Aim: Implement a Client/server using RPC/RMI

Theory:

RMI (Remote Method Invocation)

RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application issometimesreferred to as a distributed object application.

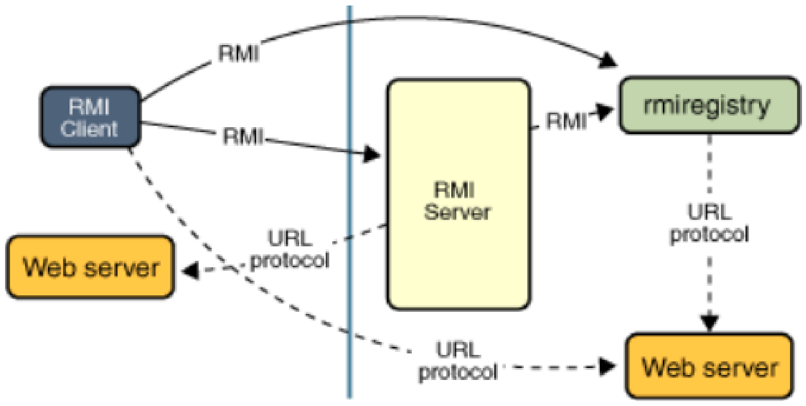
Distributed object applications need to do the following:

● Locate remote objects. Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.

● Communicate with remote objects. Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.

● Load class definitions for objects that are passed around. Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

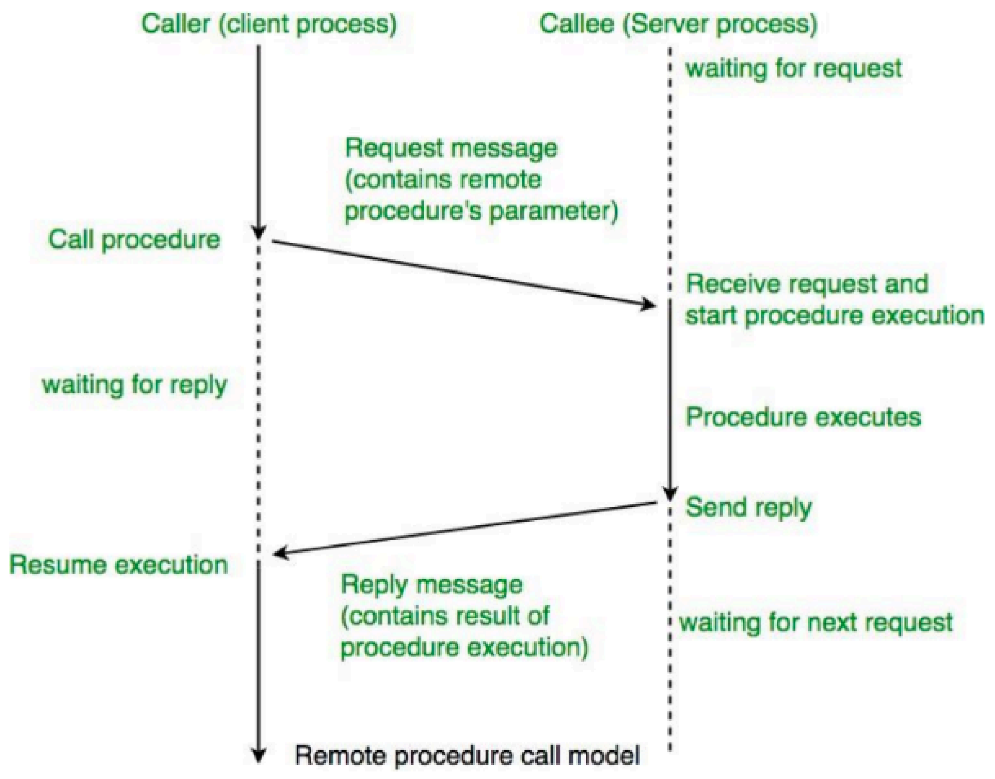
The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



Remote Procedure Call (RPC)

Remote Procedure Call (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the conventional local procedure calling so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

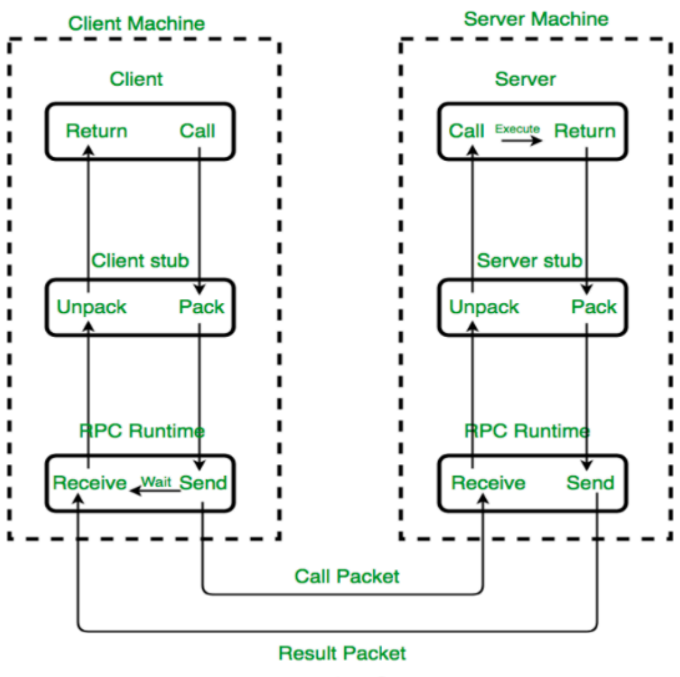
When making a Remote Procedure Call:


Caller (client process)
Callee (Server process)
waiting for request
Request message (contains remote procedure's parameter)
Call procedure
Receive request and start procedure execution
waiting for reply
Procedure executes
Send reply
Resume execution
Reply message (contains result of procedure execution)
waiting for next request
Remote procedure call model

1. The calling environment is suspended, procedure parameters are transferred across the network to the environment where the procedure is to execute, and the procedure is executed there.

2. When the procedure finishes and produces its results, its results are transferred back to the calling environment, where execution resumes as if returning from a regular procedure call.
NOTE: RPC is especially well suited for client-server (e.g. query-response) interaction in which the flow of control alternates between the caller and callee. Conceptually, the client and server do not both execute at the same time. Instead, the thread of execution jumps from the caller to the callee and then back again.

Working of RPC



Implementation of RPC mechanism

<u>The following steps take place during a RPC :</u>

1. A client invokes a client stub procedure, passing parameters in the usual way. The client stub resides within the client‘s own address space.

2. The client stub marshalls(pack) the parameters into a message. Marshaling includes converting the representation of the parameters into a standard format, and copying each parameter into the message.

3. The client stub passes the message to the transport layer, which sends it to the remote server
machine.

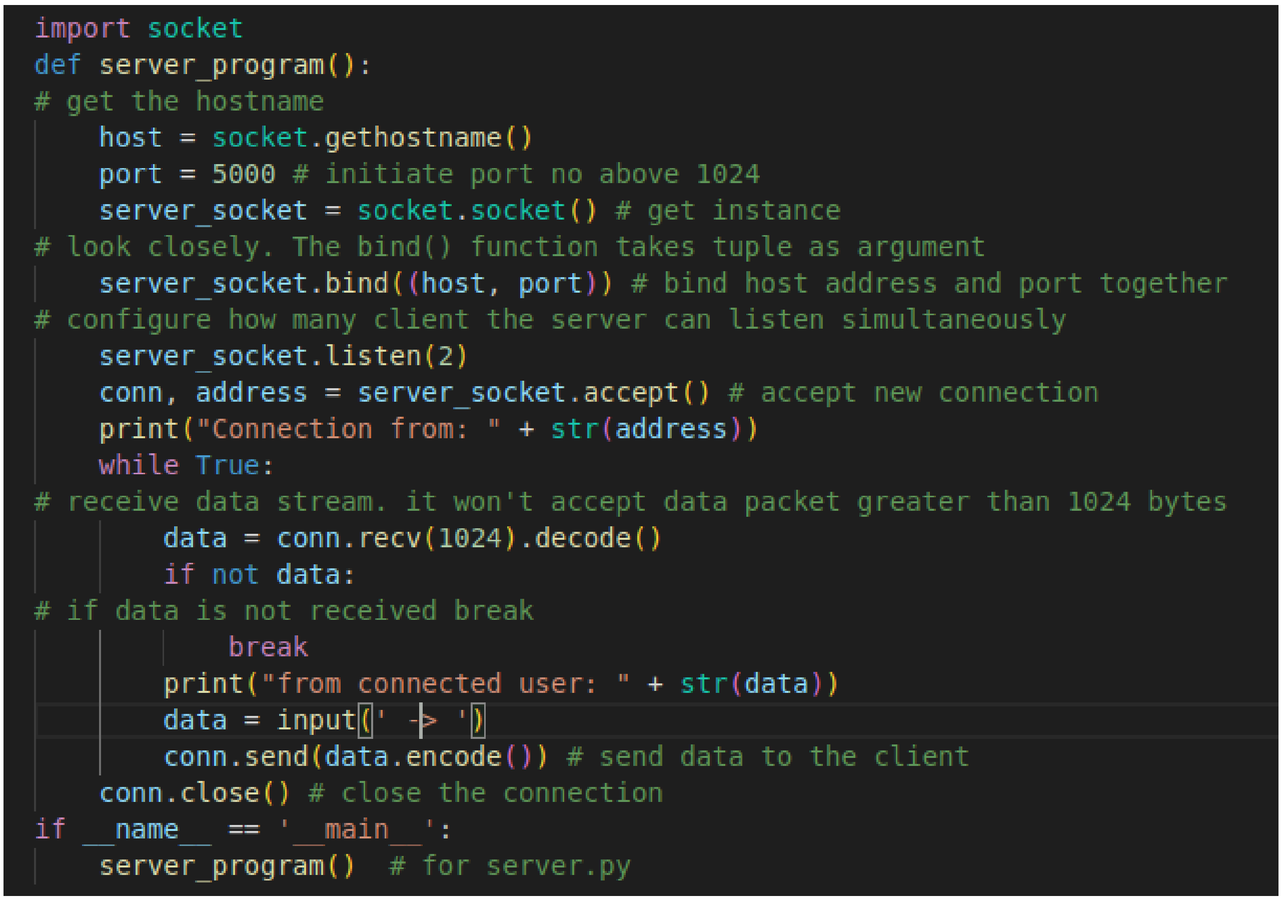
4. On the server, the transport layer passes the message to a server stub, which demarshalls(unpack) the parameters and calls the desired server routine using the regular procedure call mechanism.

5. When the server procedure completes, it returns to the server stub (e.g., via a normal procedure call return), which marshalls the return values into a message. The server stub then hands the message to the transport layer.

6. The transport layer sends the result message back to the client transport layer, which hands the message back to the client stub.

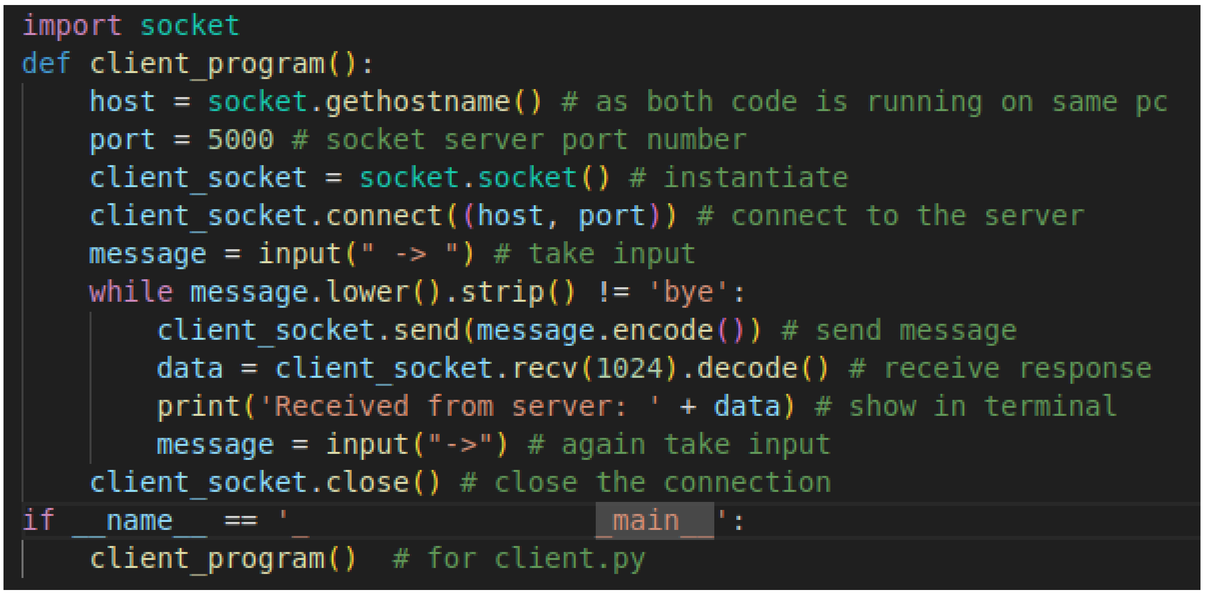
7. The client stub demarshalls the return parameters and execution returns to the caller.

Program: Server.py



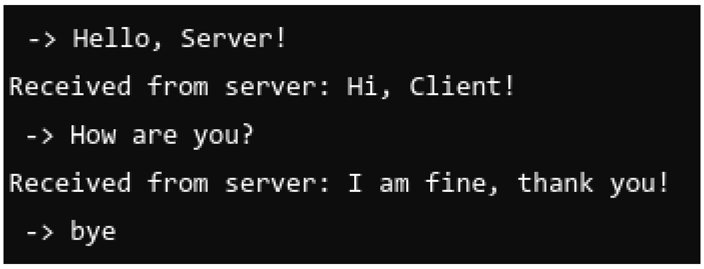
```python
import socket
def server_program():
# get the hostname
    host = socket.gethostname()
    port = 5000 # initiate port no above 1024
    server_socket = socket.socket() # get instance
# look closely. The bind() function takes tuple as argument
    server_socket.bind((host, port)) # bind host address and port together
# configure how many client the server can listen simultaneously
    server_socket.listen(2)
    conn, address = server_socket.accept() # accept new connection
    print("Connection from: " + str(address))
    while True:
# receive data stream. it won't accept data packet greater than 1024 bytes
        data = conn.recv(1024).decode()
        if not data:
# if data is not received break
            break
        print("from connected user: " + str(data))
        data = input(' -> ')
        conn.send(data.encode()) # send data to the client
    conn.close() # close the connection
if __name__ == '__main__':
    server_program()  # for server.py
```

Client.py

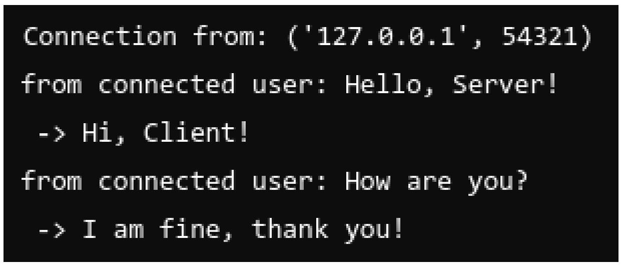


```python
import socket
def client_program():
    host = socket.gethostname() # as both code is running on same pc
    port = 5000 # socket server port number
    client_socket = socket.socket() # instantiate
    client_socket.connect((host, port)) # connect to the server
    message = input(" -> ") # take input
    while message.lower().strip() != 'bye':
        client_socket.send(message.encode()) # send message
        data = client_socket.recv(1024).decode() # receive response
        print('Received from server: ' + data) # show in terminal
        message = input("->") # again take input
    client_socket.close() # close the connection
if __name__ == '__main__':
    client_program()  # for client.py
```

Output:



```
 -> Hello, Server!
Received from server: Hi, Client!
 -> How are you?
Received from server: I am fine, thank you!
 -> bye
```



```
Connection from: ('127.0.0.1', 54321)
from connected user: Hello, Server!
 -> Hi, Client!
from connected user: How are you?
 -> I am fine, thank you!
```