RFC-028: Error Handling and Recovery

Status: Draft **Date:** January 2026 **Author:** Derrell Piper ddp@eludom.net
Implementation: Proposed

Abstract

This RFC specifies error handling and recovery for the Library of Cyberspace: how vaults detect, report, and recover from failures while maintaining integrity guarantees. Errors are first-class objects in the soup, enabling systematic analysis and automated recovery.

Motivation

Distributed systems fail in creative ways:

- **Network partitions** - Vaults lose contact
- **Storage corruption** - Bits flip, disks fail
- **Byzantine faults** - Nodes lie or misbehave
- **Resource exhaustion** - Out of memory, disk, connections
- **Protocol violations** - Malformed messages, invalid state

Recovery must be:

- **Deterministic** - Same error, same recovery
- **Auditable** - Every error logged with context
- **Automatic** - Self-healing where possible
- **Graceful** - Degrade rather than crash

Error Model

Error Types

```
(define-error-type 'storage-error
  (subtypes corruption missing full readonly))

(define-error-type 'network-error
  (subtypes timeout unreachable protocol-violation))

(define-error-type 'crypto-error
  (subtypes signature-invalid key-expired capability-denied))

(define-error-type 'consistency-error
  (subtypes hash-mismatch merkle-divergence conflict))
```

```
(define-error-type 'resource-error
  (subtypes memory-exhausted connections-exhausted rate-limited))

(define-error-type 'protocol-error
  (subtypes invalid-message invalid-state version-mismatch))
```

Error Structure

```
(define (make-error type message context)
  `(error
    (id ,(generate-error-id))
    (type ,type)
    (message ,message)
    (context ,context)
    (timestamp ,(current-time))
    (vault ,(current-vault-id))
    (principal ,(current-principal))
    (stack ,(capture-stack-trace))))

;; Errors are soup objects
(define (record-error! err)
  (let ((hash (soup-put err type: 'error)))
    (audit-append action: 'error-recorded error-hash: hash)
    hash))
```

Error Context

```
;; Capture rich context for debugging
(define (error-context operation)
  `((operation . ,operation)
    (request-id . ,(current-request-id))
    (correlation-id . ,(current-correlation-id))
    (peer . ,(current-peer))
    (attempt . ,(current-retry-count))
    (duration . ,(operation-duration))
    (inputs . ,(sanitize-inputs (operation-inputs)))))
```

Error Handling Patterns

Result Type

```
;; Operations return (ok value) or (err error)
(define (result-ok value) `(ok ,value))
(define (result-err error) `(err ,error))
```

```
(define (result-ok? r) (eq? (car r) 'ok))
(define (result-err? r) (eq? (car r) 'err))
(define (result-value r) (cadr r))
(define (result-error r) (cadr r))

;; Chain operations
(define (result-bind r f)
  (if (result-ok? r)
      (f (result-value r))
      r))

;; Example
(define (safe-read hash)
  (result-bind (cas-get-safe hash)
    (lambda (data)
      (result-bind (verify-signature data)
        (lambda (verified)
          (result-ok verified))))))
```

Error Recovery

```
(define (with-recovery operation recovery)
  "Execute operation with recovery on failure"
  (let ((result (guard (ex (else (result-err ex)))
                  (result-ok (operation)))))
    (if (result-ok? result)
        (result-value result)
        (recovery (result-error result)))))

;; Example: read with fallback to replica
(define (read-with-fallback hash)
  (with-recovery
    (lambda () (cas-get hash))
    (lambda (err)
      (audit-append action: 'read-fallback error: err)
      (cas-get-from-replica hash))))
```

Retry Logic

```
(define (with-retry operation max-retries backoff)
  "Retry operation with exponential backoff"
  (let loop ((attempt 1))
    (let ((result (guard (ex (else (result-err ex)))
                    (result-ok (operation)))))
      (cond
```

```
        ((result-ok? result)
         (result-value result))
        ((>= attempt max-retries)
         (error "Max retries exceeded" (result-error result)))
        (else
         (let ((delay (* backoff (expt 2 (- attempt 1)))))
           (audit-append action: 'retry attempt: attempt delay: delay)
           (thread-sleep! delay)
           (loop (+ attempt 1)))))))

;; Retry configuration
(define *default-retry-config*
  `((max-retries . 3)
    (initial-backoff . 100)    ; ms
    (max-backoff . 30000)      ; ms
    (jitter . 0.1)))           ; 10% randomization
```

Circuit Breaker

```
(define (make-circuit-breaker threshold timeout)
  (let ((failures 0)
        (state 'closed)
        (last-failure #f))

    (lambda (operation)
      (case state
        ((open)
         (if (> (- (current-time) last-failure) timeout)
             (begin
               (set! state 'half-open)
               (try-operation operation))
             (error "Circuit open")))
        ((half-open)
         (let ((result (try-operation operation)))
           (if (result-ok? result)
               (begin
                 (set! state 'closed)
                 (set! failures 0)
                 (result-value result))
               (begin
                 (set! state 'open)
                 (error "Circuit reopened")))))
        ((closed)
         (try-operation operation))))))

(define (try-operation cb operation)
```

```
  (guard (ex
          (else
           (cb-record-failure! cb)
           (result-err ex)))
    (result-ok (operation))))
```

Storage Errors

Corruption Detection

```
(define (detect-corruption hash)
  "Verify object integrity"
  (let* ((data (storage-read-raw hash))
         (computed-hash (content-hash data)))
    (unless (equal? hash computed-hash)
      (record-error!
        (make-error 'corruption
          "Hash mismatch detected"
          `((expected . ,hash)
            (computed . ,computed-hash)
            (size . ,(bytevector-length data))))))))

(define (scan-for-corruption)
  "Full vault integrity scan"
  (let ((corrupted '()))
    (for-each (lambda (hash)
                (unless (verify-integrity hash)
                  (set! corrupted (cons hash corrupted))))
              (all-object-hashes))
    (when (not (null? corrupted))
      (audit-append action: 'corruption-scan
                    corrupted: (length corrupted)))
    corrupted))
```

Corruption Recovery

```
(define (recover-corrupted hash)
  "Attempt to recover corrupted object"
  ;; Strategy 1: Fetch from replica
  (let ((replicas (find-replicas hash)))
    (for-each (lambda (replica)
                (let ((data (fetch-from-replica replica hash)))
                  (when (and data (verify-hash data hash))
                    (storage-write hash data)
                    (audit-append action: 'corruption-recovered
```

```
                                    source: replica)
                        (return data))))
                replicas))

    ;; Strategy 2: Reconstruct from erasure coding
    (when (erasure-coded? hash)
      (let ((data (reconstruct-from-shards hash)))
        (when data
          (storage-write hash data)
          (audit-append action: 'erasure-recovery)
          (return data))))

    ;; Strategy 3: Mark as lost
    (mark-object-lost hash)
    (audit-append action: 'object-lost hash: hash)
    #f)
```

Disk Full Handling

```
(define (handle-disk-full)
  "Emergency disk space recovery"
  ;; Phase 1: Emergency GC
  (gc-emergency)

  ;; Phase 2: Clear caches
  (clear-all-caches)

  ;; Phase 3: Compress compressible objects
  (compress-uncompressed-objects)

  ;; Phase 4: Alert and degrade
  (when (< (available-space) *critical-threshold*)
    (alert-disk-critical)
    (vault-readonly!)))
```

Network Errors

Timeout Handling

```
(define (with-timeout duration operation)
  "Execute operation with timeout"
  (let ((result (make-channel)))
    (thread-start!
      (make-thread
        (lambda ()
```

```
          (channel-put! result
            (guard (ex (else (result-err ex)))
              (result-ok (operation)))))))
    (let ((r (channel-get! result duration)))
      (if r
          (if (result-ok? r)
              (result-value r)
              (error (result-error r)))
          (error (make-error 'timeout
                   "Operation timed out"
                   `((duration . ,duration))))))))
```

Partition Detection

```
(define (detect-partition peers)
  "Detect network partition"
  (let* ((reachable (filter peer-reachable? peers))
         (unreachable (filter (lambda (p) (not (peer-reachable? p))) peers)))
    (when (> (length unreachable) 0)
      (audit-append action: 'partition-detected
                    reachable: (length reachable)
                    unreachable: (length unreachable))
      (if (> (length reachable) (/ (length peers) 2))
          (continue-with-majority reachable)
          (enter-read-only-mode)))))
```

Connection Pool Recovery

```
(define (recover-connection-pool pool)
  "Recover from connection pool exhaustion"
  ;; Close idle connections
  (pool-close-idle! pool)

  ;; Close long-running connections
  (pool-close-stale! pool *max-connection-age*)

  ;; If still exhausted, reject new requests
  (when (pool-exhausted? pool)
    (audit-append action: 'pool-exhausted)
    (enable-request-shedding)))
```

Consistency Errors

Conflict Detection

```
(define (detect-conflict hash versions)
  "Detect conflicting versions of object"
  (let ((unique-contents (delete-duplicates
                          (map version-content versions))))
    (when (> (length unique-contents) 1)
      (record-error!
        (make-error 'conflict
          "Conflicting versions detected"
          `((hash . ,hash)
            (versions . ,(length unique-contents))
            (sources . ,(map version-source versions)))))))))
```

Conflict Resolution

```
(define (resolve-conflict hash versions strategy)
  "Resolve conflicting versions"
  (let ((resolved
         (case strategy
           ((latest)
            (max-by version-timestamp versions))
           ((merge)
            (merge-versions versions))
           ((manual)
            (queue-for-manual-resolution hash versions))
           ((tombstone)
            (create-conflict-tombstone hash versions)))))
    (audit-append action: 'conflict-resolved
                  hash: hash
                  strategy: strategy)
    resolved))
```

Merkle Divergence

```
(define (handle-merkle-divergence local-root remote-root peer)
  "Handle diverged Merkle trees"
  (let ((divergence (find-divergence-point local-root remote-root)))
    (audit-append action: 'merkle-divergence
                  local: local-root
                  remote: remote-root
                  divergence: divergence)
    ;; Sync diverged subtrees
    (for-each (lambda (subtree)
                (sync-subtree subtree peer))
```

```
            (divergent-subtrees divergence))))
```

Recovery Procedures

Automatic Recovery

```
(define recovery-handlers
  `((corruption . ,recover-corrupted)
    (missing . ,fetch-from-replica)
    (timeout . ,retry-with-backoff)
    (partition . ,wait-for-reconnection)
    (conflict . ,resolve-with-default-strategy)))

(define (auto-recover error)
  "Attempt automatic recovery"
  (let ((handler (assoc-ref recovery-handlers (error-type error))))
    (if handler
        (begin
          (audit-append action: 'auto-recovery-attempt
                        error: (error-id error))
          (handler error))
        (escalate-error error))))
```

Manual Recovery

```
(define (queue-manual-recovery error)
  "Queue error for manual intervention"
  (soup-put
    `(recovery-ticket
      (error ,(error-id error))
      (status pending)
      (created ,(current-time))
      (assigned #f))
    type: 'recovery-ticket))

(define (process-recovery-ticket ticket-hash)
  "Process manual recovery ticket"
  (let ((ticket (soup-get ticket-hash)))
    ;; Mark in progress
    (soup-update ticket-hash status: 'in-progress)
    ;; ... manual recovery steps ...
    ;; Mark resolved
    (soup-update ticket-hash
                 status: 'resolved
                 resolved-at: (current-time)
```

```
                resolved-by: (current-principal))))
```

Disaster Recovery

```
(define (disaster-recovery backup-location)
  "Full vault recovery from backup"
  (audit-append action: 'disaster-recovery-start
                backup: backup-location)

  ;; Phase 1: Verify backup integrity
  (unless (verify-backup-integrity backup-location)
    (error "Backup integrity check failed"))

  ;; Phase 2: Restore CAS
  (restore-cas-from-backup backup-location)

  ;; Phase 3: Restore soup
  (restore-soup-from-backup backup-location)

  ;; Phase 4: Restore indexes
  (rebuild-indexes)

  ;; Phase 5: Verify restoration
  (verify-restoration)

  (audit-append action: 'disaster-recovery-complete))
```

Error Reporting

Error Aggregation

```
(define (aggregate-errors window)
  "Aggregate errors over time window"
  (let ((errors (soup-query type: 'error
                            timestamp: (> (- (current-time) window)))))
    (group-by error-type errors)))

(define (error-rate error-type window)
  "Calculate error rate"
  (let ((count (length (soup-query type: 'error
                                   error-type: error-type
                                   timestamp: (> (- (current-time) window))))))
    (/ count window)))
```

Alerting

```
(define alert-thresholds
  `((corruption . 1)        ; Any corruption is critical
    (timeout . 100)         ; 100 timeouts per minute
    (conflict . 10)         ; 10 conflicts per minute
    (resource . 5)))        ; 5 resource errors per minute

(define (check-alert-thresholds)
  "Check if error rates exceed thresholds"
  (for-each (lambda (threshold)
              (let ((type (car threshold))
                    (limit (cdr threshold)))
                (when (> (error-rate type 60) limit)
                  (send-alert type (error-rate type 60) limit))))
            alert-thresholds))
```

Security Considerations

Error Information Leakage

```
;; Don't expose internal details in user-facing errors
(define (sanitize-error-for-user error)
  `(error
    (type ,(error-type error))
    (message ,(user-safe-message (error-type error)))
    (request-id ,(error-request-id error))))

;; Full details only in audit log
(define (log-full-error error)
  (audit-append action: 'error
                error: error))  ; Full context preserved
```

Error-Based Attacks

```
;; Rate limit error generation to prevent DoS
(define error-rate-limiter
  (make-rate-limiter 1000 60))  ; 1000 errors per minute max

(define (rate-limited-record-error! err)
  (if (rate-limiter-allow? error-rate-limiter)
      (record-error! err)
      (audit-append action: 'error-rate-limited)))
```

References

1. Release It! - Nygard
2. Hystrix - Netflix Circuit Breaker
3. RFC-003: Cryptographic Audit Trail
4. RFC-026: Garbage Collection

Changelog

- **2026-01-07** - Initial draft

Implementation Status: Draft **Dependencies:** soup, audit, storage **Integration:** All vault operations, monitoring, alerting