Rate Limiter Project

Learning Objectives

Build a production-ready rate limiter to learn:

- Token bucket algorithm
- Distributed rate limiting challenges
- Performance optimization (sharding vs global locks)
- Testing concurrent components
- Preventing resource exhaustion

Requirements

Functional Requirements

1. **Token Bucket Algorithm**
    - Fixed rate of token generation (e.g., 100 req/sec)
    - Burst capacity (e.g., allow 200 tokens at once)
    - Tokens refill at constant rate
    - Request blocked if no tokens available
2. **Per-Client Limiting**
    - Different rate limits per client ID
    - Isolated token buckets (client A doesn't affect client B)
    - Efficient lookup (O(1) for millions of clients)
3. **Configuration**
    - Configurable rate and burst per client
    - Dynamic rate limit updates (without restart)
    - Default rate for unknown clients

Non-Functional Requirements

1. **Performance**
    - Handle 100,000+ requests/sec
    - <1ms latency for Allow() check
    - Minimal memory per client (~200 bytes)
2. **Correctness**
    - Thread-safe (no races with -race)
    - Accurate rate limiting (±1% tolerance)
    - No token leakage or overflow
3. **Observability**
    - Metrics (allowed, denied, active clients)
    - Debug info (current tokens, last refill)

Three Implementations

1. Naive Implementation (naive/rate_limiter.go)

Approach:

- Global mutex protecting all client buckets
- Map of client ID → bucket state
- Synchronous token refill on every request

Problems (Intentional):

- **Global lock contention:** All clients wait on single mutex
- **No cleanup:** Memory leak from abandoned clients
- **Inefficient refill:** Calculates tokens every request
- **Poor performance:** Mutex serializes all operations

Expected Issues:

- High contency with >10 concurrent clients
- Memory grows unbounded
- CPU waste on repeated calculations
- Throughput plateaus at ~50k req/sec

2. Improved Implementation (improved/rate_limiter.go)

Fixes:

- **Sharding:** Split clients across 256 shards (reduce contention)
- **Lazy refill:** Calculate tokens on-demand, not every nanosecond
- **Periodic cleanup:** Background goroutine removes stale clients
- **sync.Map alternative:** Try lock-free reads

Improvements:

- Throughput increases to ~200k req/sec
- Memory bounded by cleanup
- Better CPU utilization

Remaining Issues:

- Still uses mutexes (not lock-free)
- Cleanup timing is arbitrary (might clean active clients)
- No metrics for observability

3. Final Implementation (final/rate_limiter.go)

Production-Ready:

- **Sharded architecture:** 256 shards with RWMutex per shard
- **Atomic operations:** Lock-free token checks where possible
- **Smart cleanup:** Track last access time, remove after inactivity
- **Metrics:** Prometheus-style counters (allowed, denied, clients)
- **Testing:** Comprehensive tests with race detector
- **Benchmarks:** Prove scalability with b.RunParallel

Key Optimizations:

```
// Sharding reduces contention by 256x
shard := fnv32(clientID) % numShards

// RWMutex: parallel reads, exclusive writes
shard.RLock() // Fast read path
defer shard.RUnlock()

// Atomic counters for metrics (lock-free)
atomic.AddUint64(&rl.metrics.allowed, 1)
```

Performance Characteristics:

- Throughput: ~500k req/sec (10 clients)
- Latency: p50=0.5µs, p99=5µs
- Memory: Fixed overhead + 200 bytes/active client
- Cleanup: Configurable TTL (default 5min)

Success Criteria

By the end, you should be able to:

1. **Explain the progression:**
    - Why does sharding improve performance?
    - When does RWMutex outperform Mutex?
    - How does cleanup prevent memory leaks?
2. **Identify bugs:**
    - Spot race conditions in naive version
    - Explain contention with global mutex
    - Calculate memory leak growth rate
3. **Optimize for production:**
    - Choose shard count (power of 2 for fast modulo)
    - Tune cleanup interval vs memory overhead
    - Add metrics for observability
4. **Test thoroughly:**
    - Write stress tests (1000 goroutines)
    - Run with -race detector
    - Benchmark and prove scalability

Usage Examples

Naive Implementation

```
rl := naive.NewRateLimiter()
rl.SetLimit("client1", 100, 200) // 100/sec, burst 200

if rl.Allow("client1") {
```

```
    // Process request
} else {
    // Rate limited, return 429
}
```

Final Implementation

```
rl := final.NewRateLimiter(final.Config{
    DefaultRate:    100,
    DefaultBurst:   200,
    CleanupInterval: 5 * time.Minute,
    InactivityTTL:  10 * time.Minute,
    NumShards:      256,
})

// Check if request allowed
if rl.Allow("client1") {
    handleRequest()
} else {
    http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
}

// Get metrics
metrics := rl.Metrics()
fmt.Printf("Allowed: %d, Denied: %d, Clients: %d\n",
    metrics.Allowed, metrics.Denied, metrics.ActiveClients)
```

Testing Strategy

Unit Tests

```
cd final
go test -v                  # Run all tests
go test -race               # Check for races
go test -cover              # Check coverage
```

Stress Tests

```
func TestStress(t *testing.T) {
    rl := NewRateLimiter(Config{...})

    var wg sync.WaitGroup
    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()
            clientID := fmt.Sprintf("client%d", id%100)
            for j := 0; j < 10000; j++ {
```

```
                rl.Allow(clientID)
            }
        }(i)
    }
    wg.Wait()

    // Verify no races, correct counts
}
```

Benchmarks

```
cd final
go test -bench=. -benchmem
go test -bench=. -benchmem -cpuprofile=cpu.prof
go tool pprof cpu.prof
```

Expected output:

```
BenchmarkAllow/1-client-8      10000000    150 ns/op    0 B/op    0 allocs/op
BenchmarkAllow/10-clients-8     5000000    300 ns/op    0 B/op    0 allocs/op
BenchmarkAllow/100-clients-8    3000000    450 ns/op    0 B/op    0 allocs/op
```

Real-World Scenarios

1. HTTP API Rate Limiting

```
func rateLimitMiddleware(rl *final.RateLimiter) func(http.Handler) http.Handler {
    return func(next http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            clientID := r.Header.Get("X-API-Key")
            if clientID == "" {
                clientID = r.RemoteAddr
            }

            if !rl.Allow(clientID) {
                w.Header().Set("Retry-After", "1")
                http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
                return
            }

            next.ServeHTTP(w, r)
        })
    }
}
```

2. Database Query Rate Limiting

```
// Prevent query flooding
type QueryLimiter struct {
    rl *final.RateLimiter
}

func (ql *QueryLimiter) Query(ctx context.Context, userID string, query string)
error {
    if !ql.rl.Allow(userID) {
        return ErrRateLimitExceeded
    }
    return db.Query(ctx, query)
}
```

3. Background Job Throttling

```
// Limit concurrent jobs per tenant
func processBatch(rl *final.RateLimiter, tenantID string, items []Item) {
    for _, item := range items {
        // Wait if rate limited
        for !rl.Allow(tenantID) {
            time.Sleep(10 * time.Millisecond)
        }
        processItem(item)
    }
}
```

Common Pitfalls

Pitfall 1: Time-Based Race

```
// ❌ WRONG: lastRefill can get stale between read and write
lastRefill := bucket.lastRefill
elapsed := now.Sub(lastRefill)
bucket.lastRefill = now // Race! Another goroutine might have updated

// ✅ CORRECT: Read and write under lock
mu.Lock()
elapsed := now.Sub(bucket.lastRefill)
bucket.lastRefill = now
mu.Unlock()
```

Pitfall 2: Floating Point Tokens

```
// ❌ WRONG: Float precision issues
tokens := 10.5 + 0.1 + 0.1 + 0.1 // != 10.8 due to float precision

// ✅ CORRECT: Use integers (tokens in nanoseconds)
```

```
tokens := int64(10_000_000_000) // 10 seconds in nanoseconds
tokensPerNs := rate / 1e9
```

Pitfall 3: Integer Overflow

```
// ❌ WRONG: elapsed * rate can overflow int64
elapsed := time.Since(lastRefill).Nanoseconds() // Large number
newTokens := elapsed * rate / 1e9 // Overflow!

// ✅ CORRECT: Use time.Duration arithmetic
elapsed := time.Since(lastRefill)
newTokens := int64(elapsed.Seconds() * float64(rate))
```

Pitfall 4: Unbounded Map Growth

```
// ❌ WRONG: Map grows forever
clients := make(map[string]*Bucket)
clients[clientID] = newBucket() // Never removed

// ✅ CORRECT: Periodic cleanup
go func() {
    ticker := time.NewTicker(5 * time.Minute)
    for range ticker.C {
        removeInactiveClients(clients, 10*time.Minute)
    }
}()
```

Further Reading

After completing this project:

1. **Try golang.org/x/time/rate**
    - Compare your implementation
    - Study their optimizations
    - Use in production (battle-tested)
2. **Distributed Rate Limiting**
    - Redis-based rate limiter (single source of truth)
    - Sliding window algorithm (more accurate)
    - Distributed token bucket (coordination overhead)
3. **Alternative Algorithms**
    - Leaky bucket (smooth output rate)
    - Fixed window (simple but bursty)
    - Sliding window (accurate but memory-intensive)
4. **Production Considerations**
    - Circuit breaker integration

- Gradual limit increases (warm-up)
- Per-endpoint limits (not just per-client)
- Cost-based limiting (expensive operations cost more tokens)

Interview Discussion Points

Be prepared to explain:

1. **Why token bucket over fixed window?**
   - Allows bursts (better UX)
   - Smooth rate limiting (no boundary effects)
   - Mathematically sound
2. **Why sharding improves performance?**
   - Reduces lock contention
   - Parallel non-conflicting operations
   - Trade-off: complexity vs throughput
3. **When to use RWMutex vs Mutex?**
   - Read-heavy workload (Check > Update)
   - Benchmark to verify (RWMutex has overhead)
   - In rate limiter: reads ~ writes, so Mutex might be fine
4. **How to prevent memory leaks?**
   - Periodic cleanup with TTL
   - LRU eviction if max clients exceeded
   - Trade-off: cleanup frequency vs memory spikes
5. **How to test correctness?**
   - Race detector (-race flag)
   - Stress tests (1000 goroutines × 10000 ops)
   - Accuracy tests (allow exactly N requests in T seconds ±1%)
   - Benchmark for performance regression

Next Steps

After mastering rate-limiter:

1. **job-queue/** - Worker pool with priorities and persistence
2. **cache/** - Concurrent LRU cache with sharding
3. **web-crawler/** - Bounded concurrency and politeness
4. **connection-pool/** - Database pool with circuit breaker
5. **pub-sub/** - Topic-based messaging with fan-out

Let's build production-ready concurrent systems! 🚀