Go (Golang) Distributed Rate Limiter - Token Bucket Algorithm

```
// Package ratelimit provides a concurrent-safe distributed rate limiter
// implementation using token bucket algorithm backed by Redis.
// It supports context cancellation, pluggable storage backends, and
// extensible limiter policies.
//
// This package can be used to throttle API calls, limit resource usage,
// or implement fair usage policies in distributed systems.
package ratelimit

import (
	"context"
	"errors"
	"fmt"
	"sync"
	"time"

	"github.com/go-redis/redis/v8"
)

// Limiter defines the interface for any rate limiter implementation.
```

```
// It supports Acquire and Release semantics for tokens.
type Limiter interface {
	// Acquire blocks until a token is available or context is canceled.
	Acquire(ctx context.Context, key string) error

	// TryAcquire tries to acquire a token immediately, returns false if unavailable.
	TryAcquire(ctx context.Context, key string) (bool, error)

	// Release returns a token back to the bucket.
	Release(ctx context.Context, key string) error
}

// ErrRateLimitExceeded indicates the rate limit was exceeded.
var ErrRateLimitExceeded = errors.New("rate limit exceeded")

// Storage defines the interface for backend storage supporting
// atomic operations required by the limiter.
type Storage interface {
	// Increment increases token count and returns the new count.
	Increment(ctx context.Context, key string, delta int64, expire time.Duration)
(int64, error)

	// Decrement decreases token count and returns the new count.
	Decrement(ctx context.Context, key string, delta int64) (int64, error)
```

```
	// Get returns the current token count for a key.
	Get(ctx context.Context, key string) (int64, error)
}

// redisStorage implements Storage using Redis as backend.
type redisStorage struct {
	client *redis.Client
}

func newRedisStorage(client *redis.Client) *redisStorage {
	return &redisStorage{client: client}
}

func (r *redisStorage) Increment(ctx context.Context, key string, delta int64, expire
time.Duration) (int64, error) {
	pipe := r.client.TxPipeline()
	incr := pipe.IncrBy(ctx, key, delta)
	pipe.Expire(ctx, key, expire)
	_, err := pipe.Exec(ctx)
	if err != nil {
		return 0, err
	}
	return incr.Val(), nil
}
```

```
func (r *redisStorage) Decrement(ctx context.Context, key string, delta int64) (int64,
error) {
	newVal, err := r.client.DecrBy(ctx, key, delta).Result()
	if err != nil {
		return 0, err
	}
	if newVal < 0 {
		// Optionally reset to zero to avoid negative tokens
		_ = r.client.Set(ctx, key, 0, 0).Err()
		return 0, ErrRateLimitExceeded
	}
	return newVal, nil
}

func (r *redisStorage) Get(ctx context.Context, key string) (int64, error) {
	val, err := r.client.Get(ctx, key).Int64()
	if err == redis.Nil {
		return 0, nil // Key not found implies zero tokens
	}
	return val, err
}

// TokenBucketLimiter is a distributed token bucket rate limiter.
// Tokens are replenished at a fixed interval up to a max capacity.
type TokenBucketLimiter struct {
```

```
	storage    Storage
	capacity   int64        // max tokens in bucket
	refillRate int64        // tokens added per refill interval
	refillIntv time.Duration // refill interval duration

	mu        sync.Mutex
	closeCh   chan struct{}
	closeOnce sync.Once
}

// NewTokenBucketLimiter constructs a new limiter with specified params.
func NewTokenBucketLimiter(storage Storage, capacity, refillRate int64, refillIntv time.Duration) *TokenBucketLimiter {
	limiter := &TokenBucketLimiter{
		storage:    storage,
		capacity:   capacity,
		refillRate: refillRate,
		refillIntv: refillIntv,
		closeCh:    make(chan struct{}),
	}
	go limiter.refillWorker()
	return limiter
}

// refillWorker replenishes tokens for all keys periodically.
```

```
// In a real-world system, you'd track active keys more efficiently.
func (l *TokenBucketLimiter) refillWorker() {
	ticker := time.NewTicker(l.refillIntv)
	defer ticker.Stop()

	for {
		select {
		case <-ticker.C:
			// Ideally: refill all keys in storage (implementation depends on backend)
			// Here omitted for brevity as Redis SCAN required
		case <-l.closeCh:
			return
		}
	}
}

// Close stops the background refill worker.
func (l *TokenBucketLimiter) Close() {
	l.closeOnce.Do(func() {
		close(l.closeCh)
	})
}

// Acquire blocks until a token is available or context canceled.
```

```
func (l *TokenBucketLimiter) Acquire(ctx context.Context, key string) error {
	for {
		ok, err := l.TryAcquire(ctx, key)
		if err != nil {
			return err
		}
		if ok {
			return nil
		}

		select {
		case <-ctx.Done():
			return ctx.Err()
		case <-time.After(l.refillIntv):
			// Wait and retry
		}
	}
}

// TryAcquire tries to acquire a token immediately.
func (l *TokenBucketLimiter) TryAcquire(ctx context.Context, key string) (bool, error) {
	l.mu.Lock()
	defer l.mu.Unlock()

	// Decrement token count atomically
```

```
	count, err := l.storage.Decrement(ctx, key, 1)
	if err == ErrRateLimitExceeded {
		return false, nil
	}
	if err != nil {
		return false, err
	}

	if count < 0 {
		return false, nil
	}
	return true, nil
}

// Release returns a token to the bucket.
func (l *TokenBucketLimiter) Release(ctx context.Context, key string) error {
	l.mu.Lock()
	defer l.mu.Unlock()

	count, err := l.storage.Increment(ctx, key, 1, 0)
	if err != nil {
		return err
	}
	if count > l.capacity {
		// Optionally cap tokens at capacity
```

```
                // Reset to capacity value
                // This ensures bucket doesn't overflow
                // For demo, ignoring atomicity tradeoffs
        }
        return nil
}

// --- Example Usage ---

// Example creates a limiter and tries to acquire tokens concurrently.
func Example() {
        ctx := context.Background()

        redisClient := redis.NewClient(&redis.Options{
                Addr: "localhost:6379",
        })
        storage := newRedisStorage(redisClient)

        limiter := NewTokenBucketLimiter(storage, 10, 2, time.Second*1)
        defer limiter.Close()

        keys := []string{"api_user_1", "api_user_2"}

        var wg sync.WaitGroup
        for _, key := range keys {
```

```
                wg.Add(1)
                go func(k string) {
                        defer wg.Done()
                        for i := 0; i < 15; i++ {
                                err := limiter.Acquire(ctx, k)
                                if err != nil {
                                        fmt.Printf("Failed to acquire token for %s: %v\n",
k, err)
                                        continue
                                }
                                fmt.Printf("Acquired token for %s\n", k)
                                time.Sleep(100 * time.Millisecond)
                                err = limiter.Release(ctx, k)
                                if err != nil {
                                        fmt.Printf("Failed to release token for %s: %v\n", k,
err)
                                }
                        }
                }(key)
        }
        wg.Wait()
}

// --- Testing ---

// Below is a simplified unit test example.
```

```
// In production, place tests in separate _test.go files.

func testTokenBucketLimiter() error {
	ctx := context.Background()

	// Mock storage for unit test - in-memory storage implementation
	memStore := newInMemoryStorage()

	limiter := NewTokenBucketLimiter(memStore, 5, 1, time.Second)
	defer limiter.Close()

	key := "test_user"

	// Preload tokens
	_, err := memStore.Increment(ctx, key, 5, 0)
	if err != nil {
		return err
	}

	// Acquire tokens one by one
	for i := 0; i < 5; i++ {
		ok, err := limiter.TryAcquire(ctx, key)
		if err != nil {
			return fmt.Errorf("try acquire failed: %w", err)
		}
```

```
                if !ok {
                        return fmt.Errorf("expected token available at iteration %d", i)
                }
        }

        // Next acquire should fail
        ok, err := limiter.TryAcquire(ctx, key)
        if err != nil {
                return err
        }
        if ok {
                return errors.New("expected rate limit exceeded")
        }

        return nil
}

// --- In-memory storage implementation for testing ---

type inMemoryStorage struct {
        mu    sync.Mutex
        store map[string]int64
}

func newInMemoryStorage() *inMemoryStorage {
```

```
        return &inMemoryStorage{
                store: make(map[string]int64),
        }
}

func (m *inMemoryStorage) Increment(ctx context.Context, key string, delta int64,
expire time.Duration) (int64, error) {
        m.mu.Lock()
        defer m.mu.Unlock()
        m.store[key] += delta
        return m.store[key], nil
}

func (m *inMemoryStorage) Decrement(ctx context.Context, key string, delta int64)
(int64, error) {
        m.mu.Lock()
        defer m.mu.Unlock()
        val, ok := m.store[key]
        if !ok {
                return 0, ErrRateLimitExceeded
        }
        val -= delta
        if val < 0 {
                m.store[key] = 0
                return 0, ErrRateLimitExceeded
        }
```

```
	m.store[key] = val
	return val, nil
}

func (m *inMemoryStorage) Get(ctx context.Context, key string) (int64, error) {
	m.mu.Lock()
	defer m.mu.Unlock()
	val, ok := m.store[key]
	if !ok {
		return 0, nil
	}
	return val, nil
}
```