LAB 1 — Remote Procedure Call (RPC) Implementation & Deployment on AWS EC2

Distributed Computing — Trimester 8

Estimated time: 2–3 hours
Submission: Video demo + code repository

---

1. Lab Objectives

By completing this lab, students will be able to:

- Implement a simple **RPC protocol** in a programming language of their choice (Python, Go, Java, Node.js).
- Understand RPC components: **client stub, server stub, marshalling, network transport**.
- Deploy a real **client–server distributed application on AWS EC2**.
- Observe **communication failures**, retry logic, and behavior under delays.
- Evaluate at-least-once vs at-most-once semantics.

This lab reinforces CLOs:

- Implementing distributed algorithms (RPC).
- Deploying distributed systems on EC2.
- Analyzing communication reliability and failure behavior.

---

2. What You Will Build

You will implement a **minimal RPC system**, consisting of:

Server Node

- Exposes one remote function, e.g.:
  - add(a, b)
  - get_time()
  - reverse_string(s)
  - or any simple deterministic function
- Listens on a TCP or UDP socket
- Decodes requests, executes the function, returns the result

Client Node

- Sends remote requests to the server using your RPC format
- Handles:
    - timeouts
    - retries
    - unique request IDs
- Shows result on the terminal

Network layer

- Your own simple request/response structure, e.g. JSON, protobuf, or custom text format

Deployment

- 2 EC2 instances (t2.micro or t3.micro)
- Client on Instance A, Server on Instance B
- SSH into both, run programs, test RPC calls

---

3. Pre-Lab Setup (Required Before the Lab Session)

Step 1 — Create two EC2 Instances

- Launch **Ubuntu 22.04** (recommended)
- Name them:
    - rpc-client-node
    - rpc-server-node
- Open **port 5000** (or any port you choose) in **Security Group inbound rules**

Step 2 — Install Dependencies

On both EC2 machines:

```
sudo apt update

sudo apt install python3 python3-pip -y
```

(Or install Go/Java/Node depending on your language.)

Step 3 — Test Connectivity

From client → server:

```
ping <server-public-ip>

nc -vz <server-public-ip> 5000
```

If this works, you’re ready.

---

4. Implementation Requirements

✓ Required RPC Message Structure

You must include:

- **Request ID** (UUID or incrementing)
- **Function name** (string)
- **Arguments** (list or dict)
- **Timestamp** (optional)

Example (JSON):

```
{
  "request_id": "123-abc",
  "method": "add",
  "params": {"a": 5, "b": 7}
}
```

✓ Server Responsibilities

- Listen for requests
- Parse incoming JSON
- Execute correct function
- Return response:

```
{
  "request_id": "123-abc",
  "result": 12,
  "status": "OK"
}
```

✓ Client Responsibilities

- Send request
- Start timeout timer (e.g. 2 seconds)
- If there is no response:

    - retry the request (2–3 retries max)
- Print final result or error

✓ Error Handling

You must demonstrate **one** of the following:

- Server intentionally slowed (e.g., sleep 5 seconds)
- Client retries logic triggers
- Lost packet simulation (drop response)
- Server crash scenario

---

5. Tasks

Task 1 — Implement the RPC client & server

Language options: Python, C++, Go, Java, Node.js

Requirements:

- TCP or UDP sockets (you choose)
- JSON or custom serialization
- One or more remote functions
- Logging for each request

Task 2 — Deploy and run on EC2 (use two-node environment that you created in lab0)

On EC2 server:

```
python3 server.py
```

On EC2 client:

```
python3 client.py
```

Confirm that the remote method call works.

Task 3 — Demonstrate failure handling (REQUIRED)

Perform **one** of:

1. Kill server mid-request → client retries
2. Add artificial delay → observe timeout

3. Drop client request or response (change code)

4. Block network with firewall:

```
sudo ufw deny 5000
```

Observe and explain:

- What happened?
- Which RPC semantics were achieved?

(**Explain verbally or in writing**.)

Task 4 — Submit Deliverables

🟦 **Deliverable 1: Video Demo (1–2 minutes)**
Video must show:

- Running server on EC2
- Running client on different EC2 instance
- Successful RPC call
- A failure scenario + explanation **in voice**

🟦 **Deliverable 2: GitHub Repo Link**
Repo must include:

- client.py / server.py
- README explaining how to run it
- Requirements file (if needed)

🟦 **OR written report with screenshots and comments.**

---

6. Evaluation Criteria (100 points total)

Component	Points	Description
Functional RPC implementation	30	Request/response works correctly
Deployment on EC2	20	Two nodes communicating over the network
Failure demonstration	30	Correct timeout/retry or crash behavior

Component	Points	Description
Video or report clarity	10	Clear voice explanation / well-structured report with comments and screenshots
Code quality	10	Clean, readable, documented

---

7. Optional Enhancements

- Add **idempotency** handling
- Support **multiple RPC methods**
- Add **client-side caching**
- Implement **at-most-once semantics** using a request log