Assignment 1: RPC Factorial Calculation

Goal: Create a client-server application where the client sends a number to the server, and the server calculates its factorial and sends the result back using Remote Procedure Call (RPC).

Theory Explained Simply

1. **What is RPC?**
    - Imagine you have a calculator (the server) in another room (another computer). You want to calculate 5! (5 factorial).
    - Instead of going to the room, you use a special "remote control" (RPC).
    - You type `calculate_factorial(5)` on your remote (client program).
    - The remote control sends the instruction "calculate_factorial" and the number "5" over a network (like Wi-Fi or cable) to the calculator.
    - The calculator receives the instruction, does the math (5 * 4 * 3 * 2 * 1 = 120).
    - It sends the result "120" back over the network to your remote control.
    - Your remote control displays "120".
    - **RPC makes calling a function on another computer look almost exactly like calling a function on your own computer.** It hides the complex network stuff.
2. **Key Parts (How it Works):**
    - **Client:** The program asking for the calculation.
    - **Server:** The program doing the calculation.
    - **Stubs:** Think of these as translators or messengers.
        - **Client Stub:** Takes your function call (`calculate_factorial(5)`), packs the function name and arguments (5) into a message the network understands (marshalling). Sends it.
        - **Server Stub:** Receives the message, unpacks it (unmarshalling), figures out which *actual* function to call on the server (`factorial(n)`), and calls it with the argument (5).
    - **Network:** How the messages travel between client and server.
    - **Execution:** The server actually runs the `factorial` code.
    - **Response:** The server stub packs the result (120) into a message, sends it back. The client stub unpacks it and gives it to your client program.
3. **Factorial:** The factorial of a non-negative integer 'n' (written as n!) is the product of all positive integers less than or equal to n. (e.g., 5! = 5 * 4 * 3 * 2 * 1 = 120). 0! is defined as 1.

4. **Advantages of RPC:**
    - **Simplicity:** Programmers don't need to write complex network code.
    - **Modularity:** Can build systems with separate client and server parts. Server logic can be reused.
5. **Limitations of RPC:**
    - **Network Issues:** Depends on a stable network. Slow networks = slow calls. Network down = calls fail.
    - **Security:** Basic RPC often isn't secure (data isn't encrypted). Needs extra work.
    - **Error Handling:** Handling network errors is trickier than local errors.

Practical Steps (using Python `xmlrpc`)

The code uses Python's built-in `xmlrpc` library, which is a simple way to implement RPC.

Step 1: Write the Server Code (`server.py`)

```
# server.py
from xmlrpc.server import SimpleXMLRPCServer
import threading
import math # Using math.factorial for efficiency, or implement recursive

# The actual function the server will perform
def factorial_calc(n):
    print(f"Received request to calculate factorial for: {n}")
    if not isinstance(n, int) or n < 0:
        # Raise an error that XML-RPC can transmit
        raise ValueError("Factorial not defined for non-negative integers
    try:
        result = math.factorial(n) # Or your recursive/iterative functior
        print(f"Calculated factorial: {result}")
        return result
    except Exception as e:
        print(f"Error during calculation: {e}")
        raise # Re-raise the exception

# A function to allow the client to shut down the server gracefully
def shutdown_server():
    print("Shutdown request received. Shutting down server...")
    # Run the server shutdown in a separate thread
    # so this function can return the confirmation message
    threading.Thread(target=server.shutdown).start()
```

```
    return "Server is shutting down."

# Create the server instance
# Listens on localhost (127.0.0.1) at port 8000
# allow_none=True allows methods to potentially return None (not used her
server_address = ('localhost', 8000)
server = SimpleXMLRPCServer(server_address, allow_none=True)
print(f"Server listening on {server_address[0]}:{server_address[1]}...")

# Register the functions that the client can call
# server.register_function(function_object, 'name_client_uses')
server.register_function(factorial_calc, 'factorial')
server.register_function(shutdown_server, 'shutdown')

# Start the server's main loop to listen for requests
try:
    server.serve_forever()
except KeyboardInterrupt:
    print("\nServer shutting down (Keyboard Interrupt).")
    server.shutdown()
```

- **Why `SimpleXMLRPCServer`?** It's a ready-made class in Python to easily create an XML-RPC server.
- **Why `('localhost', 8000)`?** Tells the server to listen only for connections from the same machine (`localhost`) on port `8000`. Port 8000 is just a common port used for development; any available port could be used.
- **Why `factorial_calc(n)`?** This is the core logic the server performs. We added input validation.
- **Why `register_function`?** This makes the Python functions (`factorial_calc`, `shutdown_server`) available to be called remotely via RPC, using the specified names ('factorial', 'shutdown').
- **Why `serve_forever()`?** Starts the server listening for incoming RPC requests. It runs until stopped.
- **Why `shutdown_server` and `threading`?** Allows the client to tell the server to stop. `threading` is used so the `shutdown` function can send a confirmation message *back* to the client before the server actually stops listening (which blocks).

Step 2: Write the Client Code (`client.py`)

```
# client.py
import xmlrpc.client
```

```
# Create a proxy object that represents the remote server
# Connects to the server URL
server_url = "http://localhost:8000/"
try:
    proxy = xmlrpc.client.ServerProxy(server_url)
    print(f"Connected to server at {server_url}")
except ConnectionRefusedError:
    print(f"Error: Connection refused. Is the server running at {server_u
    exit()
except Exception as e:
    print(f"An error occurred connecting to the server: {e}")
    exit()


while True:
    user_input = input("Enter a non-negative integer for factorial (or ty

    if user_input.lower() == 'exit':
        try:
            print("Sending shutdown request to server...")
            shutdown_message = proxy.shutdown() # Call the remote shutdow
            print(f"Server response: {shutdown_message}")
        except xmlrpc.client.Fault as err:
             print(f"A fault occurred trying to shut down: {err.faultStri
        except Exception as e:
            # This might happen if the server shut down before response,
            print(f"Could not communicate with server for shutdown (maybe
        finally:
            print("Exiting client.")
            break # Exit the client loop

    try:
        num = int(user_input) # Convert input to integer
        if num < 0:
            print("Factorial is not defined for negative numbers.")
            continue

        # This is the actual Remote Procedure Call!
        result = proxy.factorial(num)
        print(f"Factorial of {num} is {result}")

    except ValueError:
        print("Invalid input. Please enter an integer or 'exit'.")
    except xmlrpc.client.Fault as err:
```

```
        # Catches errors raised by the server function (like our ValueErr
        print(f"Server Fault: {err.faultString}")
    except Exception as e:
        # Catches other potential errors (network, server unavailable)
        print(f"An error occurred during RPC call: {e}")
        # Optionally, try to reconnect or exit
        # print("Attempting to reconnect or exit...")
        # break # Or implement reconnection logic
```

- **Why `xmlrpc.client.ServerProxy`?** Creates an object (`proxy`) that acts as a stand-in for the remote server. You call methods on this object.
- **Why `proxy.factorial(num)`?** This is the magic! It *looks* like a normal function call, but `ServerProxy` turns it into an RPC request, sends it to the server URL, waits for the response, and returns the result.
- **Why `proxy.shutdown()`?** Calls the remote function named 'shutdown' that we registered on the server.
- **Why `try...except` blocks?** Essential for handling errors: user entering non-numbers (`ValueError`), server functions raising errors (`xmlrpc.client.Fault`), network problems or server down (`ConnectionRefusedError`, other `Exception`s).

Step 3: Run the Practical

1. Open **Terminal 1**.
2. Navigate to the directory where you saved the files.
3. Run the server: `python server.py`
    - You should see: `Server listening on localhost:8000...`
4. Open **Terminal 2**.
5. Navigate to the same directory.
6. Run the client: `python client.py`
    - You should see: `Connected to server at http://localhost:8000/`
    - Then the prompt: `Enter a non-negative integer...`
7. In Terminal 2 (Client):
    - Enter `5`. Output: `Factorial of 5 is 120`. (Check Terminal 1 for server logs).
    - Enter `0`. Output: `Factorial of 0 is 1`.
    - Enter `-3`. Output: `Factorial is not defined for negative numbers.`
    - Enter `abc`. Output: `Invalid input...`
    - Enter `exit`. Output: `Sending shutdown request...`, `Server response: Server is shutting down.`, `Exiting client.` (Check Terminal 1 - server should stop).

Potential Viva Questions & Answers

1. **Q: What is RPC?**
   - **A:** Remote Procedure Call. It's a way for a program on one computer to call a function or procedure on another computer over a network, making it seem like a local call.
2. **Q: Explain the role of the client and server stubs.**
   - **A:** The client stub takes the function call details, packs them into a network message (marshalling), and sends it. The server stub receives the message, unpacks it (unmarshalling), calls the actual server function, gets the result, packs it, and sends it back. They handle the communication details.
3. **Q: What is marshalling/unmarshalling?**
   - **A:** Marshalling is converting data (like function arguments or results) into a format suitable for network transmission (e.g., XML, JSON, binary). Unmarshalling is converting it back from the network format into usable data in the receiving program.
4. **Q: In the Python code, which line performs the actual RPC?**
   - **A:** In `client.py`, the line `result = proxy.factorial(num)` performs the call to the remote 'factorial' function. Similarly, `proxy.shutdown()` calls the remote 'shutdown'.
5. **Q: What does `server.register_function(factorial_calc, 'factorial')` do?**
   - **A:** It tells the XML-RPC server that the Python function `factorial_calc` should be executed when a client calls the remote procedure named 'factorial'.
6. **Q: Why did we use `threading` in the `shutdown_server` function?**
   - **A:** The `server.shutdown()` method stops the server, which can block. To allow the server to send a confirmation message ("Server is shutting down.") *back* to the client *before* it stops listening, we run `server.shutdown()` in a separate thread.
7. **Q: What happens if the server is not running when the client starts?**
   - **A:** The `xmlrpc.client.ServerProxy(server_url)` call or the first RPC call (`proxy.factorial`) will likely fail, raising an error like `ConnectionRefusedError`. The client's `try...except` block should catch this.
8. **Q: What are the advantages of using RPC for this factorial problem?**
   - **A:** It separates the calculation logic (server) from the user interaction (client). The client doesn't need to know *how* factorial is calculated. If the calculation was very complex, it offloads work from the client.
9. **Q: What are the disadvantages or potential problems?**
   - **A:** Network latency can make the call slower than a local calculation. Network errors can cause failures. Security isn't built-in with basic XML-RPC.
10. **Q: Can the server handle multiple clients at once?**
    - **A:** `SimpleXMLRPCServer` is typically single-threaded, meaning it handles requests one after another. For concurrent handling, you might need `ForkingXMLRPCServer` or `ThreadingXMLRPCServer` (or other frameworks).

Assignment 2: RMI String Concatenation

Goal: Create a Java client-server application where the client sends two strings to the server, and the server concatenates them and returns the result using Remote Method Invocation (RMI).

Theory Explained Simply

1. **What is RMI?**
    - Remote Method Invocation is **Java's specific way of doing RPC**.
    - It's more integrated with the Java language and focuses on calling *methods* on remote *objects*.
    - Think of it like having a remote Java object that you can use almost like a local one.
    - Client needs to find this remote object first, then it can call its methods.
2. **Key Parts (How it Works):**
    - **Remote Interface:** A Java `interface` that declares the methods that can be called remotely. Both client and server need access to this interface definition. It acts as a contract. *Must extend* `java.rmi.Remote`.
    - **Remote Object Implementation (Server-side):** A Java class that *implements* the Remote Interface and contains the actual code for the remote methods. *Often extends* `java.rmi.server.UnicastRemoteObject`.
    - **RMI Registry:** A simple naming service (like a phonebook). The server "registers" its remote object with the registry under a specific name (e.g., "ConcatService"). Clients "look up" this name in the registry to get a reference (stub) to the remote object. *Needs to be run as a separate process.*
    - **Client:** The Java program that looks up the remote object in the registry and then calls methods on it.
    - **Stub:** (Client-side proxy) An object on the client that represents the remote object. When the client calls a method on the stub, the stub handles the network communication to the actual remote object. (Generated automatically by RMI).
    - **Skeleton:** (Server-side helper) Receives the request from the stub, calls the appropriate method on the actual server object, and sends the result back. (Also handled automatically by RMI).
    - **RemoteException:** RMI methods must declare they can throw `java.rmi.RemoteException` to signal potential network or remote errors.
3. **String Concatenation:** Simply joining two strings together (e.g., "Hello" + "World" becomes "HelloWorld").
4. **Advantages of RMI:**

- **Object-Oriented:** Works naturally with Java objects.
- **Seamless:** Remote method calls look very similar to local ones in Java code.
- **Automatic Stubs/Skeletons:** RMI handles generating the communication code.
- **Type Safety:** Uses Java's type system.

5. **Limitations of RMI:**

- **Java Only:** Generally only works between Java Virtual Machines (JVMs).
- **Tight Coupling:** Client and server often need shared interface classes.
- **Performance:** Can be slower than local calls due to network and serialization overhead.
- **Setup:** Requires running the RMI Registry.

Practical Steps (using Java RMI)

Step 1: Define the Remote Interface (`ConcatService.java`)

```
// ConcatService.java
import java.rmi.Remote;
import java.rmi.RemoteException;

// 1. Must extend java.rmi.Remote (marker interface)
public interface ConcatService extends Remote {
    // 2. Each method must declare throwing java.rmi.RemoteException
    String concatenate(String str1, String str2) throws RemoteException;
}
```

- **Why `extends Remote` ?** This tells Java that methods in this interface might be called remotely.
- **Why `throws RemoteException` ?** Any RMI call can fail due to network issues, so methods must declare this potential exception.

Step 2: Implement the Remote Object (`ConcatServer.java`)

```
// ConcatServer.java
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.registry.LocateRegistry; // Needed to create registry if

public class ConcatServer extends UnicastRemoteObject implements ConcatSe

    // 1. Constructor must also declare RemoteException
    protected ConcatServer() throws RemoteException {
```

```
        super(); // Call the UnicastRemoteObject constructor
    }

    // 2. Implement the method from the interface
    @Override
    public String concatenate(String str1, String str2) throws RemoteExce
        System.out.println("Received request: concatenate(\"" + str1 + "\
        String result = str1 + str2;
        System.out.println("Returning result: \"" + result + "\"");
        return result;
    }

    public static void main(String[] args) {
        try {
            // Optional: Start RMI registry programmatically if not runni
            try {
                LocateRegistry.createRegistry(1099); // Default RMI port
                System.out.println("RMI registry created on port 1099.");
            } catch (RemoteException e) {
                // Registry might already be running
                System.out.println("RMI registry might already be running
            }

            // 3. Create an instance of the remote object implementation
            ConcatServer server = new ConcatServer();
            System.out.println("ConcatServer object created.");

            // 4. Bind the remote object to the RMI registry with a name
            // URL format: rmi://hostname:port/objectname
            Naming.rebind("rmi://localhost:1099/ConcatService", server);
            // Naming.rebind("rmi://localhost/ConcatService", server); //

            System.out.println("ConcatService bound in registry. Server i

        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

- **Why `extends UnicastRemoteObject`?** This class provides the necessary functionality for a standard RMI object that listens on a TCP port for incoming requests.

- **Why `implements ConcatService`?** The server *must* provide the actual code for the methods defined in the remote interface.
- **Why constructor throwing `RemoteException`?** The superclass (`UnicastRemoteObject`) constructor can throw this, e.g., if it fails to export the object (make it available for remote calls).
- **Why `LocateRegistry.createRegistry(1099)`?** This line attempts to start the RMI registry service within the server program itself on the default port 1099. This is convenient but often the registry is run separately.
- **Why `Naming.rebind(...)`?** This registers the `server` object with the RMI registry running on `localhost` (port 1099 is default), under the name "ConcatService". Clients will use this name to find the object. `rebind` will overwrite any existing binding with that name.

Step 3: Write the Client Code (`ConcatClient.java`)

```
// ConcatClient.java
import java.rmi.Naming;
import java.util.Scanner;

public class ConcatClient {
    public static void main(String[] args) {
        try {
            // 1. Look up the remote object in the RMI registry
            // Use the same name the server used to bind
            ConcatService service = (ConcatService) Naming.lookup("rmi://
            // ConcatService service = (ConcatService) Naming.lookup("rmi

            System.out.println("Successfully looked up ConcatService.");

            Scanner scanner = new Scanner(System.in);

            while (true) {
                System.out.println("\nMenu:");
                System.out.println("1. Concatenate Strings");
                System.out.println("2. Exit");
                System.out.print("Enter choice: ");
                int choice = scanner.nextInt();
                scanner.nextLine(); // Consume newline

                if (choice == 1) {
                    System.out.print("Enter first string: ");
                    String s1 = scanner.nextLine();
                    System.out.print("Enter second string: ");
                    String s2 = scanner.nextLine();
```

```
                    // 2. Call the remote method just like a local methoc
                    String result = service.concatenate(s1, s2);

                    System.out.println("Concatenated Result: " + result);
                } else if (choice == 2) {
                    System.out.println("Exiting...");
                    break;
                } else {
                    System.out.println("Invalid choice. Please try again.
                }
            }
            scanner.close();

        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
            System.err.println("\nMake sure the RMI Registry and ConcatSe
        }
    }
}
```

- **Why `Naming.lookup(...)` ?** This contacts the RMI registry at the specified URL and asks for the object bound to the name "ConcatService". It returns a stub object.
- **Why `(ConcatService)` cast?** `Naming.lookup` returns a generic `Remote` object. We cast it to our specific interface (`ConcatService`) so we can call its methods (`concatenate`).
- **Why `service.concatenate(s1, s2)` ?** This looks like a local call, but RMI uses the stub (`service`) to send the request to the remote server object, get the result, and return it.
- **Why `try...catch` ?** To handle potential errors during lookup (registry not running, object not bound) or during the remote call (`RemoteException` , server down).

Step 4: Compile and Run the Practical

1. Save the three files (`ConcatService.java` , `ConcatServer.java` , `ConcatClient.java`) in the same directory.
2. Open a terminal or command prompt in that directory.
3. **Compile:** `javac *.java`
    - This creates `.class` files for all three source files.
4. **Start RMI Registry (Option A - Separate Terminal):**
    - Open a **NEW** terminal (Terminal 1).
    - Navigate to the **directory containing the `.class` files**.
    - Run: `rmiregistry`
    - Leave this terminal running. It doesn't usually show output unless there's an error.

- *(Note: If Option B was used in Server code, this step is not needed)*

5. **Run the Server:**
   - Open another **NEW** terminal (Terminal 2).
   - Navigate to the directory containing the `.class` files.
   - Run: `java ConcatServer`
   - You should see output like "ConcatServer object created.", "RMI registry...", "ConcatService bound...", "Server is running...".
6. **Run the Client:**
   - Open another **NEW** terminal (Terminal 3).
   - Navigate to the directory containing the `.class` files.
   - Run: `java ConcatClient`
   - You should see "Successfully looked up ConcatService." and the menu.
7. **Interact:** Use the client menu to concatenate strings (e.g., enter '1', then "Hello", then " World"). Observe the result in the client terminal and the log messages in the server terminal. Choose '2' to exit the client.
8. **Stop:** Close the client (if not exited). Stop the server (Ctrl+C in Terminal 2). Stop the RMI registry (Ctrl+C in Terminal 1, if run separately).

Potential Viva Questions & Answers

1. **Q: What is RMI and how is it different from RPC?**
   - **A:** RMI (Remote Method Invocation) is Java's built-in mechanism for distributed computing. It's like RPC but specifically designed for Java objects. It allows calling methods on objects located in different Java Virtual Machines (JVMs). RPC is a more general concept, while RMI is Java-specific and object-oriented.
2. **Q: What are the main components of an RMI application?**
   - **A:** Remote Interface (defines methods), Remote Object Implementation (server-side logic), RMI Registry (naming service), Client (looks up and calls methods), Stubs/Skeletons (handle communication, usually transparent).
3. **Q: What is the role of the RMI Registry?**
   - **A:** It acts like a directory or phonebook. The server registers its remote object with the registry under a name. Clients query the registry using that name to get a reference (stub) to the remote object.
4. **Q: Why does the remote interface need to extend `java.rmi.Remote`?**
   - **A:** `Remote` is a marker interface. It doesn't have methods, but it signals to the JVM that the methods declared in the implementing interface can be invoked remotely.
5. **Q: Why do remote methods need to declare `throws RemoteException`?**
   - **A:** Because remote calls involve network communication, many things can go wrong (network down, server unavailable, errors during data transfer). Declaring `RemoteException` forces the caller (client) to handle these potential issues.

6. **Q: What does `UnicastRemoteObject` provide?**
    - **A:** It provides the basic functionality needed for a remote object, such as listening for connections on a TCP port, exporting the object (making it available for remote calls), and handling remote reference management.
7. **Q: What is the difference between `Naming.bind` and `Naming.rebind`?**
    - **A:** `bind` registers an object with the registry. It fails if a binding with the same name already exists. `rebind` also registers an object, but it will *replace* any existing binding with the same name. `rebind` is often used in development for convenience.
8. **Q: What happens if the RMI Registry is not running when the server tries to `rebind`?**
    - **A:** The server will throw an exception (likely a `ConnectException` or similar `RemoteException`), because it cannot contact the registry service.
9. **Q: What happens if the server is not running (or the object isn't bound) when the client tries to `lookup`?**
    - **A:** The client's `Naming.lookup` call will throw an exception (likely `NotBoundException` if the name isn't found, or `ConnectException` if the registry/server host is unreachable).
10. **Q: Can RMI work between a Java application and a Python application?**
    - **A:** Not directly. RMI is primarily designed for Java-to-Java communication. For interoperability between different languages, you'd typically use other technologies like web services (REST, SOAP) or a more general RPC framework (like gRPC, Thrift, or even XML-RPC as in Assignment 1).

Assignment 3: MapReduce Word/Character Count

Goal: Use the MapReduce programming model (with Hadoop) to count: a) the occurrences of each character, and b) the occurrences of each word in a given text file. This assignment uses Hadoop Streaming with Python for the character count.

Theory Explained Simply

1. **What is MapReduce?**
    - A programming model for processing *very large* datasets (too big for one computer) across a *cluster* of computers (a Hadoop cluster).
    - It breaks the problem into two main phases: **Map** and **Reduce**.
2. **How it Works:**
    - **Input Splitting:** Hadoop automatically splits the large input file(s) into smaller chunks. Each chunk is processed by a separate Map task.

- **Map Phase:**
  - Mapper tasks read their assigned input chunk (line by line, or record by record).
  - For each input element, the mapper applies a function and outputs zero or more **key-value pairs**.
  - *Example (Word Count):* Input line "hello world hello". Mapper outputs: `(hello, 1)`, `(world, 1)`, `(hello, 1)`.
  - *Example (Character Count):* Input line "abc". Mapper outputs: `(a, 1)`, `(b, 1)`, `(c, 1)`.
- **Shuffle and Sort Phase (Automatic):**
  - This is the magic performed by the Hadoop framework *between* Map and Reduce.
  - It collects all the key-value pairs output by *all* the mappers.
  - It sorts these pairs based on the key.
  - It groups all the values associated with the *same key* together and sends them to a single Reducer task.
  - *Example (Word Count):* Input to Reducer for key "hello": `(hello, [1, 1])`. Input for key "world": `(world, [1])`.
  - *Example (Character Count):* Input to Reducer for key "a": `(a, [1])`. Input for key "b": `(b, [1])`. Input for key "c": `(c, [1])`.
- **Reduce Phase:**
  - Reducer tasks receive input as `(key, list_of_values)`.
  - For each key, the reducer applies a function to the list of values to produce the final output (usually zero or one output pair per key).
  - *Example (Word Count):* Input `(hello, [1, 1])`. Reducer sums the values: `1 + 1 = 2`. Outputs: `(hello, 2)`. Input `(world, [1])`. Outputs: `(world, 1)`.
  - *Example (Character Count):* Input `(a, [1])`. Outputs: `(a, 1)`. Input `(b, [1])`. Outputs: `(b, 1)`. Input `(c, [1])`. Outputs: `(c, 1)`. (If input was "aba", reducer gets `(a, [1, 1])` -> outputs `(a, 2)`)
- **Output:** The outputs from all reducer tasks are collected and typically written to files in a specified HDFS directory.

3. **Hadoop:** An open-source framework for distributed storage (HDFS) and distributed processing (MapReduce, YARN).
   - **HDFS (Hadoop Distributed File System):** Stores huge files across many machines, providing reliability.
   - **YARN (Yet Another Resource Negotiator):** Manages the cluster resources (CPU, memory) and schedules tasks (like Map and Reduce tasks).
4. **Hadoop Streaming:** A utility that allows you to use *any* executable (like Python scripts, shell scripts) as your Mapper and Reducer, instead of being limited to Java. It works by passing data between Hadoop and your scripts via standard input (stdin) and standard output (stdout).

Practical Steps (Focusing on Character Count with Python)

(Note: The Word Count part in the PDF uses a pre-built Java example within Hadoop, not custom Python scripts, so we'll focus on the Character Count which uses the provided Python code.)

Step 1: Prepare the Environment

1. **Log in as Hadoop User:** `su hduser` (or whatever your Hadoop user is). Enter password.
2. **Navigate to Home (Optional):** `cd`
3. **Create Mapper Script (`mapper.py`):**
   - `nano mapper.py`
   - Paste the `mapper.py` code from the PDF.
   - Save (Ctrl+O, Enter) and Exit (Ctrl+X).
4. **Create Reducer Script (`reducer.py`):**
   - `nano reducer.py`
   - Paste the `reducer.py` code from the PDF.
   - Save and Exit.
5. **Make Scripts Executable:**
   - `chmod +x mapper.py`
   - `chmod +x reducer.py`
   - **Why `chmod +x` ?** Hadoop needs permission to execute these script files.

Step 2: Start Hadoop Services

1. **Start HDFS:** `start-dfs.sh`
2. **Start YARN:** `start-yarn.sh`
3. **Check Daemons:** `jps`
   - You should see processes like `NameNode` , `DataNode` , `SecondaryNameNode` , `ResourceManager` , `NodeManager` . If any are missing, there might be a configuration issue.
   - **Why `jps` ?** To confirm that the essential Hadoop background processes (daemons) needed for storage and processing are running.

Step 3: Prepare Input Data in HDFS

1. **Create an Input Text File (Local):**
   - `nano character_count.txt`
   - Type some text, e.g.:

     ```
     abbc
     ccba
     ```

```
aacb
hello world
```

- Save and Exit.

2. **Remove Old HDFS Directories (Important!)**: MapReduce jobs usually fail if the output directory already exists.
   - `hdfs dfs -rm -r /input` (If it exists from previous runs)
   - `hdfs dfs -rm -r /output` (If it exists from previous runs)
   - *(Use with caution!)*
3. **Create HDFS Input Directory:** `hdfs dfs -mkdir -p /input`
   - **Why HDFS?** MapReduce operates on data stored in Hadoop's distributed file system.
   - **Why `/input`?** Just a conventional name for the input directory in HDFS.
4. **Upload Input File to HDFS:** `hdfs dfs -put character_count.txt /input/`
   - **Why `hdfs dfs -put`?** Copies the local file (`character_count.txt`) into the HDFS directory (`/input`).

Step 4: Run the MapReduce Job (Hadoop Streaming)

1. **Find Hadoop Streaming JAR:** You need the path to `hadoop-streaming-*.jar`. The PDF uses `/usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-3.3.4.jar`. Your path might differ. Use `find /usr/local/hadoop -name "hadoop-streaming*.jar"` if unsure.
2. **Execute the Command:**

```
hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-
-input /input/character_count.txt \
-output /output/character_output \
-mapper mapper.py \
-reducer reducer.py \
-file mapper.py \
-file reducer.py
```

   - **Why `hadoop jar ...`?** This is the command to run a Hadoop job packaged in a JAR file (here, the streaming JAR).
   - **Why `-input /input/...`?** Specifies the input path in HDFS.
   - **Why `-output /output/...`?** Specifies the *new* directory in HDFS where results will be stored. **Must not exist before running.**
   - **Why `-mapper mapper.py`?** Tells streaming to use `mapper.py` for the Map phase.
   - **Why `-reducer reducer.py`?** Tells streaming to use `reducer.py` for the Reduce phase.

- **Why `-file mapper.py -file reducer.py`?** This is crucial! It tells Hadoop to ship these local script files to all the cluster nodes that will run the map/reduce tasks, so they can be executed there.

Step 5: Check the Output

1. **List Output Directory Contents:** `hdfs dfs -ls /output/character_output`
   - You should see a `_SUCCESS` file (if the job finished successfully) and one or more `part-r-xxxxx` files (the actual output from the reducers). Usually `part-r-00000`.
2. **View the Output:** `hdfs dfs -cat /output/character_output/part-r-00000`
   - You should see the character counts, one per line, tab-separated (e.g., `a 5`, `b 4`, `c 4`, `d 1`, etc., based on the input file).
   - **Why `part-r-00000`?** Default naming convention for output files from reducers. If you had multiple reducers, you'd see `part-r-00001`, etc.

Step 6: Stop Hadoop Services

1. **Stop YARN:** `stop-yarn.sh`
2. **Stop HDFS:** `stop-dfs.sh`

Potential Viva Questions & Answers

1. **Q: Explain the Map and Reduce phases in the context of the Character Count.**
   - **A: Map:** Reads the input text line by line. For each character in a line, it outputs the character as the key and the number 1 as the value (e.g., 'a' -> `a 1`). **Reduce:** Receives a character (key) and a list of all the 1s associated with it (e.g., 'a', `[1, 1, 1, 1, 1]`). It sums up the list of 1s to get the total count for that character and outputs the character and its total count (e.g., `a 5`).
2. **Q: What does the Shuffle and Sort phase do? Why is it important?**
   - **A:** It happens automatically between Map and Reduce. It collects all the (character, 1) pairs from all mappers, sorts them by character, and groups the 1s for the same character together. It's crucial because it ensures that all counts for a single character arrive at the *same* reducer task, so the reducer can calculate the correct total sum.
3. **Q: What is Hadoop Streaming?**
   - **A:** It's a Hadoop utility that allows using any executable script (like Python, Perl, bash) as the mapper and/or reducer, communicating via standard input and output, instead of requiring Java code.
4. **Q: In the streaming command, what is the purpose of the `-file` option?**
   - **A:** It tells Hadoop to copy the specified local files (our `mapper.py` and `reducer.py` scripts) to the cluster nodes where the tasks will run. Without `-file`, the nodes wouldn't have the scripts to execute.

5. **Q: Why do the Python scripts read from `sys.stdin` and write to `sys.stdout`?**
    - **A:** Because that's how Hadoop Streaming communicates with the scripts. Hadoop feeds the input data to the script's standard input, and it reads the script's standard output to get the key-value pairs.
6. **Q: What format does Hadoop Streaming expect for the output of the mapper and the input of the reducer?**
    - **A:** By default, it expects key-value pairs separated by a tab character (`\t`) on each line. Our Python scripts adhere to this (e.g., `print(f"{char}\t1")`).
7. **Q: What does the `part-r-00000` file contain?**
    - **A:** It contains the final output generated by the reducer task(s). In this case, the character counts.
8. **Q: Why do we need to delete the output directory before running the job again?**
    - **A:** Hadoop MapReduce is designed to prevent accidental data loss. By default, it refuses to write output to a directory that already exists.
9. **Q: How would the Mapper change for Word Count instead of Character Count?**
    - **A:** The Word Count mapper would split each line into *words* (e.g., using spaces as delimiters) instead of characters. For each word, it would output `word\t1`.
10. **Q: What are HDFS and YARN?**
    - **A:** HDFS (Hadoop Distributed File System) is the storage part of Hadoop, designed to store very large files across multiple machines. YARN (Yet Another Resource Negotiator) is the resource management and job scheduling part, deciding where and when to run MapReduce tasks on the cluster nodes.

Assignment 4: Load Balancing Simulation

Goal: Simulate different load balancing algorithms (Round Robin, Weighted Round Robin, Least Connections, Least Response Time) distributing client requests among a set of servers using Python.

Theory Explained Simply

1. **What is Load Balancing?**
    - Imagine a popular website with many visitors. If all visitors connect to a single server, it might get overloaded and become slow or crash.
    - Load balancing is like having a smart traffic director at the entrance. It distributes incoming visitor requests across *multiple* servers.
    - **Goals:** Prevent server overload, improve response time, ensure the website stays available even if one server fails, make it easy to add more servers (scalability).
2. **Load Balancing Algorithms (Methods):**

- **Round Robin:** Simplest method. Sends requests to servers in a cycle: Server1, Server2, Server3, Server1, Server2, Server3, ...
  - *Pro:* Even distribution. Easy.
  - *Con:* Doesn't care if Server1 is super busy or Server2 is idle, or if Server3 is much weaker.
- **Weighted Round Robin:** Like Round Robin, but smarter if servers have different capacities. You assign "weights" (e.g., Server1 weight 5, Server2 weight 1, Server3 weight 1). Server1 will get 5 requests for every 1 request Server2 and Server3 get.
  - *Pro:* Better utilizes servers with different power.
  - *Con:* Still doesn't consider real-time load. Weights might need tuning.
- **Least Connections:** Sends the next request to the server that currently has the *fewest active connections*.
  - *Pro:* Dynamic, adapts to current load. Prevents overloading one server.
  - *Con:* Needs to track connections accurately. Assumes all connections are equal (a complex request counts the same as a simple one).
- **Least Response Time:** Sends the next request to the server that is currently responding the *fastest* (lowest latency). Often combined with Least Connections.
  - *Pro:* Aims for best user experience (speed). Very dynamic.
  - *Con:* Requires constantly monitoring server response times, which adds overhead. Can be sensitive to temporary network fluctuations.

Practical Steps (using Python Simulation)

Step 1: Write the Simulation Code

- Copy the entire Python code block from the PDF (containing all four classes: `RoundRobin`, `WeightedRoundRobin`, `LeastConnections`, `LeastResponseTime`, and the helper functions `simulate_response_time`, `demonstrate_algorithm`, plus the main execution block) into a single file named `load_balancer.py`.

- **Code Explanation:**

  - **Classes:** Each class (`RoundRobin`, etc.) represents one load balancing algorithm. It holds the list of servers and any internal state needed (like `current_index`, `weights`, connection counts, response times).
  - `__init__`: The constructor initializes the load balancer with the list of servers and any specific parameters (like weights).
  - `get_next_server()`: This is the core method for each class. It implements the logic of that specific algorithm to choose which server should handle the *next* incoming request.
  - `release_connection()` **(in** `LeastConnections`**):** Simulates a request finishing, reducing the connection count for that server.

- `update_response_time()` **(in** `LeastResponseTime`**):** Updates the recorded response time for a server after a request.
- `simulate_response_time()` **:** A helper function that pauses for a random short time to mimic network/server processing delay.
- `demonstrate_algorithm()` **:** A general function to run the simulation for any given load balancer object, printing which server gets each request and optionally showing response times or simulating connection releases.
- `if __name__ == "__main__":` **block:** This is where the simulation is set up and run. It defines the servers, creates instances of each load balancer class, and calls `demonstrate_algorithm` to show each one in action.

Step 2: Run the Simulation

1. Open a terminal.
2. Navigate to the directory where you saved `load_balancer.py`.
3. Run the script: `python load_balancer.py`

Step 3: Observe the Output

- The script will print the output for each algorithm sequentially.
  - **Round Robin:** You'll see requests cycling through Server1, Server2, Server3, Server1, ...
  - **Weighted Round Robin:** You'll see Server1 getting significantly more requests (5 times more in this setup) than Server2 and Server3 over the 7 iterations shown.
  - **Least Connections:** The output will vary slightly due to the random choice when ties occur. It simulates sending to the server with 0 connections, then incrementing its count. `release_connection` is called after each assignment in the demo, keeping counts low.
  - **Least Response Time:** You'll see requests going to different servers based on the simulated random response times calculated after each request. The server with the lowest simulated time gets the next request.

Potential Viva Questions & Answers

1. **Q: What is the main purpose of load balancing?**
   - **A:** To distribute incoming traffic or workload across multiple servers or resources, preventing any single resource from being overwhelmed, improving performance, reliability, and scalability.
2. **Q: Explain the Round Robin algorithm. What is its main drawback?**
   - **A:** It distributes requests sequentially to each server in a list, returning to the start after reaching the end. It's simple and ensures even distribution. Its main drawback is that it doesn't consider the actual load or capacity of the servers; a slow or overloaded server gets requests just as often as a fast or idle one.
3. **Q: How does Weighted Round Robin improve on Round Robin?**

- **A:** It allows assigning different weights to servers based on their capacity (e.g., CPU power, memory). Servers with higher weights receive proportionally more requests, making better use of heterogeneous hardware.

4. **Q: Explain the Least Connections algorithm.**
   - **A:** It directs new requests to the server that currently has the fewest active connections. This helps adapt to real-time load, preventing servers from becoming overloaded with too many concurrent requests.
5. **Q: When might Least Connections not be optimal?**
   - **A:** If servers have vastly different processing power, the one with fewer connections might still be slower if it has computationally intensive tasks. Also, it assumes all connections represent equal load.
6. **Q: How does the Least Response Time algorithm work?**
   - **A:** It sends requests to the server that is currently responding the fastest (lowest measured latency). This directly targets improving user experience by minimizing wait times.
7. **Q: What are the challenges of implementing Least Response Time in a real system?**
   - **A:** It requires continuous monitoring of server health and response times, which adds complexity and overhead. Response times can fluctuate rapidly due to temporary issues, potentially causing frequent shifts in traffic (rebalancing).
8. **Q: In the Python code, how does the `LeastConnections` class track connections?**
   - **A:** It uses a dictionary (`self.servers`) where keys are server names and values are the current count of active connections for that server. `get_next_server` increments the count, and `release_connection` decrements it.
9. **Q: Is this Python script a real load balancer?**
   - **A:** No, it's a simulation. A real load balancer sits in a network, intercepts actual network traffic (like HTTP requests), communicates with backend servers, performs health checks, and forwards traffic based on the chosen algorithm. This script just simulates the decision-making logic.
10. **Q: Which algorithm demonstrated in the code is the most dynamic in adapting to server conditions?**
    - **A:** Least Response Time is generally the most dynamic as it directly uses recent performance metrics. Least Connections is also dynamic but based on connection count rather than speed. Round Robin and Weighted Round Robin are static (or semi-static if weights change).

Assignment 5: MapReduce Weather Data Analysis

Goal: Use MapReduce (via Hadoop Streaming and Python) to process weather data (containing date, min temperature, max temperature) to find the minimum and maximum temperature recorded for each year. (Finding the overall hottest/coolest year is a post-processing step on the output).

Theory Explained Simply

(This repeats much of Assignment 3's MapReduce theory, applied to a different problem).

1. **The Problem:** Given a dataset with daily min/max temperatures over many years, find the lowest minimum temperature and the highest maximum temperature *for each year*.
2. **MapReduce Approach:**
    - **Map Phase:**
        - Read each line of weather data (e.g., `YYYY-MM-DD minT maxT`).
        - Extract the **Year** from the date.
        - Extract the **min Temperature** and **max Temperature**.
        - Output a key-value pair where the key is the **Year** and the value contains the **minT** and **maxT**. The Python code outputs this as `Year minT maxT` (space-separated) on one line. *Conceptually:* `(Year, (minT, maxT))`.
    - **Shuffle and Sort Phase (Automatic):**
        - Hadoop collects all outputs from the mappers.
        - It sorts and groups them by **Year**.
        - All temperature pairs for a specific year are sent to the *same* Reducer task.
        - *Example:* Input to Reducer for key "1950": `(1950, [(minT1, maxT1), (minT2, maxT2), ...])`. (The actual streaming input is line-by-line `1950 minT1 maxT1`, `1950 minT2 maxT2`, ...)
    - **Reduce Phase:**
        - The Reducer receives the **Year** (key) and all the associated temperature pairs (values).
        - It iterates through all the `minT` values for that year and finds the overall minimum.
        - It iterates through all the `maxT` values for that year and finds the overall maximum.
        - It outputs the **Year**, the **overall min temperature for that year**, and the **overall max temperature for that year**. The Python code outputs this tab-separated: `Year\tmin_for_year\tmax_for_year`.
    - **Final Output (Post-Processing):** The result file from the Reducer(s) contains the min/max temps per year. To find the *single* coolest year overall (lowest min temp) or hottest year overall (highest max temp), you would typically process this final output file separately (e.g., with another script or manually).

Practical Steps (using Python and Hadoop Streaming)

(Requires a weather data file in the format `YYYY-MM-DD min_temp max_temp` or similar, where year can be extracted).

Step 1: Prepare Environment and Scripts

1. **Log in as Hadoop User:** `su hduser`

2. **Create Mapper Script (`mapper.py`):**
    - `nano mapper.py`
    - Paste the `mapper.py` code from the PDF (extracts year, prints `year min max`).
    - Save and Exit.
3. **Create Reducer Script (`reducer.py`):**
    - `nano reducer.py`
    - Paste the `reducer.py` code from the PDF (groups by year, finds min of mins and max of maxs, prints `year\tmin_for_year\tmax_for_year`).
    - Save and Exit.
4. **Make Scripts Executable:**
    - `chmod +x mapper.py`
    - `chmod +x reducer.py`

Step 2: Start Hadoop Services

1. **Start HDFS:** `start-dfs.sh`
2. **Start YARN:** `start-yarn.sh`
3. **Check Daemons:** `jps` (Verify NameNode, DataNode, ResourceManager, NodeManager, etc.)

Step 3: Prepare Input Data in HDFS

1. **Get Weather Data:** You need a data file. Let's assume you have `weather_data.txt` locally.

   Example content:

   ```
   1950-01-01 -5 10
   1950-01-02 -8 8
   ...
   1951-01-01 -7 12
   ...
   1950-12-31 -18 5  # Example min for 1950
   1951-07-15 15 44  # Example max for 1951
   1953-07-20 18 41
   1955-01-15 -16 1
   1955-08-01 20 45  # Example max for 1955
   ...
   ```

   (Create this file using `nano weather_data.txt` if you don't have one)
2. **Remove Old HDFS Directories:**
    - `hdfs dfs -rm -r /input` (If exists)
    - `hdfs dfs -rm -r /output` (If exists)
3. **Create HDFS Input Directory:** `hdfs dfs -mkdir -p /input`
4. **Upload Weather Data to HDFS:** `hdfs dfs -put weather_data.txt /input/`

Step 4: Run the MapReduce Job

1. **Execute the Command (adjust JAR path if needed):**

```
hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-
-input /input/weather_data.txt \
-output /output/weather_output \
-mapper mapper.py \
-reducer reducer.py \
-file mapper.py \
-file reducer.py
```

(This is the same command structure as Assignment 3, just different input/output paths and potentially different data/scripts).

Step 5: Check the Output

1. **List Output Directory:** `hdfs dfs -ls /output/weather_output` (Look for `_SUCCESS` and `part-r-00000`).
2. **View the Output:** `hdfs dfs -cat /output/weather_output/part-r-00000`
   - The output should be lines like:

```
1950    -18     43  # (Actual values depend on your data)
1951    -17     44
1952    -12     32
1953    -20     41
1954    -13     40
1955    -16     45
...
```

   - Each line shows: `Year <tab> MinTempForYear <tab> MaxTempForYear`.

Step 6: Find Hottest/Coolest Year (Post-Processing)

1. **Manually Inspect Output:** Look through the output from Step 5. Find the line with the lowest minimum temperature (coolest year) and the line with the highest maximum temperature (hottest year).
   - *Example:* If the output includes `1953 -20 41` and `-20` is the lowest min temp overall, then 1953 is the coolest year. If it includes `1955 -16 45` and `45` is the highest max temp overall, then 1955 is the hottest year.
   - *(Alternatively, you could pipe the output to `sort` or write another simple script to process this `part-r-00000` file).*

Step 7: Stop Hadoop Services

1. **Stop YARN:** `stop-yarn.sh`
2. **Stop HDFS:** `stop-dfs.sh`

Potential Viva Questions & Answers

1. **Q: What is the goal of the Mapper in this assignment?**
   - **A:** To read each line of weather data, extract the year and the min/max temperatures for that day, and output the year as the key and the temperatures as the value (specifically, it prints `year minT maxT`).
2. **Q: What is the goal of the Reducer?**
   - **A:** To receive all the daily min/max temperatures for a specific year, find the absolute minimum temperature recorded in that year and the absolute maximum temperature recorded in that year, and output the year along with its overall min and max temperatures.
3. **Q: What data structure is used in the Python reducer script to group temperatures by year?**
   - **A:** It uses `collections.defaultdict(list)`. This dictionary maps each year (key) to a list of `(min_temp, max_temp)` tuples recorded for that year.
4. **Q: Explain the line `mins, maxs = zip(*temps_by_year[year])` in the reducer.**
   - **A:** `temps_by_year[year]` is a list of tuples, like `[(min1, max1), (min2, max2), ...]`. The `*` operator unpacks this list into separate arguments for `zip`. `zip` then takes corresponding elements from each tuple and groups them. So, it creates two new tuples: one with all the minimum temperatures (`mins`) and one with all the maximum temperatures (`maxs`). This efficiently separates the min and max values for finding the overall min and max for the year.
5. **Q: Does the MapReduce job directly output the single hottest and coolest year?**
   - **A:** No. The MapReduce job outputs the minimum and maximum temperature *for each year*. Finding the single overall hottest/coolest year requires processing this final output file, either manually or with another script/tool.
6. **Q: How could you modify this MapReduce job to find the average maximum temperature per year?**
   - **A:** The Mapper would output `year maxT`. The Reducer would receive `(year, [maxT1, maxT2, ...])`. It would then sum all the `maxT` values in the list and divide by the number of values (the count) to calculate the average, outputting `year averageMaxT`.
7. **Q: What happens if the input `weather_data.txt` contains lines with incorrect formatting or non-numeric temperatures?**
   - **A:** The Mapper might fail to split the line correctly or extract the year. The Reducer uses a `try-except` block around the `int()` conversion, so it would likely skip lines where temperatures are not valid integers, preventing the whole job from crashing but potentially leading to inaccurate results if much data is bad. Robust error handling might involve logging skipped lines or using a placeholder value.

8. **Q: Why use MapReduce for this problem instead of just writing a single Python script to process the file locally?**
    - **A:** If the weather data file is *extremely* large (terabytes or petabytes), a single script on one machine would be too slow or might run out of memory. MapReduce, running on a Hadoop cluster, can process the data in parallel across many machines, making it much faster and scalable for big data scenarios. For a small file, a single script is simpler and faster.
9. **Q: What information does the Mapper need from the Reducer, or vice versa?**
    - **A:** They don't directly communicate. The Mapper processes its input chunk and produces key-value pairs. The framework handles collecting, sorting, and grouping these. The Reducer receives the grouped data for one key at a time and processes it independently of other reducers or the mappers.
10. **Q: How does Hadoop ensure fault tolerance (e.g., if one computer in the cluster fails during the job)?**
    - **A:** HDFS stores data redundantly across multiple machines. If a machine running a task fails, YARN can reschedule that task (e.g., re-run the Map task for that input split) on another available machine in the cluster.