful implementation of a distributed application architecture using RPC for remote computation.
- Creation of a functional server program capable of computing factorials based on client requests.
- Development of a client program enabling users to submit integer values for factorial computation and receive results from the server.
- Demonstration of effective communication and data exchange between the client and server components.
- Evaluation of the performance and reliability of the implemented solution in handling remote computation tasks.

Conclusion:

The experiment successfully demonstrated the design and implementation of a distributed application using Remote Procedure Call (RPC) for remote computation. The developed solution effectively facilitated communication between client and server components, enabling seamless submission of integer values for factorial computation and retrieval of results. Through this experiment, we gained a deeper understanding of RPC principles and their practical application in distributed computing scenarios. Further optimizations and enhancements could be explored to improve the performance and scalability of the implemented solution for real-world deployment.

Code:

Factserver.py

C: > Vridhi > BE - AIDS > sem 8 > DC > a1 > Factserver.py > ...

```python
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
class FactorialServer:
    def calculate_factorial(self, n):
        if n < 0:
            raise ValueError("Input must be a non-negative integer.")
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result
# Restrict to a particular path.
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)
# Create server
with SimpleXMLRPCServer(('localhost', 8000),
                        requestHandler=RequestHandler) as server:
    server.register_introspection_functions()
    # Register the FactorialServer class
    server.register_instance(FactorialServer())
    print("FactorialServer is ready to accept requests.")
    # Run the server's main loop
    server.serve_forever()

```

Factclient.py

C: > Vridhi > BE - AIDS > sem 8 > DC > a1 > Factclient.py

```python
import xmlrpc.client
# Create an XML-RPC client
with xmlrpc.client.ServerProxy("http://localhost:8000/RPC2") as proxy:
    try:
        # Replace 5 with the desired integer value
        #input_value = 5
        input_value_str=input("enter the number: ")
        input_value=int(input_value_str)

        result = proxy.calculate_factorial(input_value)
        print(f"Factorial of {input_value} is: {result}")
    except Exception as e:
        print(f"Error: {e}")

```

Output: