SIMPLIFYING BY IMPLEMENTING DISTRIBUTED SYSTEMS

Distributed Systems Simplified

Read Replication:

Scaling reads without breaking a sweat.

AUTHOR:

Niv Rave

Github: https://github.com/NivRave/Distributed-Systems-Simplified

The Reality of Modern Apps

The Read-Heavy Reality

The Reality of Modern Apps: Most applications are **Read-Heavy**.

Think about LinkedIn, Twitter, or Amazon:

- For every 1 post you write, 1,000 people read it.
- For every 1 profile update, 10,000 views occur.
- The Ratio: 99% Reads vs. 1% Writes.

The Problem: If your single Primary database handles 100% of the traffic, the 99% (Reads) are constantly competing with the 1% (Writes) for disk I/O and memory.

The Scaling Wall:

- Vertical Scaling: Buying a $20,000 server to handle more reads is inefficient and has a hard ceiling.
- Contention: Heavy read queries can "lock" tables, slowing down critical user writes.

The Fix: **Read Replication.**

The Architectural Solution

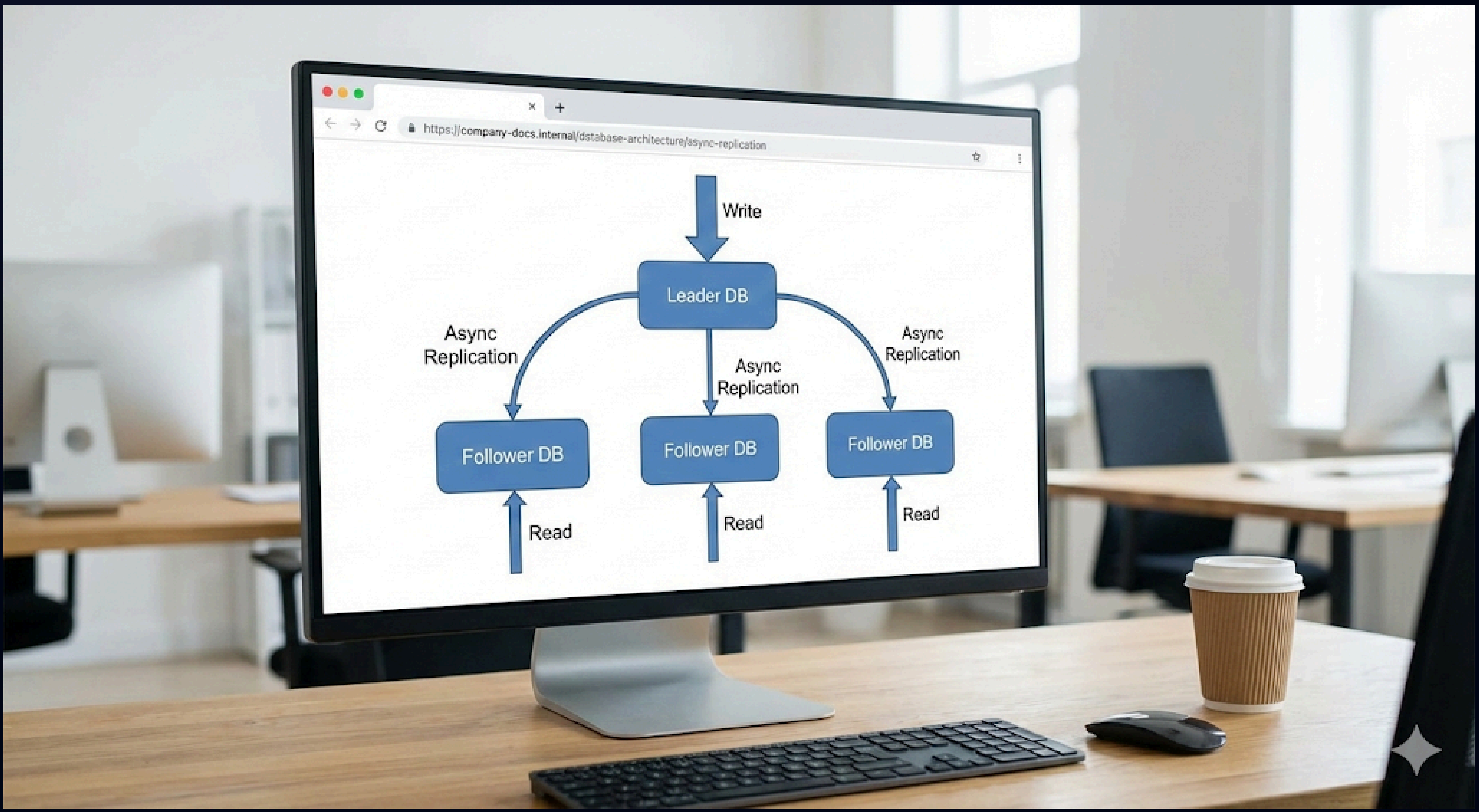
Separating Concerns: Leader vs. Follower

Instead of one massive server, we build a Cluster:

The Leader (Primary): The single source of truth. It is the only node that accepts Write operations.

The Followers (Replicas): Read-only copies. They subscribe to an update stream from the Leader.

The Result: Your write capacity is preserved on the Leader, while your read capacity scales horizontally by adding more Followers.



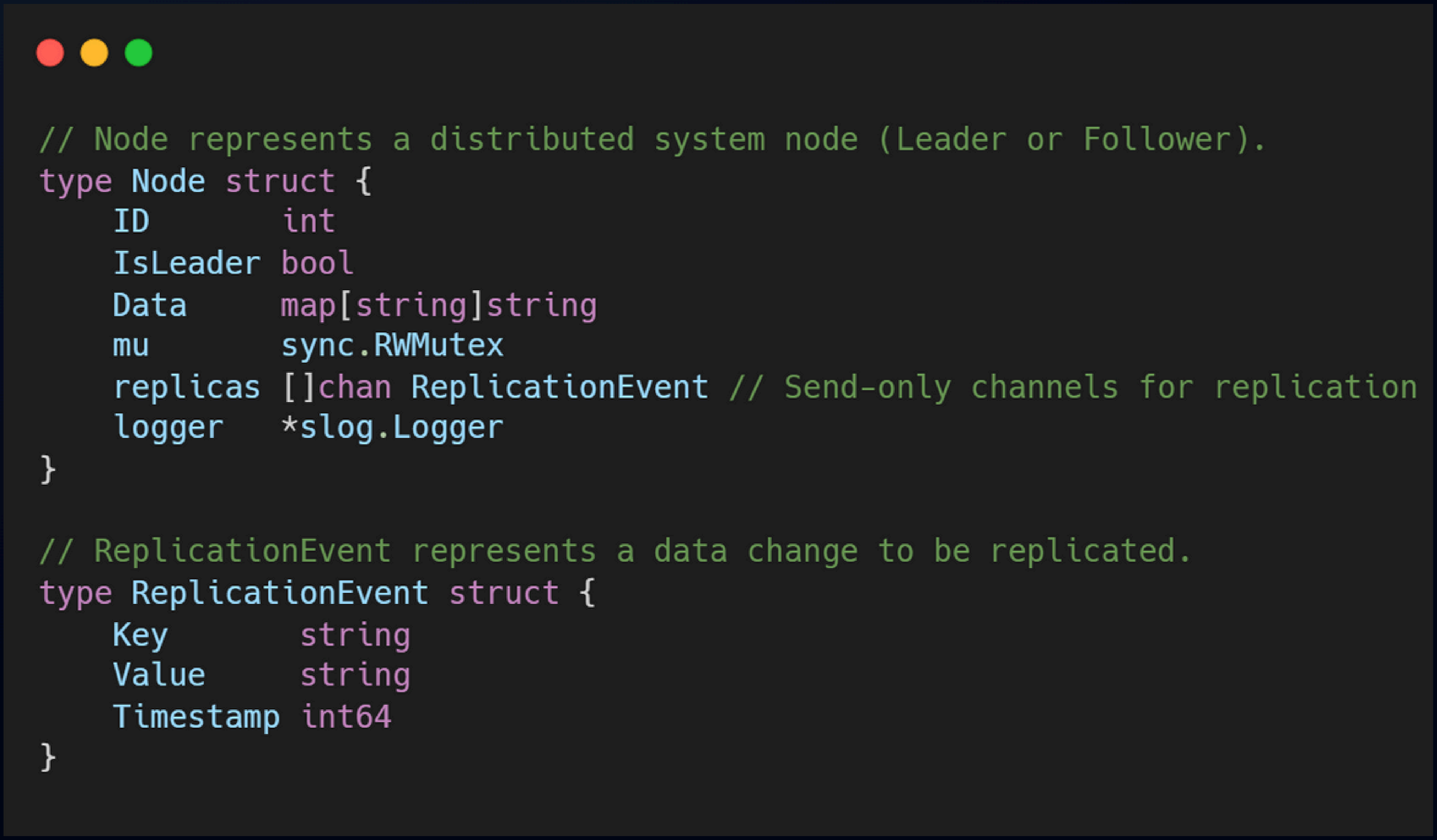
Code: the Node

Modeling the Node

We represent a database node using a Go struct.

- *sync.RWMutex*: Essential for high-performance distributed systems. It allows multiple simultaneous "Reads" but ensures exclusive "Writes."
- *replicas*: These Go channels act as our simulated "network cables" connecting the Leader to its Followers.

Below is the implementation under *node.go*:



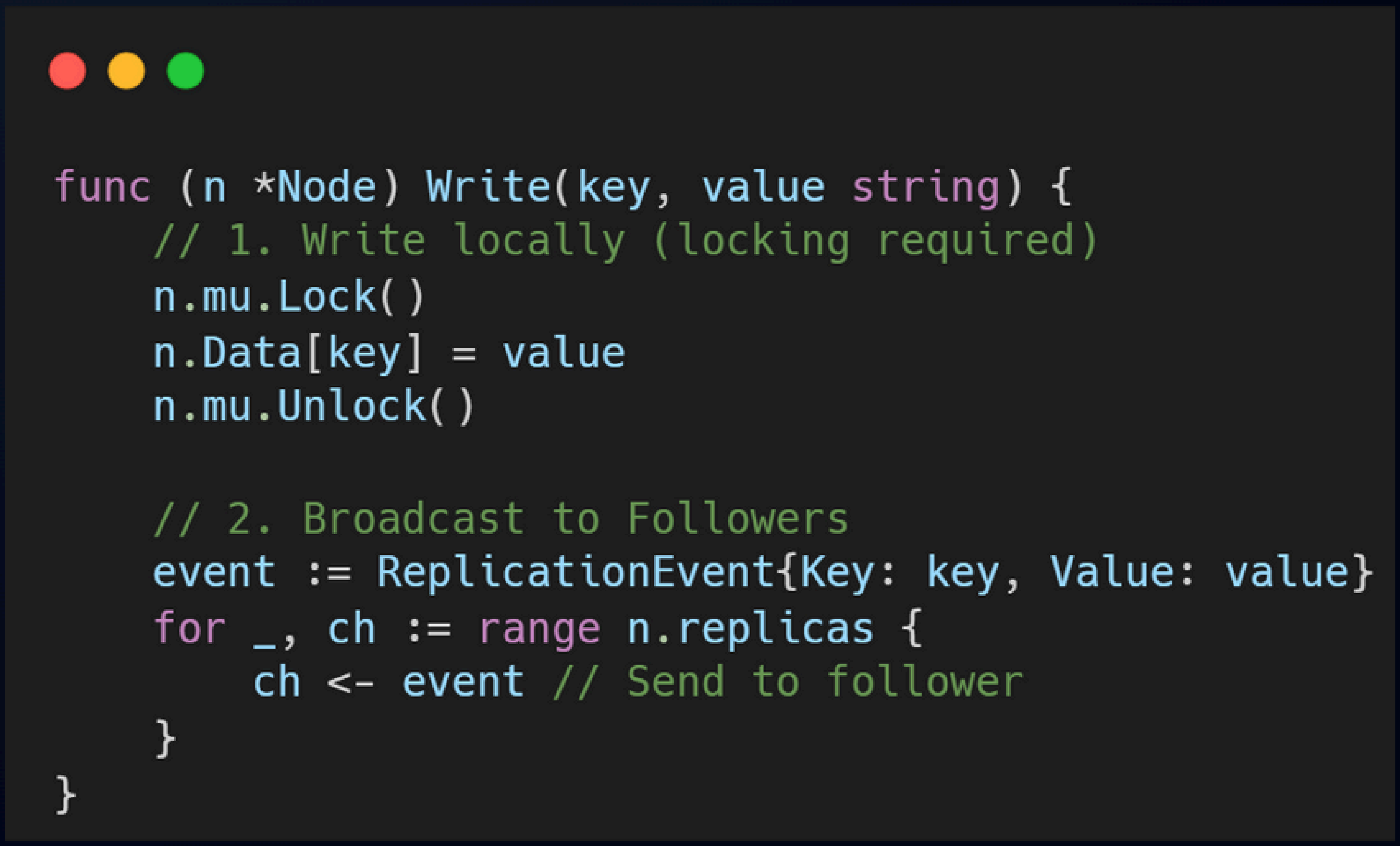
```go
// Node represents a distributed system node (Leader or Follower).
type Node struct {
    ID       int
    IsLeader bool
    Data     map[string]string
    mu       sync.RWMutex
    replicas []chan ReplicationEvent // Send-only channels for replication
    logger   *slog.Logger
}

// ReplicationEvent represents a data change to be replicated.
type ReplicationEvent struct {
    Key       string
    Value     string
    Timestamp int64
}
```

Code: The Leader

The Leader: Write & Broadcast

The Leader writes to its own disk (map) first, then "broadcasts" the change to followers.
Note the use of the map within a lock-consistency starts at the node level.
The broadcast ensures all followers eventually receive the update.

Below is the implementation under *node.go*:



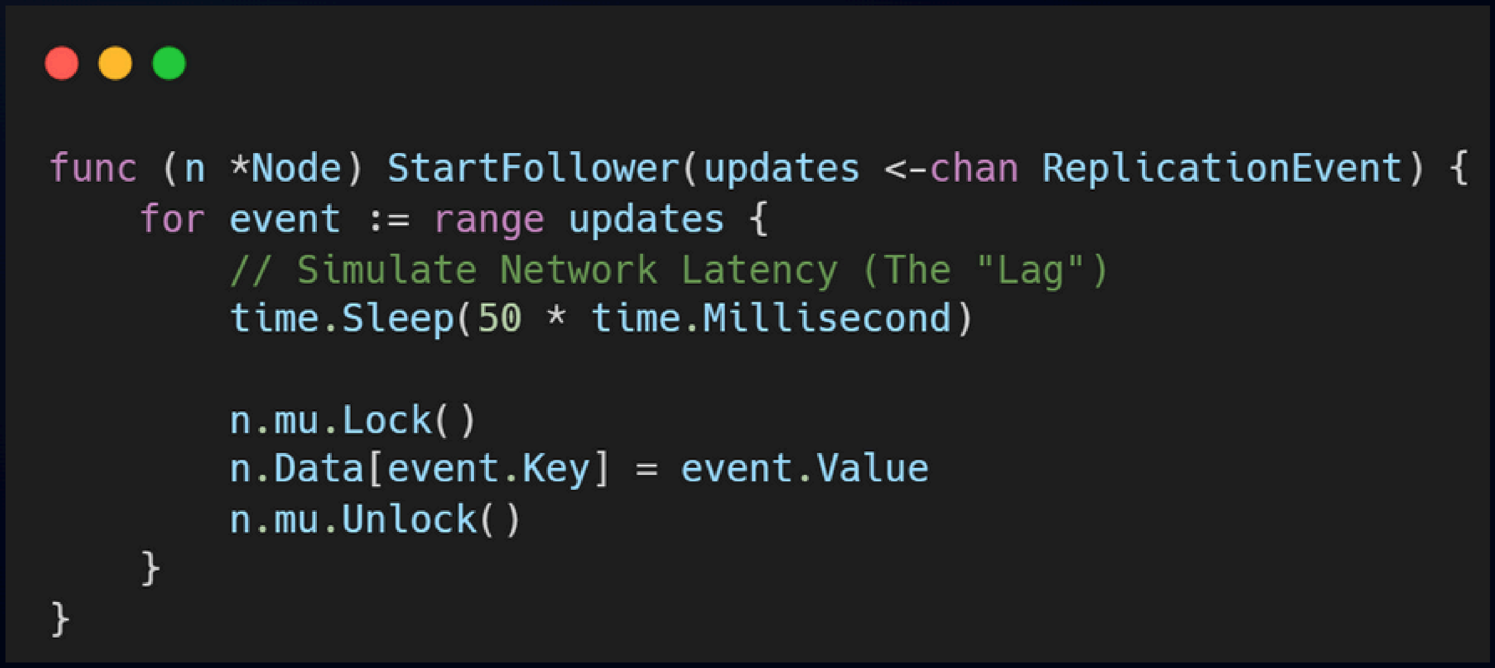
```go
func (n *Node) Write(key, value string) {
    // 1. Write locally (locking required)
    n.mu.Lock()
    n.Data[key] = value
    n.mu.Unlock()

    // 2. Broadcast to Followers
    event := ReplicationEvent{Key: key, Value: value}
    for _, ch := range n.replicas {
        ch <- event // Send to follower
    }
}
```

Code: The Follower

The Follower & "Replication Lag"

Followers listen for events in a background loop. We simulate Network Latency (*time.Sleep*) to demonstrate the reality of distributed systems. This sleep represents the time it takes for data to travel across the wire - in real systems it would take time for the changes to propagate and achieve consistency.
If a client reads from this node before the sleep finishes, they get "stale" data.

Below is the implementation under *node.go*:



```go
func (n *Node) StartFollower(updates <-chan ReplicationEvent) {
    for event := range updates {
        // Simulate Network Latency (The "Lag")
        time.Sleep(50 * time.Millisecond)

        n.mu.Lock()
        n.Data[event.Key] = event.Value
        n.mu.Unlock()
    }
}
```

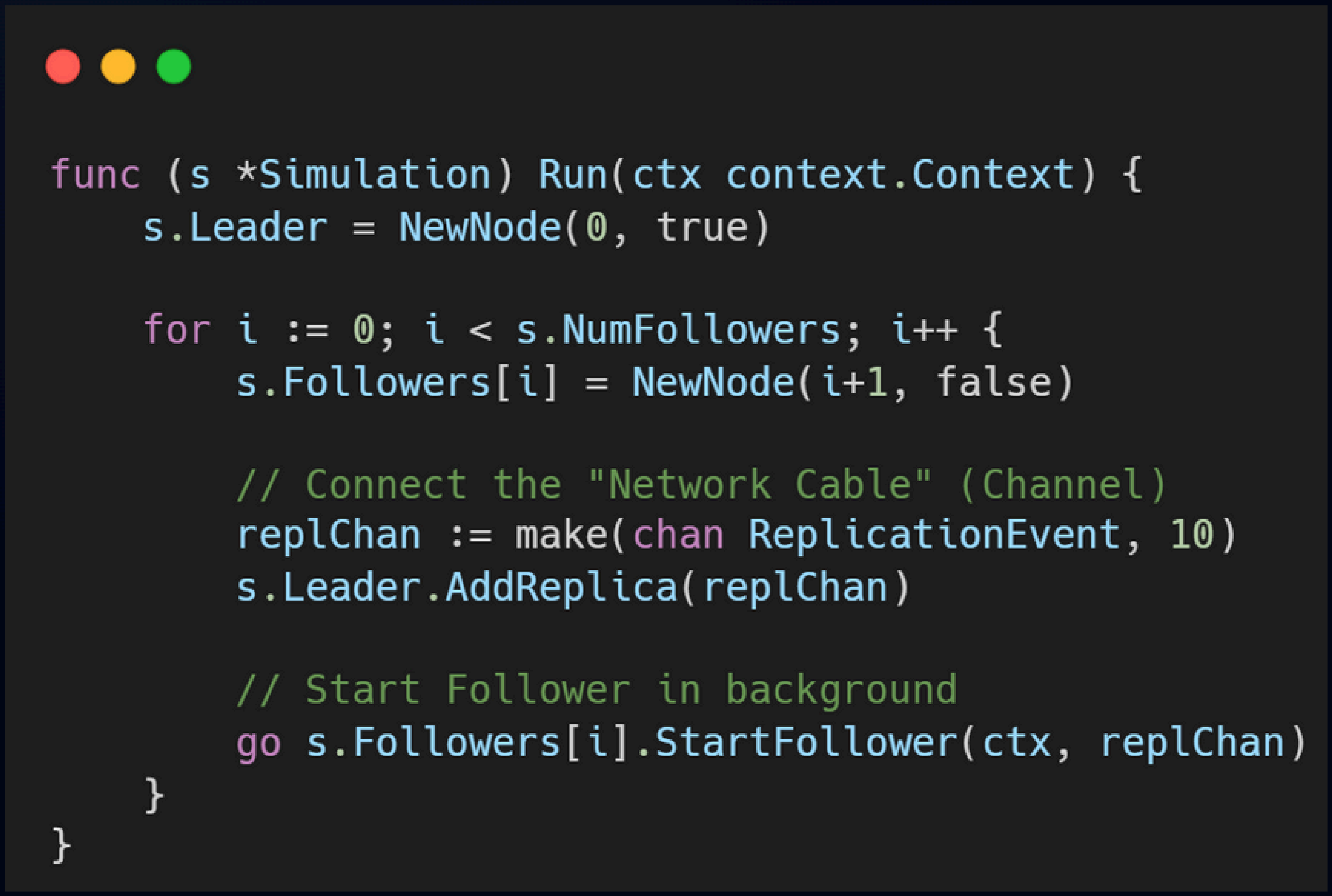
Code: Visualizing the Execution

The Simulation in Action - try it out!

To see theory in action, we need a Simulation Harness.

- **Topology**: We initialize 1 Leader and 3 Followers.
- **Concurrency**: Followers run in background goroutines, listening to their specific "network pipes" (channels).
- **Graceful Shutdown**: We use *context.WithCancel* to ensure that when we hit Ctrl+C, all goroutines exit cleanly - a must-have for any Go developer (Remember that post from the Go Build to Go Run series?).

Below is the implementation under *simulation.go*:



```go
func (s *Simulation) Run(ctx context.Context) {
    s.Leader = NewNode(0, true)

    for i := 0; i < s.NumFollowers; i++ {
        s.Followers[i] = NewNode(i+1, false)

        // Connect the "Network Cable" (Channel)
        replChan := make(chan ReplicationEvent, 10)
        s.Leader.AddReplica(replChan)

        // Start Follower in background
        go s.Followers[i].StartFollower(ctx, replChan)
    }
}
```

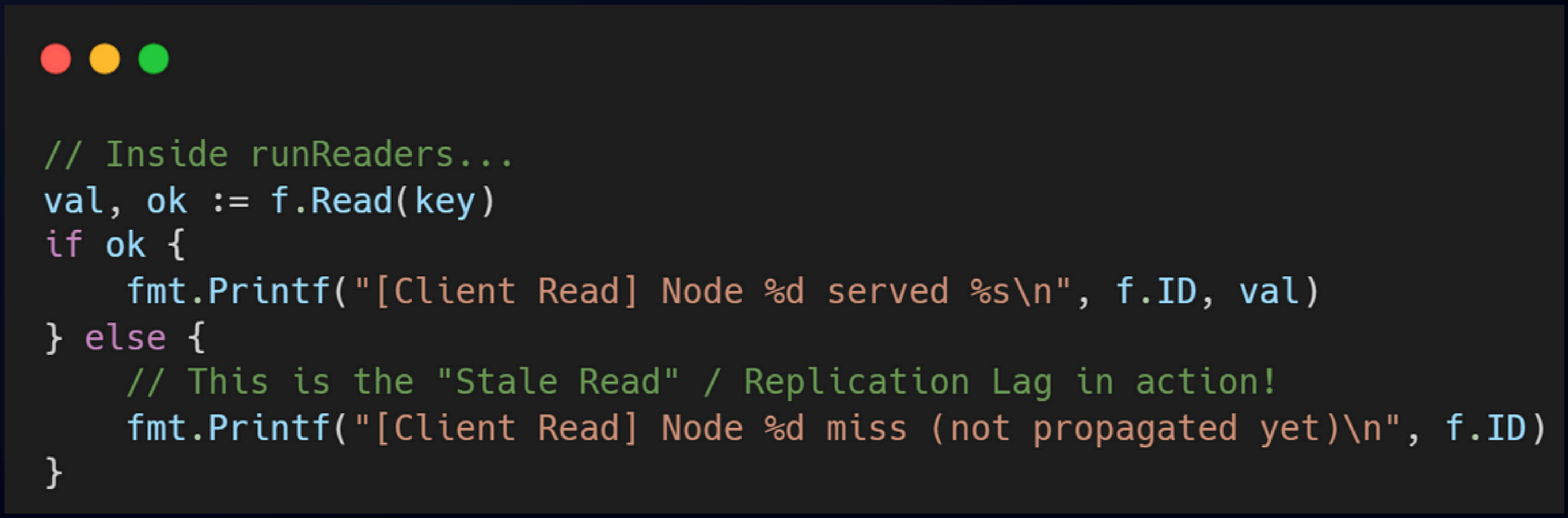
Code: Generating the Load

Stress Testing the System

How do we "prove" eventual consistency?

- **Writers**: Randomly update keys on the Leader.
- **Readers**: Randomly query Followers.
- **The Conflict**: Because of the artificial lag we added in Slide 6, you'll see the **[Client Read]** log report a "Miss" while the Leader has already "Persisted" the data.

Below is the implementation under *simulation.go*:



```go
// Inside runReaders...
val, ok := f.Read(key)
if ok {
    fmt.Printf("[Client Read] Node %d served %s\n", f.ID, val)
} else {
    // This is the "Stale Read" / Replication Lag in action!
    fmt.Printf("[Client Read] Node %d miss (not propagated yet)\n", f.ID)
}
```

Summary & Takeaways

✓ **SCALE READS**

Offload traffic from the primary node to replicas to handle millions of requests.

✓ **CONSISTENCY**

Choose between synchronous or asynchronous replication based on your latency needs.

✓ **HIGH AVAILABILITY**

Maintain uptime even if the primary node fails by promoting a standby replica.

✓ **GO POWER**

Using channels, goroutines, and RWMutex, we can model complex distributed behaviors in under 200 lines of code.

Want to dive deeper?

Explore the full source code and deep-dive documentation on my GitHub repository or connect via LinkedIn.

GitHub LinkedIn

LIKE • COMMENT • SHARE