We all know that large-scale platforms like Instagram or Facebook serve millions, if not billions, of users every day.

This raises an important question:
How do these companies manage such an enormous volume of traffic efficiently?

Clearly, relying on a single server would be insufficient to handle this kind of demand.
To ensure scalability and reliability, multiple servers must work in coordination to manage user requests.

Purpose of the Project

The aim of my project is to demonstrate how a distributed system architecture, using multiple services or servers, can effectively handle user requests. Let's dive into how this architecture works in practice.

System Overview

At the core of the system is a **master server**, responsible for delegating computational tasks to **slave servers**. Here's a step-by-step breakdown:

1. **Task Assignment Initiation:**
   When a client sends a request (e.g., a mathematical expression to evaluate), the master server must determine which slave is available to handle the task.

2. **Slave Availability Check:**
   The master contacts the first slave with a binary question:
   "*Are you available to perform this task?*"
   The slave responds with either:

   - "*I am available.*"
   - "*I am not available.*"

3. **Delegation Based on Availability:**

   - If the first slave is available, the master assigns the task.
   - Suppose the calculation takes 7 seconds to complete. While this is in progress, another request may arrive.
   - The master again queries the first slave. If it’s still busy, the master checks with the second slave, and so on.

4. **Handling Multiple Requests:**

   - Each new request is routed to the next available slave.
   - If all slaves are busy, the master replies to the client indicating the system is currently at full capacity.

5. **Result Communication:**

   - Once a slave completes its task, it returns the result to the master.
   - The master logs the completed task and stores it in a task list.

- Clients can then query the master to retrieve the results of their submitted tasks.

Technical Implementation

To simulate this system, I used **Hyper-V** to set up three virtual machines: one master and two slaves. Once the environment is configured:

- The master service registers available slaves.
- A new task can be submitted via a POST request.
- The system dynamically assigns tasks based on current availability.
- You can verify the state of each task (e.g., pending, in progress, or completed) by querying the task status.

For example:

- After submitting a task, the master may respond:
  "*Task successfully assigned to Slave 2.*"
- A subsequent request to `/get-tasks` will reflect updated statuses such as “in progress” or “done.”

Conclusion

This project showcases the core principles of distributed computing—task distribution, load balancing, and asynchronous processing. By simulating a real-world architecture on a small scale, it helps us understand how large tech companies manage millions of simultaneous operations efficiently.

Thank you for your time, and I hope you found this project insightful.