Rate Limiter Pattern

What is Rate Limiting?

Rate limiting: Controlling the rate at which operations occur, typically measured in operations per time unit (requests/second, events/minute, etc.).

Purpose:

- **Protect services:** Prevent overload from too many requests
- **Fair usage:** Ensure all clients get fair share
- **Cost control:** Limit expensive operations (API calls, database writes)
- **Compliance:** Respect third-party API rate limits

Common Algorithms

1. Token Bucket

Concept: Bucket holds tokens. Each operation consumes token. Tokens refill at fixed rate.

Properties:

- Allows bursts (if bucket has tokens)
- Smooth long-term rate
- Most common in production

```
type TokenBucket struct {
    tokens     float64
    capacity   float64
    rate       float64  // Tokens per second
    lastRefill time.Time
    mu         sync.Mutex
}

func NewTokenBucket(capacity, rate float64) *TokenBucket {
    return &TokenBucket{
        tokens:     capacity,
        capacity:   capacity,
        rate:       rate,
        lastRefill: time.Now(),
    }
}

func (tb *TokenBucket) Allow() bool {
    tb.mu.Lock()
    defer tb.mu.Unlock()

    // Refill tokens
    now := time.Now()
    elapsed := now.Sub(tb.lastRefill).Seconds()
    tb.tokens += elapsed * tb.rate
    if tb.tokens > tb.capacity {
```

```
        tb.tokens = tb.capacity
    }
    tb.lastRefill = now

    // Check if token available
    if tb.tokens >= 1 {
        tb.tokens--
        return true
    }

    return false
}

// Usage:
limiter := NewTokenBucket(10, 5)  // Capacity 10, refill 5/sec

for _, request := range requests {
    if limiter.Allow() {
        process(request)
    } else {
        reject(request)
    }
}
```

2. Leaky Bucket

Concept: Requests enter bucket, processed at fixed rate. Bucket overflows if full.

Properties:

- Smooths bursty traffic
- Fixed output rate
- Good for protecting downstream

```
type LeakyBucket struct {
    queue    chan Request
    rate     time.Duration
    capacity int
}

func NewLeakyBucket(capacity int, rate time.Duration) *LeakyBucket {
    lb := &LeakyBucket{
        queue:    make(chan Request, capacity),
        rate:     rate,
        capacity: capacity,
    }

    go lb.process()
    return lb
}

func (lb *LeakyBucket) process() {
```

```
    ticker := time.NewTicker(lb.rate)
    defer ticker.Stop()

    for range ticker.C {
        select {
        case req := <-lb.queue:
            handleRequest(req)
        default:
            // Queue empty
        }
    }
}

func (lb *LeakyBucket) Add(req Request) bool {
    select {
    case lb.queue <- req:
        return true  // Added to queue
    default:
        return false  // Queue full, drop request
    }
}
```

3. Fixed Window

Concept: Count requests in time window. Reset counter at window boundary.

Properties:

- Simple implementation
- Potential burst at window boundary (boundary problem)

```
type FixedWindow struct {
    limit       int
    window      time.Duration
    count       int
    windowStart time.Time
    mu          sync.Mutex
}

func NewFixedWindow(limit int, window time.Duration) *FixedWindow {
    return &FixedWindow{
        limit:       limit,
        window:      window,
        windowStart: time.Now(),
    }
}

func (fw *FixedWindow) Allow() bool {
    fw.mu.Lock()
    defer fw.mu.Unlock()

    now := time.Now()
```

```
    // Check if new window
    if now.Sub(fw.windowStart) >= fw.window {
        fw.count = 0
        fw.windowStart = now
    }

    // Check limit
    if fw.count < fw.limit {
        fw.count++
        return true
    }

    return false
}

// Problem: Burst at window boundary
// Time:  00:00:00 - 00:00:59 - 00:01:00 - 00:01:59
// Count:    100         100         100         100
// In 2 seconds (00:00:59 to 00:01:00): 200 requests!
```

4. Sliding Window

Concept: Track requests with timestamps. Count requests in rolling time window.

Properties:

- No boundary problem
- More memory (store timestamps)
- More accurate

```
type SlidingWindow struct {
    limit      int
    window     time.Duration
    requests   []time.Time
    mu         sync.Mutex
}

func NewSlidingWindow(limit int, window time.Duration) *SlidingWindow {
    return &SlidingWindow{
        limit:    limit,
        window:   window,
        requests: make([]time.Time, 0, limit),
    }
}

func (sw *SlidingWindow) Allow() bool {
    sw.mu.Lock()
    defer sw.mu.Unlock()

    now := time.Now()
    cutoff := now.Add(-sw.window)
```

```
    // Remove expired requests
    i := 0
    for ; i < len(sw.requests); i++ {
        if sw.requests[i].After(cutoff) {
            break
        }
    }
    sw.requests = sw.requests[i:]

    // Check limit
    if len(sw.requests) < sw.limit {
        sw.requests = append(sw.requests, now)
        return true
    }

    return false
}
```

Standard Library: golang.org/x/time/rate

```
import "golang.org/x/time/rate"

// Create limiter: 10 requests per second, burst of 20
limiter := rate.NewLimiter(10, 20)

// Wait for token (blocks)
err := limiter.Wait(context.Background())

// Try to take token (non-blocking)
if limiter.Allow() {
    // Process request
}

// Reserve token (returns reservation)
r := limiter.Reserve()
if !r.OK() {
    // Rate limit exceeded
    return
}
time.Sleep(r.Delay())  // Wait if necessary
```

Features:

- Token bucket algorithm
- Context support
- Dynamic rate adjustment
- Burst handling

Real-World Example: HTTP Rate Limiter Middleware

```go
type RateLimiter struct {
    limiters map[string]*rate.Limiter
    mu       sync.Mutex
    rate     rate.Limit
    burst    int
}

func NewRateLimiter(rps, burst int) *RateLimiter {
    return &RateLimiter{
        limiters: make(map[string]*rate.Limiter),
        rate:     rate.Limit(rps),
        burst:    burst,
    }
}

func (rl *RateLimiter) getLimiter(key string) *rate.Limiter {
    rl.mu.Lock()
    defer rl.mu.Unlock()

    limiter, exists := rl.limiters[key]
    if !exists {
        limiter = rate.NewLimiter(rl.rate, rl.burst)
        rl.limiters[key] = limiter
    }

    return limiter
}

func (rl *RateLimiter) Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // Rate limit by IP
        ip := r.RemoteAddr
        limiter := rl.getLimiter(ip)

        if !limiter.Allow() {
            http.Error(w, "Rate limit exceeded", http.StatusTooManyRequests)
            return
        }

        next.ServeHTTP(w, r)
    })
}

// Usage:
rateLimiter := NewRateLimiter(100, 200)  // 100 req/sec, burst 200

mux := http.NewServeMux()
mux.HandleFunc("/api", apiHandler)

http.ListenAndServe(":8080", rateLimiter.Middleware(mux))
```

Per-User Rate Limiting

```
type UserRateLimiter struct {
    limiters sync.Map  // map[userID]*rate.Limiter
    rate     rate.Limit
    burst    int
}

func (url *UserRateLimiter) Allow(userID string) bool {
    limiterI, _ := url.limiters.LoadOrStore(userID,
        rate.NewLimiter(url.rate, url.burst))

    limiter := limiterI.(*rate.Limiter)
    return limiter.Allow()
}

// Cleanup: Remove inactive users periodically
func (url *UserRateLimiter) cleanup(inactiveThreshold time.Duration) {
    ticker := time.NewTicker(time.Minute)
    defer ticker.Stop()

    for range ticker.C {
        now := time.Now()
        url.limiters.Range(func(key, value interface{}) bool {
            limiter := value.(*rate.Limiter)

            // Check last token consumption (requires custom limiter)
            if now.Sub(limiter.LastUsed()) > inactiveThreshold {
                url.limiters.Delete(key)
            }

            return true
        })
    }
}
```

Distributed Rate Limiting

For multi-instance deployments, use shared state (Redis, etc.).

```
type DistributedRateLimiter struct {
    redis  *redis.Client
    key    string
    limit  int
    window time.Duration
}

func (drl *DistributedRateLimiter) Allow(ctx context.Context) bool {
    script := `
```

```
        local current = redis.call('INCR', KEYS[1])
        if current == 1 then
            redis.call('EXPIRE', KEYS[1], ARGV[1])
        end
        return current
    `

    count, err := drl.redis.Eval(ctx, script,
        []string{drl.key},
        int(drl.window.Seconds())).Int()

    if err != nil {
        return false
    }

    return count <= drl.limit
}

// Usage:
limiter := &DistributedRateLimiter{
    redis:  redisClient,
    key:    "rate_limit:api",
    limit:  1000,
    window: time.Minute,
}

if limiter.Allow(ctx) {
    // Process request
}
```

Adaptive Rate Limiting

Adjust rate based on system load.

```
type AdaptiveRateLimiter struct {
    limiter      *rate.Limiter
    targetCPU    float64
    adjustTicker *time.Ticker
    mu           sync.RWMutex
}

func NewAdaptiveRateLimiter(initialRate float64, targetCPU float64)
*AdaptiveRateLimiter {
    arl := &AdaptiveRateLimiter{
        limiter:      rate.NewLimiter(rate.Limit(initialRate), int(initialRate)),
        targetCPU:    targetCPU,
        adjustTicker: time.NewTicker(10 * time.Second),
    }

    go arl.adjustRate()
    return arl
```

```
}

func (arl *AdaptiveRateLimiter) adjustRate() {
    for range arl.adjustTicker.C {
        cpuUsage := getCurrentCPUUsage()

        arl.mu.Lock()
        currentRate := float64(arl.limiter.Limit())

        if cpuUsage > arl.targetCPU*1.1 {
            // Decrease rate by 20%
            newRate := currentRate * 0.8
            arl.limiter.SetLimit(rate.Limit(newRate))
        } else if cpuUsage < arl.targetCPU*0.9 {
            // Increase rate by 10%
            newRate := currentRate * 1.1
            arl.limiter.SetLimit(rate.Limit(newRate))
        }

        arl.mu.Unlock()
    }
}

func (arl *AdaptiveRateLimiter) Allow() bool {
    arl.mu.RLock()
    defer arl.mu.RUnlock()
    return arl.limiter.Allow()
}
```

Rate Limiting with Retry

```
func callWithRateLimit(ctx context.Context, limiter *rate.Limiter, fn func() error)
error {
    maxRetries := 3
    backoff := time.Second

    for i := 0; i < maxRetries; i++ {
        // Wait for rate limit
        err := limiter.Wait(ctx)
        if err != nil {
            return err
        }

        // Call function
        err = fn()
        if err == nil {
            return nil
        }

        // Check if rate limit error
```

```
        if isRateLimitError(err) {
            time.Sleep(backoff)
            backoff *= 2
            continue
        }

        return err
    }

    return fmt.Errorf("max retries exceeded")
}
```

Comparing Algorithms

Algorithm	Burst Handling	Memory	Accuracy	Best For
Token Bucket	✓ Allows bursts	Low	Good	APIs, general use
Leaky Bucket	✗ Smooths bursts	Medium	Excellent	Protecting downstream
Fixed Window	✗ Boundary problem	Low	Poor	Simple use cases
Sliding Window	✓ No boundary issue	High	Excellent	Financial, analytics

Common Mistakes

Mistake 1: Forgetting to Handle Rejection

```
// WRONG: Blocks forever if rate limited
for _, req := range requests {
    limiter.Wait(ctx)  // Blocks
    process(req)
}

// Better: Check and retry later
for _, req := range requests {
    if limiter.Allow() {
        process(req)
    } else {
        // Queue for retry, return 429, etc.
        queueRetry(req)
    }
}
```

Mistake 2: Per-Request Limiter Creation

```
// WRONG: Creates new limiter per request
func handler(w http.ResponseWriter, r *http.Request) {
    limiter := rate.NewLimiter(10, 20)  // New limiter!
    if !limiter.Allow() {
```

```
        // This never triggers (new limiter has tokens)
    }
}

// Right: Shared limiter
var globalLimiter = rate.NewLimiter(10, 20)

func handler(w http.ResponseWriter, r *http.Request) {
    if !globalLimiter.Allow() {
        http.Error(w, "Rate limited", 429)
    }
}
```

Mistake 3: Not Considering Burst

```
// WRONG: No burst capacity
limiter := rate.NewLimiter(10, 1)  // Burst=1

// Can't handle any bursts
// 11 simultaneous requests all rejected except 1
```

Fix:

```
// Allow reasonable burst
limiter := rate.NewLimiter(10, 20)  // 10/sec, burst up to 20
```

Performance Benchmarks

```
func BenchmarkTokenBucket(b *testing.B) {
    limiter := NewTokenBucket(1000, 1000)

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            limiter.Allow()
        }
    })
}

func BenchmarkStdlibRateLimiter(b *testing.B) {
    limiter := rate.NewLimiter(1000, 1000)

    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            limiter.Allow()
        }
    })
}
```

```
// Results:
// TokenBucket:    ~50 ns/op (simple implementation)
// Stdlib:         ~100 ns/op (more features)
```

Interview Questions

Q: "Token bucket vs. leaky bucket?"

"Token bucket allows bursts if tokens available, refills at fixed rate. Leaky bucket smooths bursts, outputs at fixed rate regardless of input. Token bucket better for APIs (allow burst, then throttle). Leaky bucket better for protecting downstream (constant output rate). Implementation: token bucket uses counter + time; leaky bucket uses queue + ticker."

Q: "How do you rate limit in distributed system?"

"Use shared state (Redis, memcached). Fixed window: INCR key with TTL. Sliding window: sorted set with timestamps. Token bucket: store token count + last refill time, atomic ops. Trade-off: distributed state adds latency (~~1-5ms per check) vs. local rate limiter (~~100ns). Use local + distributed: local limiter for fast path, distributed for global limit."

Q: "What's the boundary problem in fixed window?"

"Fixed window resets count at window boundary. Example: 100 req/min limit, 100 requests at 00:00:59, 100 requests at 00:01:00 → 200 requests in 1 second (59s to 00s). Sliding window solves this by tracking exact timestamps, no artificial boundaries. Trade-off: sliding window uses more memory (store all request times)."

Key Takeaways

1. **Token bucket = allow bursts, smooth long-term**
2. **Leaky bucket = smooth output, no bursts**
3. **Fixed window = simple but boundary problem**
4. **Sliding window = accurate but more memory**
5. **Use golang.org/x/time/rate for production**
6. **Per-user limiting requires map of limiters**
7. **Distributed limiting needs shared state (Redis)**
8. **Adaptive limiting adjusts to system load**
9. **Always handle rejection (429, retry, queue)**
10. **Burst size = 1-2x rate for normal traffic**

Exercises

1. Implement token bucket from scratch, benchmark against stdlib.
2. Build HTTP middleware with per-IP rate limiting (100 req/min).
3. Create distributed rate limiter using Redis.
4. Implement adaptive rate limiter that adjusts based on response time.
5. Compare fixed window vs. sliding window: create traffic pattern that exploits boundary problem.

Next: context-cancellation.md - Graceful cancellation with context.Context.