1. Model the load balancer's server selection process. Sketch the data structure for the problem.

The load balancer (LB) is responsible for distributing incoming transaction requests to the least busy server. This process can be modeled using the following data structure:

Data Structure:

Servers:

Each server has:

- ☐ A queue of transactions.
- ☐ An active_count variable to track the number of active requests assigned to it.

Load Balancer:

- ☐ A priority queue (min-heap) to efficiently keep track of servers based on their active_count.
- ☐ Each entry in the priority queue is a pair (server_id, active_count).

Selection Process:

- ☐ The load balancer checks the priority queue for the server with the smallest active_count.
- ☐ If multiple servers have the same active_count, a random server among the tied servers is selected.
- ☐ The transaction is assigned to the selected server, and its active_count is updated.

Load Balancer:

Priority Queue: [(Server_1, active_count_1), (Server_2, active_count_2), ..., (Server_S, active_count_S)]
Servers:

Server_1: Queue: [Transaction_1, Transaction_2, ...], active_count: int
Server_2: Queue: [Transaction_1, Transaction_2, ...], active_count: int
...
Server_S: Queue: [Transaction_1, Transaction_2, ...], active_count: int

2. Is there a sorting-related problem here?

Yes, there are multiple sorting-related problems:

Server Selection:

- ☐ The load balancer must always find the server with the smallest `active_count`. Maintaining this ordering requires a sorting mechanism, such as a priority queue.
- ☐ **Transaction Processing:**
    a. Within each server, transactions are processed in the order they arrive (FIFO). However, sorting may become necessary if priorities or waiting times are introduced.
- ☐ **Reordering Internal Records:**
    a. After processing a transaction, the server requires time to reorder its internal records. This can involve sorting based on criteria such as transaction size, timestamp, or priority.

3. Which type of algorithm would be better suited for this problem? Why?

A **priority queue-based algorithm** is the best choice for this problem because:

- **Server Selection:**
  - A priority queue (min-heap) efficiently maintains the order of servers based on their `active_count`. The least busy server can be selected in $O(1)$O(1), and updates take $O(\log S)$O(logS), where SS is the number of servers.
- **Scalability:**
  - Priority queues handle dynamic updates efficiently, making them suitable for real-time systems with frequent changes.
- **Fairness:**
  - The algorithm ensures that servers with the least load are prioritized, leading to balanced load distribution.

4. Handling High-Priority Customers

To implement a "high-priority customer" program, the system needs to process high-priority transactions before ordinary transactions.

Changes in Data Structure:

1. Add a **priority field** to each transaction:
    a. Example: `Transaction(priority, timestamp, size)`.
2. Replace each server's queue with a **priority queue**, where:
    a. Transactions with higher priority are processed first.
    b. If priorities are equal, transactions are processed based on their arrival time (FIFO).

Algorithm Changes:

1. The load balancer assigns transactions to servers as usual.
2. Within each server:
    a. Use a priority queue to process high-priority transactions before ordinary transactions.
    b. When a transaction is added to the server's queue, it is automatically ordered by priority.

5. Handling Transactions Waiting the Longest Duration

If regulations require prioritizing transactions based on their waiting time:

Changes in Data Structure:

1. Add a **timestamp field** to each transaction:
    a. Example: `Transaction(priority, timestamp, size)`.
2. Use a **priority queue** within each server, where:
    a. Priority is determined by the waiting time, calculated as `current_time - timestamp`.

Algorithm Changes:

1. The load balancer assigns transactions to servers as usual.
2. Within each server:
    a. Dynamically re-prioritize transactions based on their waiting time.
    b. The priority queue ensures that the transaction with the longest waiting time is processed first.

6. Sorting Stability

Current Design:

- **Server Selection:**

    - The current design uses `active_count` to sort servers. This is stable because tied servers are resolved randomly.
- **Transaction Processing:**
    - Within each server, transactions are processed in FIFO order, ensuring stability.

Proposed Designs:

1. **High-Priority Customers:**
    a. Sorting is stable within the same priority level (FIFO is maintained for transactions with the same priority).
2. **Longest Waiting Transactions:**
    a. Sorting stability may be affected because transactions are dynamically re-prioritized based on waiting time.

Summary of the Proposed Solution

1. **Data Structure for Load Balancer:**
    a. A priority queue to maintain servers by `active_count`.
2. **Data Structure for Servers:**
    a. A priority queue for managing transactions based on priority or waiting time.
3. **Algorithms:**
    a. Use a min-heap for server selection.
    b. Use a dynamic priority queue for transaction processing.
4. **Sorting Stability:**
    a. Maintain FIFO stability for equal-priority transactions.
    b. Re-prioritization based on waiting time may affect stability.

This approach ensures an efficient and fair system while adapting to new requirements like high-priority customers and waiting-time prioritization.