Cloud Computing (CC) Presentation

Load Balancing & Its Strategies

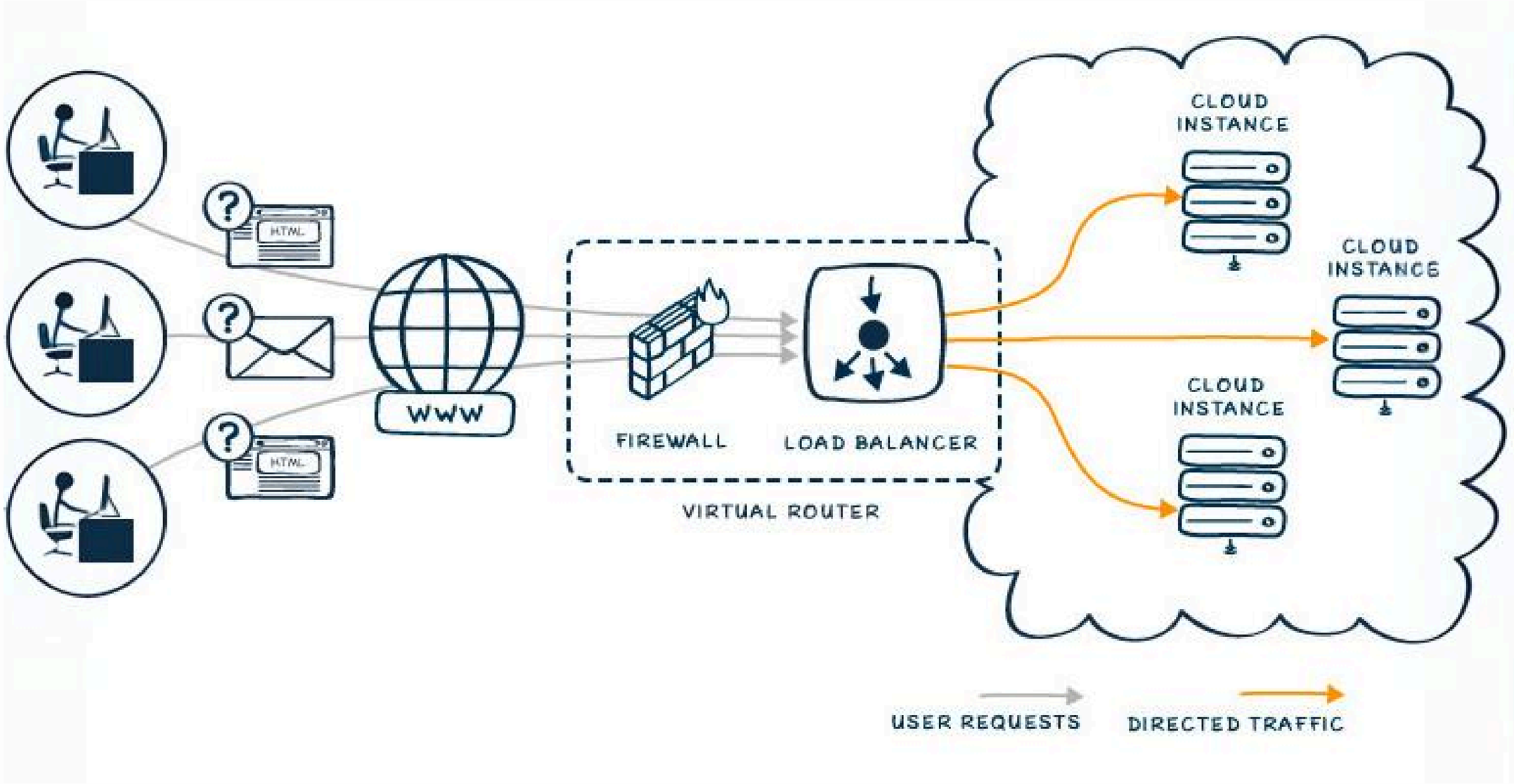
Introduction

What is a Load Balancer?

- A networking device or software application
- Distributes and balances incoming traffic among the horizontally scaled homogeneous servers

Advantages

- Higher availability
- Higher scalability
- More efficient utilization of servers
- Better application security
- Better application performance


HTML
HTML
WWW
FIREWALL
LOAD BALANCER
VIRTUAL ROUTER
CLOUD INSTANCE
CLOUD INSTANCE
CLOUD INSTANCE
USER REQUESTS
DIRECTED TRAFFIC

Availability

- Server failure can increase application downtime
- Load balancers increase the **fault tolerance** of systems
- Automatically detects server problems and redirects client traffic to available servers

- Make these tasks easier:
  - Run application server maintenance or upgrades **without application downtime**
  - Provide automatic **disaster recovery** to backup sites
  - Perform **health checks** and prevent issues that can cause downtime

Scalability

- **Direct network traffic** intelligently among multiple servers
- Handle millions of client requests
- Easier horizontal scaling

- Functionalities:
  - Prevents **traffic bottlenecks** at any one server
  - Predicts application traffic so that you can **add or remove** different **servers**, if needed
  - Adds **redundancy** to your system so that you can **scale with confidence**

Security

- **Built-in security features** to add another layer of security
- Useful tool to deal with **DDoS attacks**
- Concurrent requests which cause server failure are redirected.

- Handles the following:
  - **Monitor traffic** and **block malicious** content
  - Automatically **redirect attack traffic** to multiple backend servers to minimize impact
  - Route traffic through a **group of network firewalls** for additional security

Performance

- Improve application performance by:
  - Increasing response time
  - Reducing network latency.

- Critical tasks such as:
  - **Distribute the load evenly** between servers to improve application performance
  - Redirect client requests to a geographically closer server to **reduce latency**
  - Ensure the **reliability** and **performance** of physical and virtual computing resources

Load Balancing Strategies

What are LB Strategies?

- A load balancing strategy/algorithm is the set of rules that a load balancer follows
- Goal is to determine the best server for each of the different client requests
- For a request “X” which server “Y” is the most optimal to be routed to

Types of LB Strategies

- Load balancing algorithms fall into two main categories:
  - Static LB Strategies - Static load balancing strategies/algorithms follow fixed rules and are independent of the current server state
  - Dynamic LB Strategies - Dynamic load balancing strategies/algorithms examine the current state of the servers before distributing traffic, whixh includes no. of connections, response time of the server, etc.

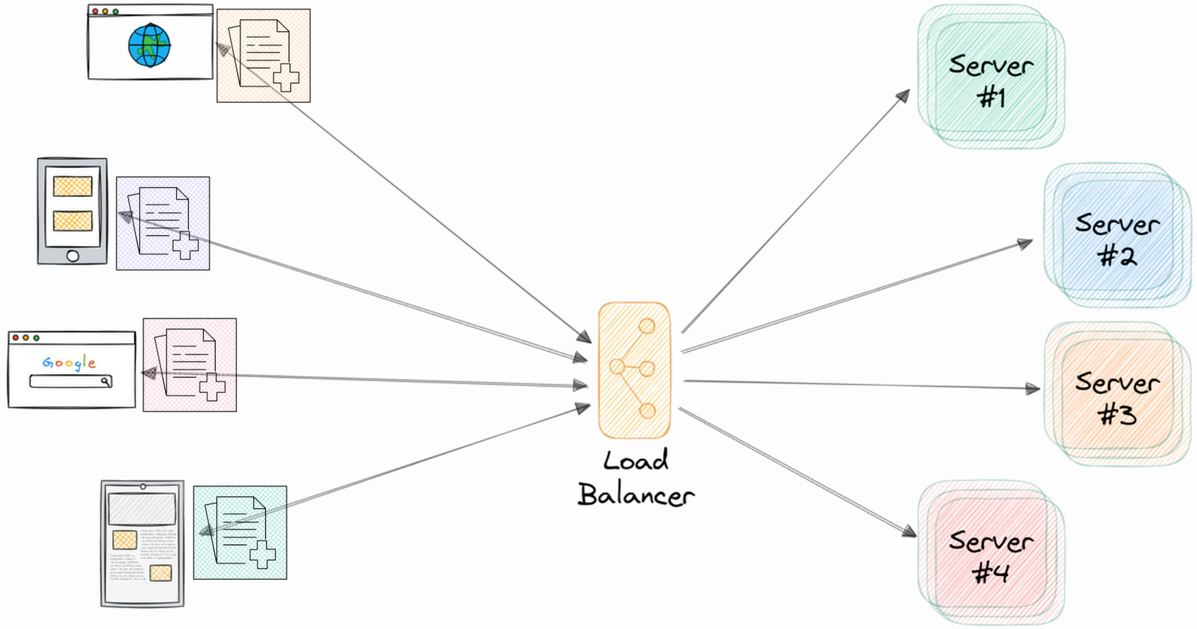
Implementation Details

- Protocol (HTTP, WebSocket, etc) agnostic load balancing
- Forwarding TCP segments instead of HTTP, WS, etc requests and responses
- Client or server is oblivious to whether the requests and responses are going through a load-balancer or to the server or client respectively directly
- Load balancer handles two TCP connections for each client-to-server connection, one between client and LB and the other between LB and one of the servers
- IO between both the connections can be handled in user-space (infinite loop reading and writing) or kernel-space (sendfile(2) and copy_file_range(2) syscalls in Linux)

(S) Random

- Distributes incoming requests randomly across servers
- Simple and easy to implement

- Pros:
  - No need for tracking server load
  - Even distribution over time if request volume is high
- Cons:
  - Can lead to uneven distribution for smaller volumes
  - Doesn't account for server capacity or load

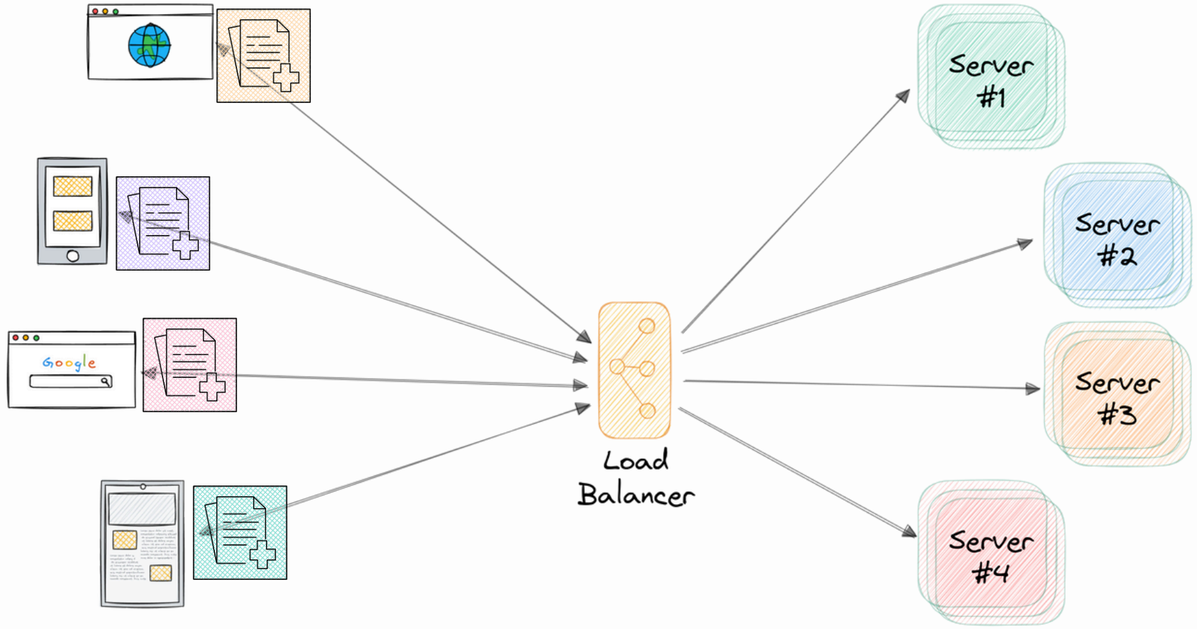


```go
type RandomStrategy struct {
    servers []ServerAddr
}

func NewRandomStrategy(servers []ServerAddr) *RandomStrategy {
    return &RandomStrategy{servers}
}

func (strategy *RandomStrategy) ServerAddr(ClientAddr string) (ServerAddr, error) {
    if len(strategy.servers) == 0 {
        return "", errors.New("no servers available")
    }

    server := strategy.servers[rand.IntN(len(strategy.servers))]

    return server, nil
}

func (strategy *RandomStrategy) Connected(addr ServerAddr)    {}
func (strategy *RandomStrategy) Disconnected(addr ServerAddr) {}
```

(S) Round Robin

- Sequentially distributes requests across servers in a circular order
- Equal distribution of traffic

- Pros:
  - Simple to implement
  - Ensures that no server is overloaded early on
- Cons:
  - Doesn't account for server load or capacity differences
  - Can cause delays if a server becomes overloaded

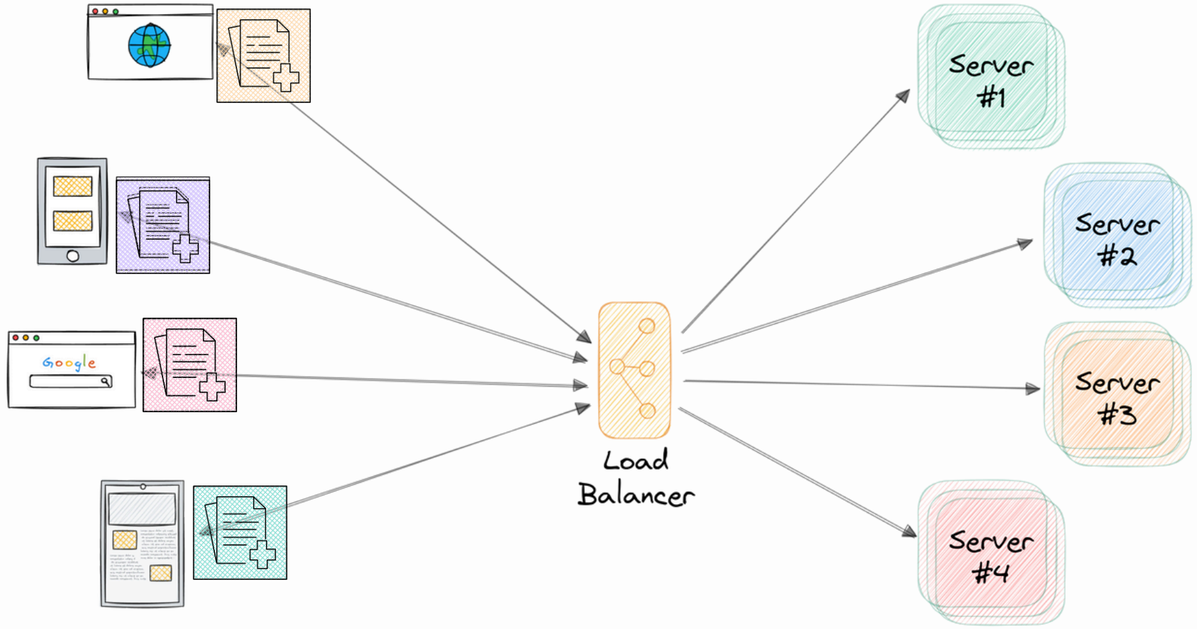


```go
type RoundRobinStrategy struct {
    servers   []ServerAddr
    nextIndex int
}

func NewRoundRobinStrategy(servers []ServerAddr) *RoundRobinStrategy {
    return &RoundRobinStrategy{servers, 0}
}

func (strategy *RoundRobinStrategy) ServerAddr(ClientAddr string) (ServerAddr, error) {
    if len(strategy.servers) == 0 {
        return "", errors.New("no servers available")
    }

    server := strategy.servers[strategy.nextIndex]
    strategy.nextIndex = (strategy.nextIndex + 1) % len(strategy.servers)

    return server, nil
}

func (strategy *RoundRobinStrategy) Connected(addr ServerAddr)    {}
func (strategy *RoundRobinStrategy) Disconnected(addr ServerAddr) {}

```

(S) Weighted Round Robin

- Distributes requests based on preassigned weights to each server
- Servers with higher capacity receive more requests

- Pros:
  - Accounts for server performance differences
  - Customizable for different server configurations
  - Leads to better resource utilization in a heterogeneous server environment
- Cons:
  - Weights need to be manually calculated and assigned
  - Not ideal for dynamic load changes

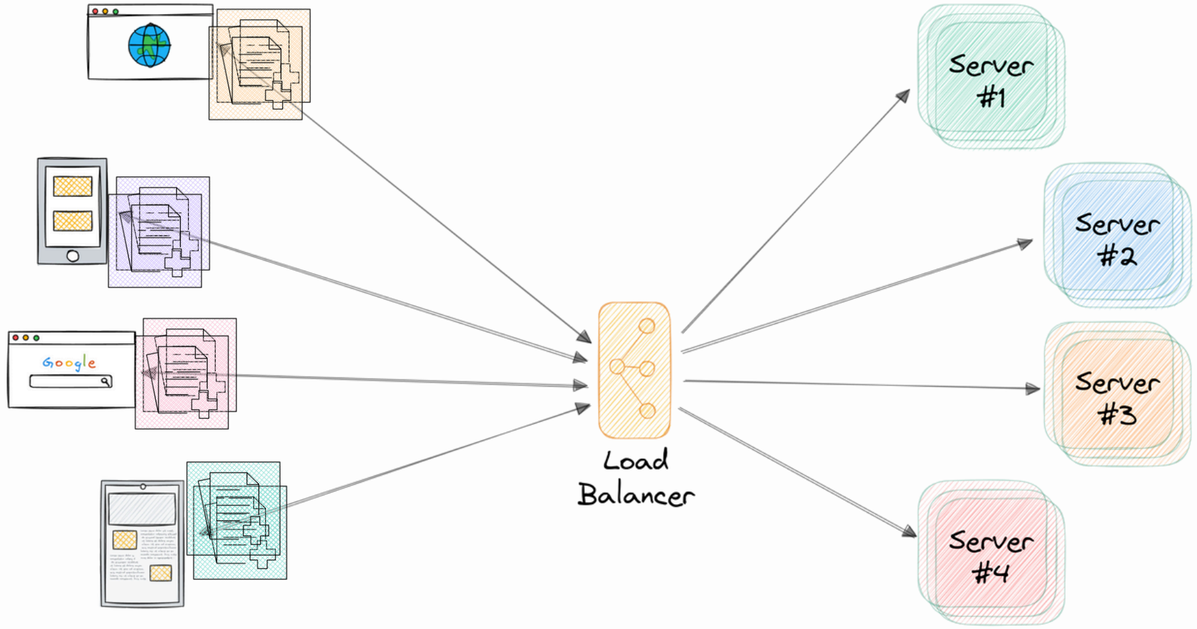


```go
type WeightedRoundRobinStrategy struct {
    servers       []ServerAddr
    weights       []float32
    nextIndex     int
    selectedTimes int
}

func NewWeightedRoundRobinStrategy(servers []ServerAddr, weights []float32) *WeightedRoundRobinStrategy {
    // Normalize the weights to 1
    sum := float32(0)
    for _, weight := range weights {
        sum += weight
    }

    for i, weight := range weights {
        weights[i] = weight / sum
    }

    return &WeightedRoundRobinStrategy{servers, weights, 0, 0}
}

func (strategy *WeightedRoundRobinStrategy) ServerAddr(ClientAddr string) (ServerAddr, error) {
    if len(strategy.servers) == 0 {
        return "", errors.New("no servers available")
    }

    server := strategy.servers[strategy.nextIndex]
    strategy.selectedTimes++

    if strategy.selectedTimes >= int(strategy.weights[strategy.nextIndex]*WeightFactor) {
        strategy.nextIndex = (strategy.nextIndex + 1) % len(strategy.servers)
        strategy.selectedTimes = 0
    }

    return server, nil
}

func (strategy *WeightedRoundRobinStrategy) Connected(addr ServerAddr)    {}
func (strategy *WeightedRoundRobinStrategy) Disconnected(addr ServerAddr) {}
```

(S) IP Hash

- Uses client IP addresses to consistently direct requests to the same server
- Ensures session persistence or sticky sessions

- Pros:
  - Useful for session-based applications
  - Simple hash-based mapping
- Cons:
  - Load distribution might become uneven if client IPs are clustered and the hash function is weak
  - Not suitable for environments with changing client IPs (e.g., mobile clients) as the session persistence has no benefits in this case

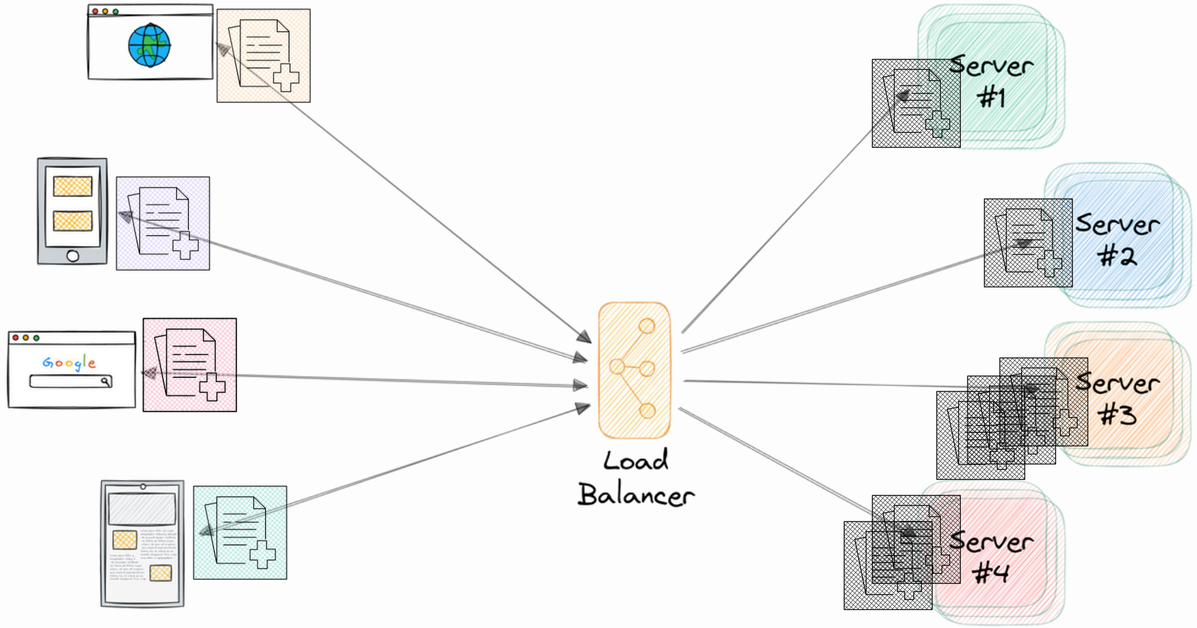


```go
type IPHashStrategy struct {
    servers []ServerAddr
}

func NewIPHashStrategy(servers []ServerAddr) *IPHashStrategy {
    return &IPHashStrategy{servers}
}

func (strategy *IPHashStrategy) ServerAddr(ClientAddr string) (ServerAddr, error) {
    if len(strategy.servers) == 0 {
        return "", errors.New("no servers available")
    }

    fragments := strings.Split(ClientAddr, ":")
    addr := strings.Join(fragments[:len(fragments)-1], ":")

    // Hash the IP address (Based on the Sum of Bytes of the Client Address) to select a server.
    // TODO: Implement a better hashing algorithm.
    hash := 0
    for i := 0; i < len(addr); i++ {
        hash += int(addr[i])
    }

    server := strategy.servers[hash%len(strategy.servers)]

    return server, nil
}

func (strategy *IPHashStrategy) Connected(addr ServerAddr)    {}
func (strategy *IPHashStrategy) Disconnected(addr ServerAddr) {}
```

(D) Least Connection

- Directs requests to the server with the fewest active connections
- Active connections provides a good heuristic for server load. Getting server load information like CPU, memory, etc usage repeatedly can induce latency in the system which LBs should aim to minimize.

- Pros:
  - Ensures that servers with less load handle more requests
  - Better performance during traffic spikes
- Cons:
  - More computationally intensive as connections need to be constantly tracked
  - Not always effective with varying request sizes. Active connections is only a heuristic.

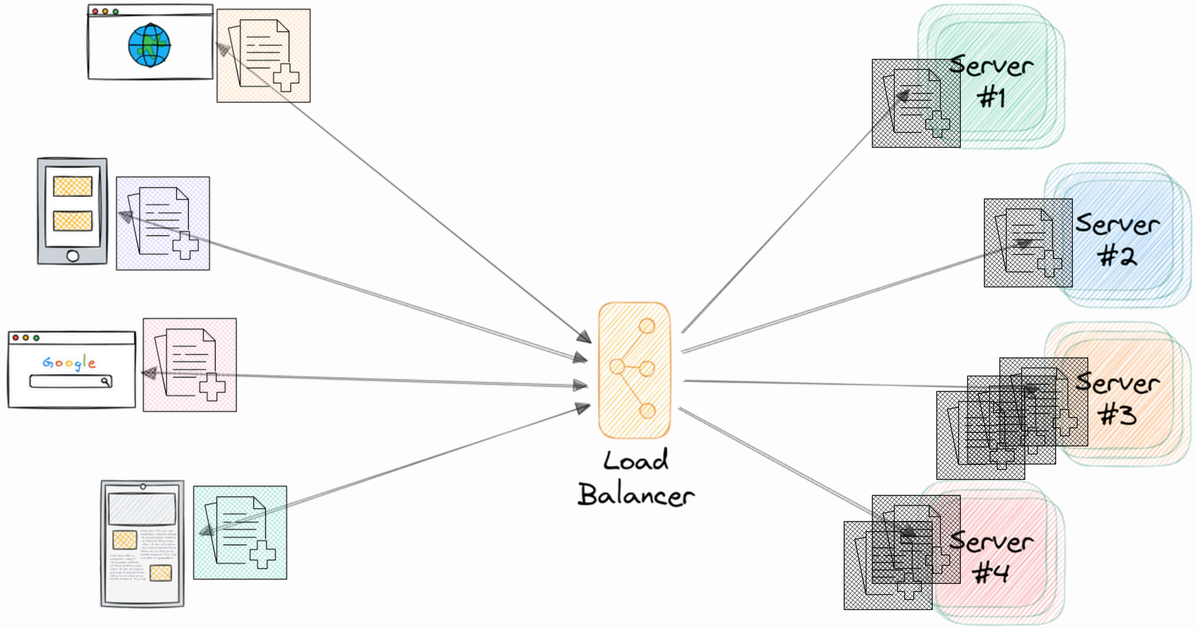


```go
type LeastConnectionsStrategy struct {
    servers     []ServerAddr
    connections []int
}

func NewLeastConnectionsStrategy(servers []ServerAddr) *LeastConnectionsStrategy {
    connections := make([]int, len(servers))
    for i := range connections {
        connections[i] = 0
    }

    return &LeastConnectionsStrategy{servers, connections}
}

func (strategy *LeastConnectionsStrategy) ServerAddr(ClientAddr string) (ServerAddr, error) {
    if len(strategy.servers) == 0 {
        return "", errors.New("no servers available")
    }

    minConnections := strategy.connections[0]
    minIndex := 0

    for i, connections := range strategy.connections {
        if connections < minConnections {
            minConnections = connections
            minIndex = i
        }
    }

    server := strategy.servers[minIndex]
    strategy.connections[minIndex]++

    return server, nil
}

func (strategy *LeastConnectionsStrategy) Connected(addr ServerAddr) {}

func (strategy *LeastConnectionsStrategy) Disconnected(addr ServerAddr) {
    log.Println(strategy.connections)
    for i, serverAddr := range strategy.servers {
        if serverAddr == addr && strategy.connections[i] > 0 {
            strategy.connections[i]--
            break
        }
    }
    log.Println(strategy.connections)
}
```

(D) Weighted Least Connection

- Similar to Least Connections but considers server weights, giving more connections to higher-capacity servers
- Combines load-based balancing with server capacity
- Performs better in both a heterogeneous server environment and one with varying response times

- Pros:
  - More precise load distribution for different server capabilities
  - Suitable for mixed environments (servers with different specs)
- Cons:
  - Higher complexity in terms of configuration and tracking



```go
type WeightedLeastConnectionsStrategy struct {
    servers     []ServerAddr
    connections []int
    weights     []float32
}

func WeightedNewLeastConnectionsStrategy(servers []ServerAddr, weights []float32) *WeightedLeastConnectionsStrategy {
    connections := make([]int, len(servers))
    for i := range connections {
        connections[i] = 0
    }

    // Normalize the weights to 1
    sum := float32(0)
    for _, weight := range weights {
        sum += weight
    }

    for i, weight := range weights {
        weights[i] = weight / sum
    }

    return &WeightedLeastConnectionsStrategy{servers, connections, weights}
}

func (strategy *WeightedLeastConnectionsStrategy) ServerAddr(ClientAddr string) (ServerAddr, error) {
    if len(strategy.servers) == 0 {
        return "", errors.New("no servers available")
    }

    maxRatio := float32(0.0)
    maxIndex := 0

    totalConnections := 0
    for _, connections := range strategy.connections {
        totalConnections += connections
    }

    // Below code works for Least Connection Strategy
    for i, connections := range strategy.connections {
        normalizedConnections := float32(connections) / (float32(totalConnections) + 1e-6)

        ratio := float32(strategy.weights[i]) / (float32(normalizedConnections) + 1e-6)

        if ratio > maxRatio {
            maxRatio = ratio
            maxIndex = i
        }
    }

    server := strategy.servers[maxIndex]
    strategy.connections[maxIndex]++

    return server, nil
}

func (strategy *WeightedLeastConnectionsStrategy) Connected(addr ServerAddr) {}

func (strategy *WeightedLeastConnectionsStrategy) Disconnected(addr ServerAddr) {
    log.Println(strategy.connections)
    for i, serverAddr := range strategy.servers {
        if serverAddr == addr && strategy.connections[i] > 0 {
            strategy.connections[i]--
            break
        }
    }
    log.Println(strategy.connections)
}

```

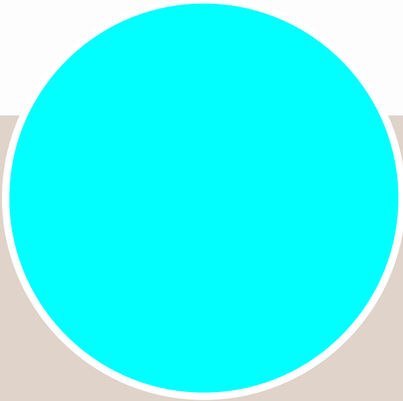
Pitfalls

1. Connections handles by Load Balancer
    a. LB handles 2x TCP connections
    b. So the connection limit on the LB server can be a bottleneck
2. LB servers need to be scaled horizontally too
3. An orchestrator service is needed for monitoring server health in a more granular way
4. Dynamic handling of servers (on-the-go addition & deletion of servers)

Team Members

Vinayak Anand

S20220010243



Saurabh Pal

S20220010196

Bishwajeet Sahoo

S20220010039

Thank You...