

Backpressure Pattern

What is Backpressure?

Backpressure: When a consumer cannot keep up with producer, system pushes back to slow down production.

Why it matters:

- **Prevent overload:** Unchecked producers cause memory exhaustion
- **Maintain latency:** Queue buildup increases response time
- **System stability:** Graceful degradation under load

Without backpressure:

```
Producer: 10000 req/sec  
Consumer: 100 req/sec  
Result: Queue grows unbounded → OOM crash
```

With backpressure:

```
Producer: 10000 req/sec  
Consumer: 100 req/sec  
Result: Producer slowed/blocked to 100 req/sec
```

Backpressure Strategies

1. Blocking (Synchronous)

Concept: Producer blocks when queue full.

```
func producer(queue chan Task) {  
    for {  
        task := generateTask()  
        queue <- task // Blocks if queue full  
    }  
}  
  
func consumer(queue chan Task) {  
    for task := range queue {  
        process(task)  
    }  
}  
  
// Usage:  
queue := make(chan Task, 100) // Bounded buffer  
  
go producer(queue)  
go consumer(queue)
```

Pros: Simple, automatic backpressure

Cons: Blocks producer goroutine

2. Dropping (Lossy)

Concept: Drop new requests when overloaded.

```
func producerWithDrop(queue chan Task) {
    for {
        task := generateTask()

        select {
        case queue <- task:
            // Sent successfully
        default:
            // Queue full, drop task
            log.Printf("Dropped task: %v", task.ID)
            metrics.DroppedTasks.Inc()
        }
    }
}
```

Pros: Non-blocking, maintains latency

Cons: Data loss

When to use: Metrics, logs, non-critical data

3. Buffering (Temporary Storage)

Concept: Use larger buffer to absorb bursts.

```
// Small buffer: frequent backpressure
queue := make(chan Task, 10)

// Large buffer: absorbs bursts
queue := make(chan Task, 10000)
```

Pros: Handles bursts

Cons: Delays backpressure signal, uses memory

4. Shedding (Selective Drop)

Concept: Drop low-priority requests first.

```
type PriorityQueue struct {
    high  chan Task
    medium chan Task
    low   chan Task
}

func (pq *PriorityQueue) Add(task Task) bool {
    switch task.Priority {
```

```

case High:
    select {
    case pq.high <- task:
        return true
    default:
        return false
    }

case Medium:
    select {
    case pq.medium <- task:
        return true
    default:
        // Try to drop low priority to make room
        select {
        case <-pq.low:
            // Dropped low priority
        default:
        }
        return false
    }

case Low:
    select {
    case pq.low <- task:
        return true
    default:
        return false // Drop low priority
    }
}

return false
}

```

5. Rate Limiting (Proactive)

Concept: Limit production rate to match consumption.

```

func rateLimitedProducer(queue chan Task, limit rate.Limit) {
    limiter := rate.NewLimiter(limit, int(limit))

    for {
        // Wait for rate limiter
        limiter.Wait(context.Background())

        task := generateTask()
        queue <- task
    }
}

// Usage: Match consumer capacity

```

```
queue := make(chan Task, 100)
go ratelimitedProducer(queue, 100) // 100 tasks/sec
```

6. Circuit Breaker (Failure Protection)

Concept: Stop sending requests when downstream is failing.

```
type CircuitBreaker struct {
    maxFailures int
    resetTime   time.Duration

    failures    int
    lastAttempt time.Time
    state       string // closed, open, half-open
    mu          sync.Mutex
}

func (cb *CircuitBreaker) Call(fn func() error) error {
    cb.mu.Lock()
    defer cb.mu.Unlock()

    // Check state
    switch cb.state {
    case "open":
        // Check if should transition to half-open
        if time.Since(cb.lastAttempt) > cb.resetTime {
            cb.state = "half-open"
        } else {
            return fmt.Errorf("circuit breaker open")
        }

    case "half-open":
        // Try one request
    }

    // Call function
    err := fn()
    cb.lastAttempt = time.Now()

    if err != nil {
        cb.failures++
        if cb.failures >= cb.maxFailures {
            cb.state = "open"
        }
        return err
    }

    // Success
    cb.failures = 0
    cb.state = "closed"
}
```

```
    return nil
}
```

Real-World Example: HTTP Server with Backpressure

```
type Server struct {
    queue      chan *http.Request
    maxWorkers int
    sem        chan struct{}
}

func NewServer(queueSize, maxWorkers int) *Server {
    s := &Server{
        queue:      make(chan *http.Request, queueSize),
        maxWorkers: maxWorkers,
        sem:        make(chan struct{}, maxWorkers),
    }

    // Start workers
    for i := 0; i < maxWorkers; i++ {
        go s.worker()
    }

    return s
}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    select {
    case s.queue <- r:
        // Queued successfully
        <-s.sem // Wait for worker to finish
        w.WriteHeader(http.StatusOK)

    default:
        // Queue full, apply backpressure
        w.WriteHeader(http.StatusServiceUnavailable)
        w.Write([]byte("Server overloaded, try again later"))
        metrics.RejectedRequests.Inc()
    }
}

func (s *Server) worker() {
    for req := range s.queue {
        s.process(req)
        s.sem <- struct{}{} // Signal completion
    }
}
```

Adaptive Backpressure

Adjust queue size based on latency.

```
type AdaptiveQueue struct {
    queue      chan Task
    targetLatency time.Duration
    minSize     int
    maxSize     int

    currentSize int
    mu          sync.RWMutex
}

func (aq *AdaptiveQueue) adjustSize() {
    ticker := time.NewTicker(10 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        p99 := getP99Latency()

        aq.mu.Lock()

        if p99 > aq.targetLatency*2 {
            // Latency too high, reduce queue
            newSize := int(float64(aq.currentSize) * 0.8)
            if newSize < aq.minSize {
                newSize = aq.minSize
            }
            aq.resize(newSize)

        } else if p99 < aq.targetLatency/2 {
            // Latency low, increase queue
            newSize := int(float64(aq.currentSize) * 1.2)
            if newSize > aq.maxSize {
                newSize = aq.maxSize
            }
            aq.resize(newSize)
        }

        aq.mu.Unlock()
    }
}

func (aq *AdaptiveQueue) resize(newSize int) {
    oldQueue := aq.queue
    aq.queue = make(chan Task, newSize)
    aq.currentSize = newSize

    // Drain old queue into new
    go func() {
        for task := range oldQueue {
            aq.queue <- task
        }
    }
}
```

```

    }
  }()
}

```

Backpressure with Context

```

func processWithBackpressure(ctx context.Context, tasks <-chan Task, workers int)
error {
    sem := make(chan struct{}, workers)
    errCh := make(chan error, 1)

    for task := range tasks {
        select {
        case <-ctx.Done():
            return ctx.Err()

        case sem <- struct{}{}:
            go func(t Task) {
                defer func() { <-sem }()

                if err := process(ctx, t); err != nil {
                    select {
                    case errCh <- err:
                    default:
                    }
                }
            }(task)

        case err := <-errCh:
            return err
        }
    }

    // Wait for all workers
    for i := 0; i < cap(sem); i++ {
        sem <- struct{}{}
    }

    return nil
}

```

Monitoring Backpressure

```

type QueueMetrics struct {
    size      int
    enqueued  prometheus.Counter
    dequeued  prometheus.Counter
    dropped   prometheus.Counter
}

```

```

    queueTime prometheus.Histogram
}

func (qm *QueueMetrics) Add(task Task) bool {
    if len(queue) == cap(queue) {
        qm.dropped.Inc()
        return false
    }

    task.EnqueueTime = time.Now()
    queue <- task
    qm.enqueued.Inc()
    return true
}

func (qm *QueueMetrics) Get() Task {
    task := <-queue
    qm.dequeued.Inc()

    queueTime := time.Since(task.EnqueueTime)
    qm.queueTime.Observe(queueTime.Seconds())

    return task
}

// Alerts:
// - Queue depth > 80% capacity
// - Drop rate > 1%
// - P99 queue time > 5 seconds

```

Comparing Backpressure Strategies

Strategy	Pros	Cons	Use Case
Blocking	Simple, no loss	Blocks producer	Reliable processing
Dropping	Non-blocking	Data loss	Metrics, logs
Buffering	Absorbs bursts	Memory, delays signal	Temporary spikes
Shedding	Smart dropping	Complex	Prioritized workloads
Rate Limiting	Proactive	Needs tuning	Known capacity
Circuit Breaker	Fail fast	Can miss recovery	Cascading failures

Common Mistakes

Mistake 1: Unbounded Queue

```
// WRONG: Unbounded queue
queue := make(chan Task) // No buffer

go func() {
    for {
        queue <- generateTask() // Producer never blocks
    }
}()

go func() {
    for task := range queue {
        time.Sleep(time.Second) // Slow consumer
        process(task)
    }
}()

// Result: Channel buffer grows in memory → OOM
```

Fix:

```
queue := make(chan Task, 1000) // Bounded

go func() {
    for {
        task := generateTask()
        select {
        case queue <- task:
        default:
            // Drop or log
        }
    }
}()
```

Mistake 2: Ignoring Backpressure Signal

```
// WRONG: Ignoring `default` case
select {
case queue <- task:
default:
    // Empty! Silently drops task
}

// Better: Log, metric, return error
select {
case queue <- task:
    return nil
default:
    metrics.DroppedTasks.Inc()
```

```
    return fmt.Errorf("queue full")
}
```

Mistake 3: No Monitoring

```
// WRONG: No visibility into queue state
queue := make(chan Task, 1000)

// Better: Expose metrics
func queueDepth() int {
    return len(queue)
}

func queueCapacity() int {
    return cap(queue)
}

// Prometheus metric
queueDepthGauge.Set(float64(len(queue)))
```

Load Shedding Patterns

Random Drop

```
func randomDrop(queue chan Task, dropRate float64) bool {
    if rand.Float64() < dropRate {
        return false // Drop
    }

    select {
    case queue <- task:
        return true
    default:
        return false
    }
}
```

Probabilistic Drop (Based on Queue Depth)

```
func adaptiveDrop(queue chan Task, task Task) bool {
    depth := float64(len(queue)) / float64(cap(queue))

    // Drop probability increases with depth
    dropProb := depth * depth // 0.5 depth → 0.25 drop, 0.9 depth → 0.81 drop

    if rand.Float64() < dropProb {
        return false
    }
}
```

```

select {
case queue <- task:
    return true
default:
    return false
}
}

```

Token Bucket Backpressure

```

type TokenBucketBackpressure struct {
    tokens    float64
    capacity  float64
    rate      float64
    lastFill  time.Time
    mu        sync.Mutex
}

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

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

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

    return false
}

func producerWithTokenBucket(queue chan Task, tb *TokenBucketBackpressure) {
    for {
        task := generateTask()

        if tb.Allow() {
            queue <- task
        } else {
            // Backpressure: don't generate more tasks
            time.Sleep(time.Millisecond)
        }
    }
}

```

```
}  
}  
}
```

Production Failure: AWS S3 2017

Background: S3 experienced 4-hour outage in us-east-1.

Root cause: Billing system initiated large-scale subsystem restart. Request backlog built up during restart.

Backpressure failure:

1. Request queue unbounded
2. Queue grew to many GB
3. Memory exhaustion crashed servers
4. Cascading failures across subsystems

Fix:

- Added bounded queues with backpressure
- Circuit breakers between subsystems
- Load shedding for non-critical requests
- Faster subsystem restarts

Interview Questions

Q: "What's the difference between backpressure and rate limiting?"

"Rate limiting: Proactive—control input rate before queuing. Backpressure: Reactive—signal to slow down when queue full. Rate limiting prevents problem, backpressure handles problem. Often used together: rate limit + bounded queue. Example: API rate limits (prevent), buffered channel (backpressure when buffer full)."

Q: "When should you drop requests vs. block producer?"

"Drop (non-blocking) when: lossy data acceptable (metrics, logs), maintaining latency critical, producer can retry. Block when: must not lose data (payments, orders), producer needs feedback, downstream recovery expected soon. Hybrid: block with timeout (wait 1s, then drop). Trade-off: dropping loses data but maintains latency; blocking preserves data but adds latency."

Q: "How do you implement adaptive backpressure?"

"Monitor queue depth and latency. If P99 latency > target, reduce queue size to apply backpressure sooner. If latency < target and dropping requests, increase queue size. Adjust every 10-60s to avoid oscillation. Metrics: queue depth, enqueue rate, drop rate, processing time. Implementation: resizing buffered channel by creating new channel + draining old."

Q: "How does backpressure work in distributed systems?"

"Use explicit signals across network. HTTP: Return 503 Service Unavailable + Retry-After header. gRPC: RESOURCE_EXHAUSTED status. Message queues: Nack messages, consumer rate limit. Load balancers: Remove unhealthy backends. Challenges: Propagation delay, coordinating backpressure across services. Pattern: Sidecar monitors local queue, updates load balancer health."

Key Takeaways

1. **Backpressure prevents overload by slowing producers**
2. **Bounded channels provide automatic backpressure**
3. **Drop low-priority data when overloaded**
4. **Monitor queue depth and latency**
5. **Circuit breakers prevent cascading failures**
6. **Adaptive backpressure adjusts to load**
7. **Rate limiting is proactive, backpressure is reactive**
8. **Always handle the default case in select**
9. **Unbounded queues lead to OOM**
10. **Production failures show importance of proper buffering**

Exercises

1. Implement queue with metrics: depth, enqueue rate, drop rate. Add alerts.
2. Build adaptive queue that resizes based on P99 latency.
3. Create HTTP server that returns 503 when queue > 80% full.
4. Implement priority-based load shedding (drop low priority first).
5. Benchmark different strategies: blocking, dropping, rate limiting under load.

Section 04 Complete! Next: [../05-real-world-go/http-servers.md](http://05-real-world-go/http-servers.md) - Applying patterns to production HTTP servers.