DC codes explanation

explain code line by line:
server.py:

```
from xmlrpc.server import SimpleXMLRPCServer
import logging

# Enable logging
logging.basicConfig(level=logging.INFO)

# Function to compute factorial
def factorial(n):
    if n < 0:
        return "Error: Factorial of negative numbers is not
defined."
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

# Create an RPC server
server = SimpleXMLRPCServer(("localhost", 8000),
allow_none=True)
logging.info("RPC Server is running on port 8000...")

# Register the factorial function
server.register_function(factorial, "factorial")

# Run the server
server.serve_forever()
```

client.py:

```
import xmlrpc.client

# Connect to the server
server = xmlrpc.client.ServerProxy("http://localhost:8000/")

# Take user input
num = int(input("Enter an integer to calculate factorial: "))

# Request factorial computation from server
```

```
try:
    result = server.factorial(num)
    print(f"Factorial of {num} is: {result}")
except Exception as e:
    print(f"Error: {e}")
```

Here's a **line-by-line explanation** of both `server.py` and `client.py`:

🔹 `server.py`

python

```
from xmlrpc.server import SimpleXMLRPCServer
```

- Imports the `SimpleXMLRPCServer` class, which allows the creation of an XML-RPC server for remote procedure calls over HTTP.

python

```
import logging
```

- Imports the `logging` module to enable logging messages to the console.

python

```
logging.basicConfig(level=logging.INFO)
```

- Sets up basic configuration for logging with `INFO` level, so messages like "Server started" will be printed.

python

```
def factorial(n):
```

- Defines a function `factorial` that takes an integer `n`.

python

```
    if n < 0:
        return "Error: Factorial of negative numbers is not defined."
```

- Checks if `n` is negative. If yes, returns an error message.

python

```
    result = 1
```

- Initializes the result to 1.

python

```
    for i in range(2, n + 1):
        result *= i
```

- Loops from 2 to `n`, multiplying each number to `result` to compute the factorial.

python

```
    return result
```

- Returns the final factorial result.

python

```
server = SimpleXMLRPCServer(("localhost", 8000), allow_none=True)
```

- Creates an XML-RPC server on `localhost` at port `8000`. `allow_none=True` allows `None` values to be passed.

python

```
logging.info("RPC Server is running on port 8000...")
```

- Logs that the server is running.

python

```
server.register_function(factorial, "factorial")
```

- Registers the `factorial` function to be callable remotely via the name `"factorial"`.

python

```
server.serve_forever()
```

- Starts the server to handle incoming requests indefinitely.

- 🔹 `client.py`

python

```
import xmlrpc.client
```

- Imports the `xmlrpc.client` module to connect to the XML-RPC server.

python

```
server = xmlrpc.client.ServerProxy("http://localhost:8000/")
```

- Connects to the server at `localhost` on port `8000`.

python

```
num = int(input("Enter an integer to calculate factorial: "))
```

- Prompts the user to input an integer and converts it to an `int`.

python

```
try:
    result = server.factorial(num)
```

- Calls the `factorial` function on the server and stores the result.

python

```
    print(f"Factorial of {num} is: {result}")
```

- Prints the result returned from the server.

python

```
except Exception as e:
    print(f"Error: {e}")
```

- Catches any error (like server issues) and prints an error message.

Would you like to add more functions or run this over a network instead of localhost?

explain code line by line:
server.py:
import Pyro5.api

@Pyro5.api.expose
class StringService:
    def concatenate(self, str1, str2):
        return str1 + str2

def main():
    daemon = Pyro5.api.Daemon()
    uri = daemon.register(StringService)
    print("Ready. Object URI =", uri)
    daemon.requestLoop()

if __name__ == "__main__":
    main()

client.py:
import Pyro5.api

def main():
    uri = input("Enter the server object URI: ") # e.g.
PYRO:obj_xxxxx@localhost:xxxx

```
    server = Pyro5.api.Proxy(uri)

    str1 = input("Enter first string: ")
    str2 = input("Enter second string: ")
    result = server.concatenate(str1, str2)
    print("Concatenated string:", result)

if __name__ == "__main__":
    main()
```

Here's a **line-by-line explanation** of both `server.py` and `client.py` that use **Pyro5** (Python Remote Objects) for building a remote procedure call (RPC) system:

🔹 `server.py`

python

```
import Pyro5.api
```

- Imports the Pyro5 module, which is used to create Python objects that can be accessed remotely over a network.

python

```
@Pyro5.api.expose
```

- This decorator is used to mark the methods and classes to be exposed as remote callable objects. Here, it is applied to the `StringService` class to make it accessible to remote clients.

python

```
class StringService:
```

- Defines a class `StringService`, which will be the object that clients will call remotely.

python

```
    def concatenate(self, str1, str2):
        return str1 + str2
```

- Defines a method `concatenate` that takes two strings as input and returns their concatenation. This method will be remotely accessible through the server.

python

```
def main():
```

- Defines the main function where the server setup occurs.

python

```
    daemon = Pyro5.api.Daemon()
```

- Creates a Pyro5 daemon object. The daemon handles the network communication for remote clients to interact with the server.

python

```
    uri = daemon.register(StringService)
```

- Registers the `StringService` class with the Pyro5 daemon. This makes the class available to be accessed remotely. The `uri` returned here is a unique identifier for the object.

python

```
    print("Ready. Object URI =", uri)
```

- Prints the URI (Uniform Resource Identifier) of the registered object, so the client can use it to connect to the server.

python

```
    daemon.requestLoop()
```

- Starts the daemon's event loop, which listens for incoming remote procedure calls from clients. The server will continue running indefinitely until manually stopped.

```python
if __name__ == "__main__":
    main()
```

- Ensures that the `main` function is called when this script is executed directly (not imported as a module).

🔹 `client.py`

```python
import Pyro5.api
```

- Imports the Pyro5 module to allow the client to connect to and interact with the Pyro5 server.

```python
def main():
```

- Defines the main function where the client's interaction with the server occurs.

```python
    uri = input("Enter the server object URI: ")  # e.g. PYRO:obj_xxxxx@localhost:xxxx
```

- Prompts the user to input the URI of the server object. This URI was printed by the server when it started. Example format: `PYRO:obj_xxxxx@localhost:xxxx`.

```python

```

```
    server = Pyro5.api.Proxy(uri)
```

- Creates a `Proxy` object for the remote server. This proxy will allow the client to call methods on the remote object (in this case, the `StringService` class) as if it were a local object.

python

```
    str1 = input("Enter first string: ")
    str2 = input("Enter second string: ")
```

- Prompts the user to input two strings that they want to concatenate.

python

```
    result = server.concatenate(str1, str2)
```

- Calls the `concatenate` method of the server through the proxy object, passing the two input strings as arguments. The result will be the concatenated string returned from the server.

python

```
    print("Concatenated string:", result)
```

- Prints the concatenated result returned by the server.

python

```
if __name__ == "__main__":
    main()
```

- Ensures that the `main` function is executed when the client script is run directly.

How This Works:

- **Server (`server.py`)**: The server exposes a remote object (`StringService`) with a `concatenate` method. It listens for client connections using the Pyro5 daemon.
- **Client (`client.py`)**: The client connects to the server via the URI provided, and then calls the `concatenate` method remotely to get the result.

This setup enables remote method invocation, allowing the client to call server-side methods as though they were local.

Let me know if you'd like to explore more on Pyro5 or remote communication!

explain code line by line:
mapper_char.py:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    line = line.strip().replace(" ", "")
    for char in line:
        print("{0}\t1".format(char))
```

reducer_char.py:

```
#!/usr/bin/env python
import sys

current_char = None
count = 0

for line in sys.stdin:
    char, value = line.strip().split('\t')
    value = int(value)

    if char == current_char:
        count += value
    else:
        if current_char:
            print("{0}\t{1}".format(current_char, count))
        current_char = char
        count = value

if current_char:
    print("{0}\t{1}".format(current_char, count))
```

Here's a **line-by-line explanation** of the `mapper_char.py` and `reducer_char.py` codes. These are typical for a MapReduce program, where the **mapper** processes input data and the **reducer** aggregates the results.

🔹 `mapper_char.py`

python

```
#!/usr/bin/env python
```

- This line is called a shebang and specifies the interpreter to run the script with. In this case, it uses the Python interpreter from the system's environment.

python

```
import sys
```

- Imports the `sys` module, which provides access to system-specific parameters and functions. Here, it's used to read input from standard input (`stdin`).

python

```
for line in sys.stdin:
```

- Loops through each line of input received from standard input. In a MapReduce context, the input will usually come from files or data streams.

python

```
    line = line.strip().replace(" ", "")
```

- Strips leading/trailing whitespace from each line and removes all spaces from it (`replace(" ", "")`). The purpose is to focus on the characters only.

python

```

```

```
    for char in line:
```

- Loops through each character in the processed `line`.

```python
        print("{0}\t1".format(char))
```

- For each character, it prints a key-value pair in the form of `char\t1`. The key is the character itself, and the value `1` represents the occurrence of that character. The tab (`\t`) separates the key and value. This is the output of the **mapper** that will be passed to the **reducer**.

- 🔹 `reducer_char.py`

```python
#!/usr/bin/env python
```

- Specifies the Python interpreter to run this script, similar to the mapper script.

```python
import sys
```

- Imports the `sys` module to handle standard input (`stdin`) for reading data passed from the mapper.

```python
current_char = None
count = 0
```

- Initializes two variables:
  - `current_char`: Holds the character that is currently being processed.

- `count` : Keeps track of the occurrences of `current_char` .

```python
for line in sys.stdin:
```

- Loops through each line of input received from standard input, which is the output of the mapper (key-value pairs).

```python
    char, value = line.strip().split('\t')
```

- Strips any leading or trailing whitespace from the line, and splits the line into two parts: `char` (the key) and `value` (the count). The line is expected to be in the format `char\t1` .

```python
    value = int(value)
```

- Converts the `value` (which is a string `"1"`) to an integer.

```python
    if char == current_char:
        count += value
```

- Checks if the current character (`char`) matches the `current_char` being processed. If it does, it adds `value` to `count` , aggregating the total count for that character.

```python
    else:
        if current_char:
            print("{0}\t{1}".format(current_char, count))
```

- If `char` is different from `current_char` , it means a new character has been encountered. It then prints the previous character and its total count as a key-value pair

(`char\tcount`). This is the output of the **reducer** for the previously processed character.

```python
        current_char = char
        count = value
```

- Sets the `current_char` to the new `char` and initializes the `count` with the `value` for this character.

```python
if current_char:
    print("{0}\t{1}".format(current_char, count))
```

- After finishing the loop, if there's a remaining character (`current_char`), it prints the final count for that character.

How This Works:

- **Mapper (`mapper_char.py`)**:
  - It processes each line of input by removing spaces and iterating over each character.
  - For each character, it outputs a key-value pair of `char\t1`, indicating that this character appeared once in the input.
- **Reducer (`reducer_char.py`)**:
  - It aggregates the occurrences of each character. It reads key-value pairs (`char\t1`), and for each unique character, it sums the occurrences and outputs the total count.

Example Flow:

1. **Input to Mapper**:

   ```nginx
   hello world
   ```

   Mapper Output (each character with `1` as value):

```nginx
h	1
e	1
l	1
l	1
o	1
w	1
o	1
r	1
l	1
d	1
```

2. **Input to Reducer** (sorted by key, typically done by MapReduce framework automatically):

```nginx
d	1
e	1
h	1
l	3
o	2
r	1
w	1
```

3. **Final Output of Reducer**:

```nginx
d	1
e	1
h	1
l	3
o	2
r	1
w	1
```

Let me know if you'd like further clarification!

explain code line by line:
mapper_word.py:

```
#!/usr/bin/env python
import sys
import re

for line in sys.stdin:
    words = re.findall(r'\w+', line.lower())
    for word in words:
        print("{0}\t1".format(word))

reducer_word.py:
#!/usr/bin/env python
import sys

current_word = None
count = 0

for line in sys.stdin:
    word, value = line.strip().split('\t')
    value = int(value)

    if word == current_word:
        count += value
    else:
        if current_word:
            print("{0}\t{1}".format(current_word, count))
        current_word = word
        count = value

if current_word:
    print("{0}\t{1}".format(current_word, count))
```

Here's a **line-by-line explanation** of the `mapper_word.py` and `reducer_word.py` code, which are designed to count the frequency of words in input text using MapReduce principles.

- `mapper_word.py`

```python
#!/usr/bin/env python
```

- This line is a shebang that tells the system to use the Python interpreter to execute this script.

```python
import sys
```

- Imports the `sys` module, which provides access to system-specific parameters and functions. Here, it is used to read input from standard input (`stdin`).

```python
import re
```

- Imports the `re` module for regular expression operations. It is used here to extract words from each line of text.

```python
for line in sys.stdin:
```

- Loops through each line of input received from standard input. In a MapReduce context, this input would typically be passed by the MapReduce framework from files or other data streams.

```python
    words = re.findall(r'\w+', line.lower())
```

- Uses the `findall` method of the `re` module to extract all the words in the line. The regular expression `\w+` matches sequences of word characters (letters, digits, and underscores).
- The `line.lower()` call ensures that the input is converted to lowercase before word extraction, so the word count is case-insensitive.

```python

```

```
    for word in words:
```

- Loops through each word found in the `words` list.

```python
        print("{0}\t1".format(word))
```

- For each word, it prints a key-value pair in the format `word\t1`. The word is the key, and the value `1` indicates that the word appeared once in the input. The tab (`\t`) separates the key and value.

- 🔹 `reducer_word.py`

```python
#!/usr/bin/env python
```

- Similar to the mapper, this line specifies the Python interpreter to run this script.

```python
import sys
```

- Imports the `sys` module for handling standard input (`stdin`) and standard output (`stdout`).

```python
current_word = None
count = 0
```

- Initializes two variables:
  - `current_word`: Keeps track of the word currently being processed.
  - `count`: Counts the occurrences of `current_word`.

python

```
for line in sys.stdin:
```

- Loops through each line of input received from standard input, which is the output from the mapper (key-value pairs).

python

```
    word, value = line.strip().split('\t')
```

- Strips leading/trailing whitespace from the line, then splits the line into two parts: `word` (the key) and `value` (the count). The line is expected to be in the format `word\t1`.

python

```
    value = int(value)
```

- Converts the `value` (which is a string `"1"`) into an integer.

python

```
    if word == current_word:
        count += value
```

- If the current `word` matches `current_word`, it means the word has appeared again, so the `count` is incremented by the value (`1` in this case).

python

```
    else:
        if current_word:
            print("{0}\t{1}".format(current_word, count))
```

- If the `word` is different from `current_word`, it means a new word has been encountered. The count for the previous word is printed in the format `word\tcount`.

python

```
        current_word = word
        count = value
```

- Updates the `current_word` to the new `word` and sets the `count` to the `value` (which is `1` for each occurrence).

python

```python
if current_word:
    print("{0}\t{1}".format(current_word, count))
```

- After the loop ends, if there is still a `current_word`, it prints the final count for that word.

How This Works:

- **Mapper (`mapper_word.py`):**
  - Reads each line of input, converts it to lowercase, and extracts all words.
  - For each word, it emits a key-value pair (`word\t1`), indicating that the word appeared once in the input.
- **Reducer (`reducer_word.py`):**
  - Aggregates the occurrences of each word. It reads key-value pairs (`word\t1`) and sums up the occurrences of each word.
  - For each unique word, the reducer outputs the total count in the format `word\tcount`.

Example Flow:

1. **Input to Mapper**:

   nginx

   ```
   Hello world
   Hello Python
   hello world
   ```

2. **Mapper Output** (emits each word with a `1`):

```nginx
hello   1
world   1
hello   1
python  1
hello   1
world   1
```

3. **Input to Reducer** (sorted by key, typically done by MapReduce framework automatically):

```nginx
hello   1
hello   1
hello   1
python  1
world   1
world   1
```

4. **Reducer Output** (final word counts):

```nginx
hello   3
python  1
world   2
```

This process is the core of how word count programs work in MapReduce frameworks.

Let me know if you need further clarification!

explain code line by line:
import random
import threading
import time

servers = [{"id": i, "active": 0} for i in range(3)]
idx = 0

def handle(server, req_id):
    server["active"] += 1

```
    print(f"Server {server['id']} handling {req_id} | Active:
{server['active']}")
    time.sleep(random.uniform(1, 3))
    server["active"] -= 1
    print(f"Server {server['id']} finished {req_id} | Active:
{server['active']}")

def distribute(req_id):
    global idx
    server = servers[idx]
    idx = (idx + 1) % len(servers)
    threading.Thread(target=handle, args=(server,
req_id)).start()

if __name__ == "__main__":
    for i in range(1, 11):
        time.sleep(random.uniform(0.5, 1.5))
        distribute(i)
```

Here's a **line-by-line explanation** of the code that simulates a load balancing system where requests are distributed among multiple servers using threads:

Imports:

```python
import random
```

- Imports the `random` module, which allows the generation of random numbers. It's used here to introduce randomness in request delays and server handling times.

```python
import threading
```

- Imports the `threading` module, which allows the creation and management of threads. This is used to simulate concurrent handling of requests by multiple servers.

python

```python
import time
```

- Imports the `time` module, which provides functions to work with time-related tasks. It's used here for adding delays (sleep) to simulate real-time processing.

Global Variables:

```python
servers = [{"id": i, "active": 0} for i in range(3)]
```

- Creates a list of dictionaries representing three servers. Each server has:
  - `id`: A unique identifier ranging from 0 to 2.
  - `active`: A counter that tracks how many requests the server is currently handling (initialized to 0).

```python
idx = 0
```

- Initializes `idx` to `0`. This variable tracks which server will handle the next request in a round-robin fashion (i.e., the next request is assigned to the next server in the list).

Functions:

`handle(server, req_id)`:

```python
def handle(server, req_id):
```

- Defines the `handle` function, which simulates a server handling a request. The function takes:

- `server`: A dictionary representing the server.
- `req_id`: The ID of the request being handled.

```python
    server["active"] += 1
```

- Increments the `active` counter for the server, indicating that the server is now handling one more request.

```python
    print(f"Server {server['id']} handling {req_id} | Active: {server['active']}")
```

- Prints a message to show which server is handling which request and the current number of active requests on that server.

```python
    time.sleep(random.uniform(1, 3))
```

- Simulates the server processing the request by making the thread sleep for a random duration between 1 and 3 seconds. This mimics the time taken by the server to process the request.

```python
    server["active"] -= 1
```

- Decrements the `active` counter, indicating that the server has finished processing the request.

```python
    print(f"Server {server['id']} finished {req_id} | Active: {server['active']}")
```

- Prints a message to show which server has finished processing the request and the updated number of active requests on that server.

`distribute(req_id)`:

```python
def distribute(req_id):
```

- Defines the `distribute` function, which is responsible for assigning a request to a server. It takes the request ID (`req_id`) as an argument.

```python
    global idx
```

- Declares that `idx` is a global variable, so its value can be updated within the function. This is necessary to keep track of which server will handle the next request.

```python
    server = servers[idx]
```

- Selects the server based on the current value of `idx`. The `idx` tracks which server should handle the next request.

```python
    idx = (idx + 1) % len(servers)
```

- Updates `idx` to the next server in the list using a round-robin approach. The modulo operation (`% len(servers)`) ensures that once `idx` reaches the end of the list, it wraps back to `0` (the first server).

```python
    threading.Thread(target=handle, args=(server, req_id)).start()
```

- Creates a new thread to call the `handle` function for the selected server with the given `req_id`. This allows the server to handle the request concurrently with other requests. The `start()` method starts the thread immediately.

Main Code Block:

```python
if __name__ == "__main__":
```

- Ensures that this block of code only runs if the script is executed directly (not imported as a module).

```python
    for i in range(1, 11):
```

- A loop that iterates over 10 request IDs (from 1 to 10). This simulates 10 incoming requests.

```python
        time.sleep(random.uniform(0.5, 1.5))
```

- Introduces a random delay between 0.5 and 1.5 seconds between each request to simulate variability in incoming requests.

```python
        distribute(i)
```

- Calls the `distribute` function with the current request ID (`i`). This assigns the request to a server and handles it concurrently in a new thread.

How This Works:

1. **Server Initialization**: Three servers are created with IDs `0`, `1`, and `2`. Each server has an `active` count initialized to 0.
2. **Request Handling**:

- For each incoming request (total 10 requests), the `distribute` function selects a server using round-robin and creates a new thread to handle the request using the `handle` function.
- The `handle` function updates the `active` count, simulates some processing time with `sleep`, and then updates the `active` count again.

3. **Concurrency**: The use of threads (`threading.Thread`) allows multiple requests to be handled concurrently, with each server processing requests in parallel.

Output Example:

Here's an example of what the output might look like (the exact timing will vary because of the randomness):

```arduino
Server 0 handling 1 | Active: 1
Server 1 handling 2 | Active: 1
Server 2 handling 3 | Active: 1
Server 0 finished 1 | Active: 0
Server 1 finished 2 | Active: 0
Server 2 finished 3 | Active: 0
...
```

This code simulates a round-robin load balancing mechanism for distributing requests to multiple servers and processes them concurrently using threads.

Let me know if you need further clarification!

explain code line by line:
server.py:
import Pyro5.api

@Pyro5.api.expose
class HotelBookingSystem:
    def __init__(self):
        self.bookings = {}

    def book_room(self, guest_name, room_number):
        if room_number in self.bookings:
            return f"Room {room_number} is already booked."

```
        self.bookings[room_number] = guest_name
        return f"Room {room_number} successfully booked for
{guest_name}."

    def cancel_booking(self, guest_name):
        for room, guest in list(self.bookings.items()):
            if guest == guest_name:
                del self.bookings[room]
                return f"Booking for {guest_name} canceled."
        return f"No booking found for {guest_name}."

def main():
    daemon = Pyro5.api.Daemon()
    uri = daemon.register(HotelBookingSystem)
    print("Ready. URI:", uri)
    daemon.requestLoop()

if __name__ == "__main__":
    main()

client.py:
import Pyro5.api

# Replace this URI with the one printed by the server
uri = input("Enter the server URI: ")
hotel = Pyro5.api.Proxy(uri)

while True:
    print("\n1. Book Room\n2. Cancel Booking\n3. Exit")
    choice = input("Enter choice: ")
    if choice == "1":
        name = input("Enter guest name: ")
        room = input("Enter room number: ")
        print(hotel.book_room(name, room))
    elif choice == "2":
        name = input("Enter guest name to cancel: ")
        print(hotel.cancel_booking(name))
    else:
        break
```

Here's a **line-by-line explanation** of the `server.py` and `client.py` code. This code implements a simple hotel booking system using Pyro5 for remote object communication.

🔹 `server.py` (Server Side)

```python
import Pyro5.api
```

- Imports the `Pyro5.api` module, which provides functionality to expose Python objects as remote objects that can be accessed from client code. Pyro5 is a library for building distributed systems.

```python
@Pyro5.api.expose
```

- The `@Pyro5.api.expose` decorator is used to mark the `HotelBookingSystem` class and its methods as remote accessible, making them available for remote invocation by clients.

```python
class HotelBookingSystem:
```

- Defines the `HotelBookingSystem` class, which will hold the methods for booking and canceling hotel room bookings.

```python
    def __init__(self):
        self.bookings = {}
```

- The constructor (`__init__`) initializes an empty dictionary `self.bookings`, which will hold room numbers as keys and guest names as values, representing room bookings.

```python
    def book_room(self, guest_name, room_number):
```

- Defines the `book_room` method, which allows clients to book a room by providing the guest's name and the room number.

```python
        if room_number in self.bookings:
            return f"Room {room_number} is already booked."
```

- Checks if the room is already booked by looking up `room_number` in the `self.bookings` dictionary. If it's booked, it returns a message saying the room is unavailable.

```python
        self.bookings[room_number] = guest_name
        return f"Room {room_number} successfully booked for {guest_name}."
```

- If the room is available, the booking is confirmed by adding the guest's name to the `self.bookings` dictionary. A success message is then returned.

```python
    def cancel_booking(self, guest_name):
```

- Defines the `cancel_booking` method, which allows a client to cancel a booking by providing the guest's name.

```python
        for room, guest in list(self.bookings.items()):
            if guest == guest_name:
                del self.bookings[room]
                return f"Booking for {guest_name} canceled."
```

- Loops through the `self.bookings` dictionary and checks if the guest name matches. If it does, the booking is canceled by deleting the corresponding room from the dictionary, and a cancellation message is returned.

```python
```

```
        return f"No booking found for {guest_name}."
```

- If no booking for the provided guest name is found, a message indicating that no booking was found is returned.

`main()` **Function:**

python

```
def main():
    daemon = Pyro5.api.Daemon()
```

- Creates an instance of the `Daemon` class, which is responsible for handling incoming requests and managing communication between the client and server.

python

```
    uri = daemon.register(HotelBookingSystem)
```

- Registers the `HotelBookingSystem` class as a remote object. This allows the client to access the methods of this class via the URI generated by the daemon.

python

```
    print("Ready. URI:", uri)
```

- Prints the URI of the `HotelBookingSystem` object, which clients will use to access the server.

python

```
    daemon.requestLoop()
```

- Starts the daemon's request loop, which listens for incoming client requests and handles them.

Main Code Block:

python

```
if __name__ == "__main__":
    main()
```

- Ensures that the `main()` function is called when the server script is executed directly.

🔹 `client.py` (Client Side)

python

```
import Pyro5.api
```

- Imports the `Pyro5.api` module to allow the client to interact with the remote object provided by the server.

python

```
uri = input("Enter the server URI: ")
```

- Prompts the user to enter the URI of the server. This URI is provided by the server when it starts, and the client uses it to connect to the remote `HotelBookingSystem`.

python

```
hotel = Pyro5.api.Proxy(uri)
```

- Creates a proxy object `hotel` that connects to the server using the provided URI. This proxy allows the client to call methods on the remote `HotelBookingSystem` object as if it were a local object.

Main Loop:

python

```
while True:
    print("\n1. Book Room\n2. Cancel Booking\n3. Exit")
    choice = input("Enter choice: ")
```

- Displays a simple menu with options to book a room, cancel a booking, or exit. The `choice` input determines which action the client will take.

```python
    if choice == "1":
```

- If the user selects "1" (book a room), the following code executes:

```python
        name = input("Enter guest name: ")
        room = input("Enter room number: ")
        print(hotel.book_room(name, room))
```

- Prompts the user to enter the guest name and room number, then calls the `book_room` method on the remote `hotel` object using the input values. The result is printed to the console.

```python
    elif choice == "2":
```

- If the user selects "2" (cancel a booking), the following code executes:

```python
        name = input("Enter guest name to cancel: ")
        print(hotel.cancel_booking(name))
```

- Prompts the user to enter the guest name for the booking to cancel, then calls the `cancel_booking` method on the remote `hotel` object. The result is printed to the console.

```python
    else:
        break
```

- If the user selects anything other than "1" or "2", the loop breaks, and the program ends.

How This Works:

1. **Server**:
   - The `HotelBookingSystem` class provides two main methods: `book_room` and `cancel_booking`. These methods allow clients to book rooms and cancel bookings remotely.
   - The server uses Pyro5 to expose these methods, making them accessible remotely via a URI.
   - The server listens for requests and processes them through the daemon.
2. **Client**:
   - The client connects to the server using the URI and allows the user to interact with the hotel booking system.
   - The client can choose to book a room, cancel a booking, or exit the program.
   - The client sends requests to the server and prints the server's responses (success or error messages).

Example Workflow:

1. The server is started, and the URI is printed (e.g., `PYRO:obj_xxxxx@localhost:xxxx`).
2. The client enters the server URI to establish a connection.
3. The client can then:
   - Book a room by entering a guest name and room number.
   - Cancel a booking by entering the guest name.
   - Exit the program.

Let me know if you need further clarification!