Load Balancer Design

1. Server Architecture Design

1.1 Master Server * 1

Threads / Jobs

- Frontend server
- Auto scaler

RPC calls

- setServerIsWaiting() → help tracking idle servers.
- notifyRunning() → help tracking servers that are boosted but not running.

Variables

- A queue storing request parsed by frontend, waiting for backend to process
- A queue of waiting/idle servers
- Numbers of frontend servers, backend servers, waiting servers…

1.2 FrontTier Server * N

Threads / Jobs

- Handle AcceptConnection and ParseRequest → Push request to Master server

RPC calls

- shutdown()

1.3 MidTier Server * N:

Threads / Jobs

- Poll request from Master server → Handle ProcessRequest

RPC calls

- shutdown()

2. Scaling Algorithm Design

2.1 General Idea

a. If there's a lot of waiting clients (compared to num of servers) → scale up
b. If there's a lot of idle servers (compared to num of servers) → scale down
c. If there's bottleneck – like a lot of requests in queue → don't scale up, as it won't help
d. If there's many servers boosted but not running → don't scale up, as those will reduce queue length after it starts running
e. If the first MidTier server is not running yet → drop the request, as it'll probably time out

2.2 Front Tier

Scale Up

- frontQueueLen > numFront * *threshold* + boostingFront * *threshold*: too many waiting connections in queue

- midQueueLen < numMid * *threshold* && numFront < numMid + 1: there's no bottle neck

Scale Down

- notScaledFrontUp
- waitingFront.size() > Math.max(2.0, numFront * *threshold*)

2.3 Mid Tier

Scale Up

- midQueueLen > numMid * threshold + boostingMid * *threshold*: too many waiting connections in queue
- numMid < 10: there's no bottle neck

Scale Down

- notScaledMidUp
- waitingMid.size() > Math.max(2.0, numMid * *threshold*)