A small bank is using a client/server system with a single database server for storing the transaction histories and account balances of all their customers. Users query the server by sending the account number and PIN of a customer, plus the type and parameters of the required transaction (e.g. deposit, withdrawal, transfer). The server performs the transaction and returns the new account balance of the customer. Recently, the bank has been experiencing rapid growth. As a result, the bank server has become overloaded. The bank has decided to reorganize its system to improve its performance by adding more servers. To make the transition to the new system as smooth as possible, the bank does not wish to make any changes to the client portion of its software (the part which resides on every ATM and every bank teller's workstation and implements the user interface). However, the current client software has been designed to communicate with a single server only. The network address of the database server is hard coded in the client software.

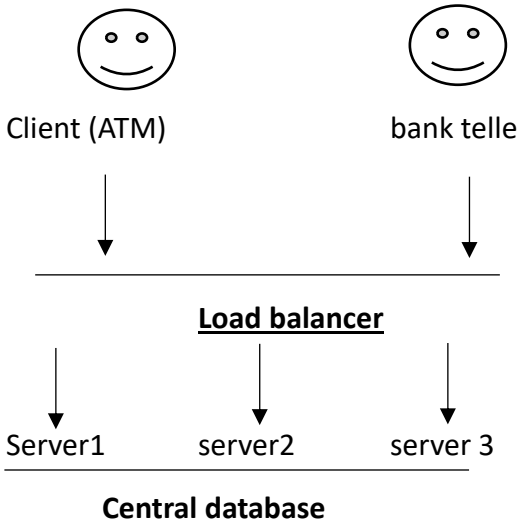
Question 1

(a) Suitable Computing Environment

The most suitable computing environment for the given scenario is **a distributed computing environment**, taking into consideration the need for:

1. **Scalability:** Distributed systems allow the bank to add multiple servers to share the load. This will prevent overloading a single server and ensure better performance.
2. **Fault Tolerance:** If one server fails, other servers can continue to handle requests, improving system reliability.
3. **Load Balancing:** Traffic can be evenly distributed among servers using load balancers, ensuring optimal response times.
4. **Transparency:** Changes in the backend (adding servers) will not require modifications to the client software since the client can still interact with the same interface (potentially through a load balancer or middleware).

b) Digramatical representation of the environment



roles	
client	Send requests with account number, PIN, and transaction details
Load balancer	Acts as an intermediary, directing client requests to the least busy or most suitable server
servers	Perform requested transactions (e.g., deposit, withdrawal, transfer) and communicate results back to clients.
Central database	Stores and updates all customer account data, ensuring consistency across servers.

C) System Programming Concept that can be applicable to enable the above process to take place involves :-

Remote Procedure Call (RPC) or **socket-based client-server communication**.

RPC allows the client to execute procedures on the server seamlessly. The client only needs to call a function with parameters (e.g., account number, PIN, transaction type), and the server handles the actual processing.

Socket programming establishes a reliable communication channel between the client and server over a network using TCP/IP, enabling the secure transfer of sensitive data like account credentials and balances.

Both methods ensure:

- Efficient communication between distributed components.
- Minimal changes to the client-side code since the interaction patterns remain consistent.

Three system calls relevant to this scenario include:

1. **socket:** Creates a communication endpoint.
    - **Unix Implementation:** int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    - **Windows Implementation:** SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
2. **connect:** Establishes a connection to the server.
    - **Unix Implementation:** connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    - **Windows Implementation:** connect(sock, (SOCKADDR*)&serverAddr, sizeof(serverAddr));
3. **send and recv:** Send and receive data through the socket.