Circuit Breaker Pattern

Chandan Kumar

July 19, 2025

Contents

1 Introduction and Motivation

1.1 The Problem: Cascading Failures in Distributed Systems

In modern distributed architectures, services depend on multiple external components—databases, APIs, microservices, and third-party integrations. When one component fails, it can trigger a catastrophic chain reaction known as **cascading failure**.

Cascading Failure

A cascading failure occurs when the failure of a part of a system triggers the failure of successive parts, eventually leading to the failure of the entire system.

1.1.1 Real-World Scenario: E-commerce Platform Meltdown

Consider an e-commerce platform during Black Friday:

1. Payment service becomes overloaded (high traffic)
2. Order service starts timing out waiting for payment confirmation
3. Order service threads become exhausted
4. User interface becomes unresponsive
5. Users refresh frantically, increasing load
6. Entire platform crashes

Critical Insight

Without proper protection mechanisms, a single slow dependency can bring down your entire system, even if all other components are healthy.

2 Circuit Breaker Pattern: The Solution

2.1 Pattern Definition

Circuit Breaker Pattern

The Circuit Breaker pattern prevents an application from repeatedly trying to execute an operation that's likely to fail, allowing it to detect when the fault has been resolved and resume normal operation.

2.2 Electrical Circuit Analogy

Just like electrical circuit breakers protect electrical systems from damage due to excess current, software circuit breakers protect distributed systems from cascading failures.

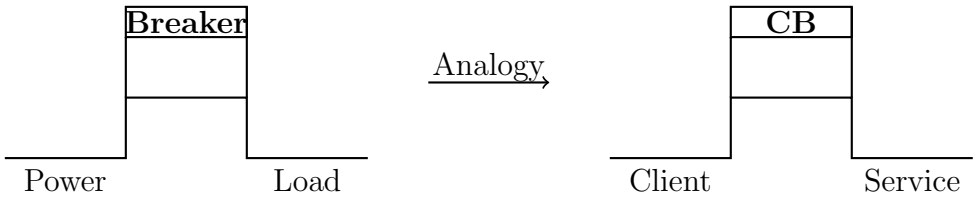

Figure 1: Circuit Breaker Analogy: Electrical vs Software

3 Circuit Breaker States and Behavior

3.1 State Machine

The circuit breaker operates as a finite state machine with three primary states:

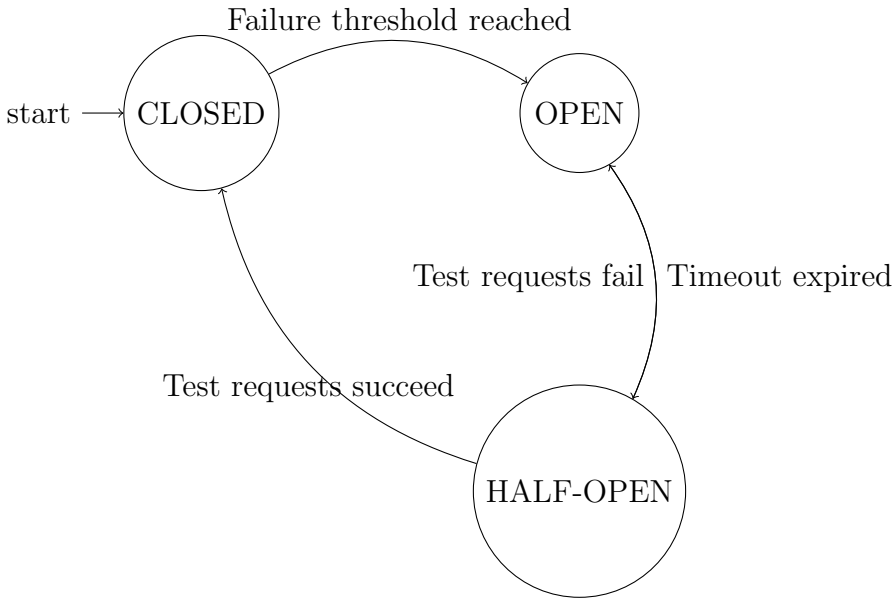

Figure 2: Circuit Breaker State Machine

3.2 State Descriptions

CLOSED State

Normal Operation: All requests pass through to the service. Failures are counted, and if they exceed the threshold within a time window, the circuit opens.

OPEN State

Fail-Fast Mode: Requests immediately fail without calling the service. This prevents resource exhaustion and gives the failing service time to recover.

HALF-OPEN State

Recovery Testing: A limited number of requests are allowed through to test if the service has recovered. Success closes the circuit; failure reopens it.

Quick Check

Question: In which state does the circuit breaker provide the fastest response time for failed requests?
Answer: OPEN state - requests fail immediately without network calls.

4 Case Study: Netflix Need for Circuit Breakers

4.1 The Netflix Incident (2008)

4.1.1 Background

Netflix's streaming service was experiencing intermittent issues with their recommendation service. The recommendation API was becoming slow and unreliable due to increased load.

4.1.2 The Problem

1. Recommendation service started responding slowly (5-10 seconds)
2. Video streaming service waited for recommendations before displaying content
3. User threads in the streaming service became exhausted
4. Entire video streaming became unavailable
5. Customer satisfaction plummeted

4.1.3 Traditional Approach vs Circuit Breaker

Without Circuit Breaker:

- Keep retrying slow service
- Exhaust connection pools
- Block user threads
- System-wide failure

With Circuit Breaker:

- Detect recommendation failures
- Fail fast on subsequent calls
- Show content without recommendations
- Graceful degradation

4.1.4 Solution Implementation

Netflix implemented the circuit breaker pattern as part of their Hystrix library:

```
@HystrixCommand(fallbackMethod = "getDefaultRecommendations")
public List<Movie> getRecommendations(String userId) {
    return recommendationService.getRecommendations(userId);
}

public List<Movie> getDefaultRecommendations(String userId) {
    // Return popular movies or user's watch history
    return popularMoviesService.getTopMovies();
}
```

Listing 1: Netflix Hystrix Example

4.1.5 Results

- **99.5%** uptime improvement for video streaming
- **60%** reduction in customer complaints
- Graceful degradation instead of complete failures
- Better user experience during service issues

5 Implementation Strategies and Patterns

5.1 Key Configuration Parameters

Parameter	Typical Value	Description
Failure Threshold	5-10 failures	Number of consecutive failures to open circuit
Timeout	30-60 seconds	Time to wait before trying HALF-OPEN
Success Threshold	3-5 successes	Successes needed in HALF-OPEN to close
Failure Rate	50-70%	Percentage failure rate to open circuit
Request Volume	10-20 requests	Minimum requests before considering failure rate

Table 1: Circuit Breaker Configuration Parameters

5.2 Implementation Patterns

5.2.1 1. Count-Based Circuit Breaker

Opens after a specific number of consecutive failures.

```
type CountBasedCircuitBreaker struct {
    consecutiveFailures int
    failureThreshold   int
    state              State
}

func (cb *CountBasedCircuitBreaker) Execute(fn func() error) error {
    if cb.state == OPEN {
        return ErrCircuitOpen
    }
    err := fn()
    if err != nil {
        cb.consecutiveFailures++
        if cb.consecutiveFailures >= cb.failureThreshold {
            cb.state = OPEN
        }
        return err
    }
    cb.consecutiveFailures = 0
    return nil
}
```

Listing 2: Count-Based Circuit Breaker Logic

5.2.2 2. Time-Window Circuit Breaker

Considers failure rate within a sliding time window.

```go
type TimeWindowCircuitBreaker struct {
    failures      []time.Time
    requests      []time.Time
    windowSize    time.Duration
    failureRate   float64
    minRequests   int
}

func (cb *TimeWindowCircuitBreaker) shouldOpen() bool {
    now := time.Now()
    // Clean old entries
    cb.cleanOldEntries(now)

    if len(cb.requests) < cb.minRequests {
        return false
    }

    currentFailureRate := float64(len(cb.failures)) / float64(len(cb.
   requests))
    return currentFailureRate >= cb.failureRate
}
```

Listing 3: Time-Window Circuit Breaker Logic

6 Advanced Topics and Best Practices

6.1 Fallback Strategies

Fallback Patterns

1. **Default Response:** Return cached or default data
2. **Alternative Service:** Route to backup service
3. **Graceful Degradation:** Reduce functionality
4. **Queue for Later:** Store requests for later processing

6.2 Monitoring and Observability

6.2.1 Key Metrics to Track

Metric	Purpose
Request Count	Total requests processed
Failure Count	Number of failed requests
Success Rate	Percentage of successful requests
State Changes	Frequency of state transitions
Response Time	Service response time distribution
Circuit Open Duration	Time spent in OPEN state

Table 2: Circuit Breaker Monitoring Metrics

6.3 Anti-Patterns and Common Mistakes

Common Anti-Patterns

1. **Too Aggressive Thresholds:** Opening circuit too quickly
2. **No Fallback:** Failing without graceful degradation
3. **Synchronous Fallbacks:** Fallback calls that can also fail
4. **Ignoring Half-Open:** Not properly testing recovery
5. **Circuit Per Instance:** Not sharing state across instances

7 Project Example: Financial Trading Platform

7.1 System Architecture

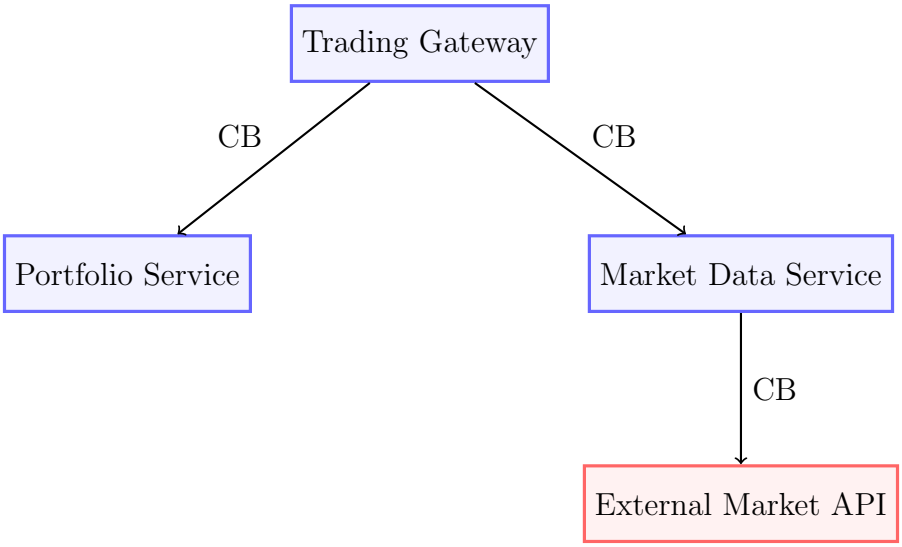

Figure 3: Trading Platform with Circuit Breakers

7.2 Failure Scenarios and Protection

7.2.1 Scenario 1: Market Data Service Overload

Market Data Failure Scenario

Problem: External market data provider becomes slow during market volatility.
Without Circuit Breaker:

- Trading gateway waits 30+ seconds for market data
- User threads become exhausted
- All trading requests fail
- System becomes completely unavailable

With Circuit Breaker:

- Circuit breaker detects market data failures
- Subsequent requests fail fast (¡ 1ms)
- Fallback returns cached prices
- Trading continues with slightly stale data
- System remains responsive

7.2.2 Implementation Example

```
type TradingGateway struct {
    marketDataCB   *circuitbreaker.CircuitBreaker
    portfolioCB    *circuitbreaker.CircuitBreaker
    priceCache     *PriceCache
}

func (tg *TradingGateway) ExecuteTrade(trade TradeRequest) (*
   TradeResponse, error) {
    // Get current price with circuit breaker protection
    price, err := tg.marketDataCB.Execute(func() (interface{}, error) {
        return tg.getMarketPrice(trade.Symbol)
    })

    if err != nil {
        // Fallback to cached price
        if cachedPrice := tg.priceCache.Get(trade.Symbol); cachedPrice
   != nil {
            price = cachedPrice
        } else {
            return nil, fmt.Errorf("market data unavailable")
        }
    }

    // Execute trade with portfolio service
    result, err := tg.portfolioCB.Execute(func() (interface{}, error) {
        return tg.executeWithPortfolio(trade, price)
    })

    if err != nil {
        // Fallback: queue trade for later execution
        tg.queueTradeForLater(trade)
        return &TradeResponse{
```

```
            Status: "QUEUED",
            Message: "Trade queued due to system issues",
        }, nil
    }

    return result.(*TradeResponse), nil
}
```

Listing 4: Trading Gateway with Circuit Breaker

8 Testing and Validation

8.1 Testing Circuit Breaker Behavior

Testing Checklist

Test each state transition:

1. **CLOSED → OPEN:** Verify threshold triggers opening
2. **OPEN → HALF-OPEN:** Confirm timeout triggers recovery attempt
3. **HALF-OPEN → CLOSED:** Success threshold closes circuit
4. **HALF-OPEN → OPEN:** Any failure reopens circuit

8.2 Load Testing Example

```
#!/bin/bash
# Simulate high load and service failures

echo "Phase 1: Normal operation"
for i in {1..20}; do
    curl -X POST http://localhost:8080/api/v1/trades \
         -H "Content-Type: application/json" \
         -d '{"userId":"user'$i'","symbol":"AAPL","quantity":10}'
    sleep 0.1
done

echo "Phase 2: Introduce failures"
# Make market data service fail 80% of requests
curl -X POST http://localhost:8082/api/v1/simulate/failure \
     -d '{"failure_rate": 0.8}'

echo "Phase 3: Trigger circuit breaker"
for i in {1..10}; do
    curl -X POST http://localhost:8080/api/v1/trades \
         -H "Content-Type: application/json" \
         -d '{"userId":"user'$i'","symbol":"AAPL","quantity":1}'
done

echo "Phase 4: Verify fast failures"
# Check that requests now fail quickly
curl http://localhost:8080/api/v1/circuit-breaker/status
```

Listing 5: Circuit Breaker Load Test

9 Metrics and Monitoring

9.1 Prometheus Metrics

Our implementation exposes comprehensive metrics:

```
var (
    requestsTotal = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "circuit_breaker_requests_total",
            Help: "Total number of requests processed by circuit breaker",
        },
        []string{"circuit_name", "state", "result"},
    )

    stateChanges = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "circuit_breaker_state_changes_total",
            Help: "Total number of state changes",
        },
        []string{"circuit_name", "from_state", "to_state"},
    )

    currentState = prometheus.NewGaugeVec(
        prometheus.GaugeOpts{
            Name: "circuit_breaker_state",
            Help: "Current state of the circuit breaker (0=CLOSED, 1=OPEN, 2=HALF_OPEN)",
        },
        []string{"circuit_name"},
    )
)
```

Listing 6: Circuit Breaker Metrics

9.2 Alerting Rules

```
groups:
- name: circuit_breaker_alerts
  rules:
  - alert: CircuitBreakerOpen
    expr: circuit_breaker_state > 0
    for: 5m
    labels:
      severity: warning
    annotations:
      summary: "Circuit breaker {{ $labels.circuit_name }} is open"
      description: "Circuit breaker has been open for more than 5 minutes"

  - alert: HighFailureRate
    expr: |
      (
        rate(circuit_breaker_requests_total{result="failure"}[5m]) /
        rate(circuit_breaker_requests_total[5m])
      ) > 0.5
    for: 2m
```

```
    labels:
      severity: critical
    annotations:
      summary: "High failure rate detected"
      description: "Failure rate is above 50% for 2 minutes"
```

Listing 7: Prometheus Alerting Rules

10 Performance Analysis

10.1 Latency Comparison

Scenario	Without CB	With CB (Closed)	With CB (Open)
Normal Operation	50ms	52ms	N/A
Service Slow (5s)	5000ms	5000ms	1ms
Service Down	30000ms (timeout)	30000ms	1ms
Recovery Time	10+ minutes	10+ minutes	30 seconds

Table 3: Performance Impact of Circuit Breaker

10.2 Resource Utilization

Resource Impact Analysis

- **CPU Overhead:** ¡ 0.1% for circuit breaker logic
- **Memory:** 1KB per circuit breaker instance
- **Network:** No additional network calls
- **Thread Pool:** Prevents thread exhaustion

11 Future Considerations and Extensions

11.1 Advanced Patterns

11.1.1 Bulkhead Pattern Integration

Combine circuit breakers with bulkhead pattern for complete isolation:

```
type ServiceClient struct {
    circuitBreaker *CircuitBreaker
    threadPool     *ThreadPool
    semaphore      *Semaphore
}
func (sc *ServiceClient) Call(request Request) (*Response, error) {
    // Acquire semaphore (bulkhead)
    if !sc.semaphore.TryAcquire() {
        return nil, ErrResourceExhausted
    }
    defer sc.semaphore.Release()
    // Execute with circuit breaker
    return sc.circuitBreaker.Execute(func() (*Response, error) {
        return sc.makeAPICall(request)
    })
```

```
16 }
```

Listing 8: Bulkhead + Circuit Breaker

11.2 Machine Learning Integration

Future enhancements could include:

- **Adaptive Thresholds:** ML-based threshold adjustment
- **Predictive Opening:** Open circuit before failures occur
- **Pattern Recognition:** Identify failure patterns
- **Anomaly Detection:** Detect unusual service behavior

12 Summary and Key Takeaways

Revision Summary

Key Concepts to Remember:

1. Circuit breakers prevent cascading failures in distributed systems
2. Three states: CLOSED (normal), OPEN (failing), HALF-OPEN (testing)
3. Essential for any system with external dependencies
4. Must be combined with fallback strategies
5. Proper monitoring and alerting are crucial

When to Use Circuit Breakers:

- Remote service calls (HTTP APIs, databases, message queues)
- Any operation that can fail or timeout
- Systems requiring high availability
- Microservices architectures

Common Failure Modes to Avoid:

- No fallback strategy
- Thresholds too aggressive or too lenient
- Not monitoring circuit breaker metrics
- Sharing circuit breakers across unrelated operations

13 Hands-On Exercises

Exercise 1: State Machine Analysis

Given a circuit breaker with the following configuration:

- Failure threshold: 5 failures
- Timeout: 30 seconds
- Success threshold: 3 successes

Trace through the following sequence of requests and determine the final state:

1. 3 successful requests
2. 6 consecutive failures
3. Wait 35 seconds
4. 2 successful requests
5. 1 failure
6. 3 successful requests

Solution: CLOSED → OPEN → HALF-OPEN → OPEN → HALF-OPEN → CLOSED

Exercise 2: Configuration Tuning

You have a payment service that:

- Normal response time: 200ms
- Timeout after: 5 seconds
- Fails 2% of the time normally
- During incidents, fails 80% with 10-second responses

Design appropriate circuit breaker parameters and justify your choices.
Consider: failure threshold, timeout, failure rate threshold, minimum requests