RFC-031: Monitoring and Observability

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

Abstract

This RFC specifies monitoring and observability for the Library of Cyberspace: how vaults expose metrics, traces, and logs for operational visibility while maintaining privacy and security. Observability data is itself content-addressed and auditable.

Motivation

Operating distributed systems requires visibility:

- **Health** - Is the vault functioning correctly?
- **Performance** - How fast are operations?
- **Capacity** - How much storage remains?
- **Errors** - What's failing and why?
- **Security** - Who's accessing what?

But observability must not compromise:

- **Privacy** - No sensitive data in metrics
- **Security** - Metrics don't leak capabilities
- **Performance** - Minimal overhead
- **Storage** - Observability data is bounded

Metrics

Core Metrics

```
(define vault-metrics
  '(;; Storage
    (storage.used.bytes gauge "Total bytes used")
    (storage.available.bytes gauge "Bytes available")
    (storage.objects.count gauge "Number of objects")

    ;; Operations
    (cas.put.count counter "CAS put operations")
    (cas.put.bytes counter "Bytes written")
    (cas.put.duration histogram "Put latency")
    (cas.get.count counter "CAS get operations")
    (cas.get.bytes counter "Bytes read")
    (cas.get.duration histogram "Get latency")

    ;; Soup
    (soup.queries.count counter "Soup queries")
    (soup.queries.duration histogram "Query latency")
    (soup.index.size.bytes gauge "Index size")

    ;; Network
    (network.connections.active gauge "Active connections")
    (network.bytes.sent counter "Bytes sent")
    (network.bytes.received counter "Bytes received")

    ;; Replication
    (replication.lag.seconds gauge "Replication lag")
    (replication.objects.pending gauge "Objects pending sync")

    ;; Errors
    (errors.total counter "Total errors" labels: (type))
    (errors.rate gauge "Error rate per second")))
```

Metric Collection

```
(define metrics-registry (make-hash-table))

(define (register-metric name type help #!key labels)
  (hash-table-set! metrics-registry name
    `((type . ,type)
      (help . ,help)
      (labels . ,labels)
      (value . ,(case type
                  ((counter) 0)
                  ((gauge) 0)
                  ((histogram) (make-histogram-buckets)))))))

(define (metric-inc! name #!key (delta 1) labels)
  (let ((metric (hash-table-ref metrics-registry name)))
    (case (assoc-ref metric 'type)
      ((counter gauge)
       (set! (assoc-ref metric 'value)
             (+ (assoc-ref metric 'value) delta))))))

(define (metric-set! name value #!key labels)
  (let ((metric (hash-table-ref metrics-registry name)))
    (set! (assoc-ref metric 'value) value)))
```

```scheme
(define (metric-observe! name value #!key labels)
  (let ((metric (hash-table-ref metrics-registry name)))
    (histogram-observe! (assoc-ref metric 'value) value)))
```

Histogram Buckets

```scheme
(define (make-histogram-buckets)
  "Default latency buckets in milliseconds"
  (let ((buckets '(1 5 10 25 50 100 250 500 1000 2500 5000 10000)))
    (map (lambda (b) (cons b 0)) buckets)))

(define (histogram-observe! histogram value)
  (for-each (lambda (bucket)
              (when (<= value (car bucket))
                (set-cdr! bucket (+ (cdr bucket) 1))))
            histogram))
```

Tracing

Trace Structure

```scheme
(define (make-trace operation)
  `(trace
    (trace-id ,(generate-trace-id))
    (span-id ,(generate-span-id))
    (parent-span-id #f)
    (operation ,operation)
    (start-time ,(current-time-ns))
    (end-time #f)
    (status pending)
    (attributes ())
    (events ())))

(define (trace-start! trace)
  (set! (assoc-ref trace 'start-time) (current-time-ns))
  trace)

(define (trace-end! trace status)
  (set! (assoc-ref trace 'end-time) (current-time-ns))
  (set! (assoc-ref trace 'status) status)
  (record-trace! trace))
```

Span Context

```scheme
(define current-trace (make-parameter #f))
(define current-span (make-parameter #f))

(define (with-span operation proc)
  "Execute proc within a traced span"
  (let* ((parent (current-span))
         (span (make-span operation (and parent (span-id parent)))))
    (parameterize ((current-span span))
      (span-start! span)
      (guard (ex
              (else
               (span-error! span ex)
               (span-end! span 'error)
               (raise ex)))
        (let ((result (proc)))
          (span-end! span 'ok)
          result)))))

;; Example usage
(define (traced-cas-get hash)
  (with-span "cas.get"
    (lambda ()
      (span-set-attribute! "hash" hash)
      (cas-get hash))))
```

Distributed Tracing

```scheme
;; Propagate trace context across vault boundaries
(define (inject-trace-context headers)
  "Inject trace context into outgoing request"
  (let ((span (current-span)))
    (when span
      (hash-table-set! headers "X-Trace-Id" (span-trace-id span))
      (hash-table-set! headers "X-Span-Id" (span-id span)))))

(define (extract-trace-context headers)
  "Extract trace context from incoming request"
  (let ((trace-id (hash-table-ref headers "X-Trace-Id" #f))
        (parent-span-id (hash-table-ref headers "X-Span-Id" #f)))
    (when trace-id
      (current-trace trace-id)
      (make-span-with-parent parent-span-id))))
```

Logging

Structured Logging

```
(define (log level message #!rest attributes)
  "Emit structured log entry"
  (let ((entry `((timestamp . ,(current-time-iso8601))
                 (level . ,level)
                 (message . ,message)
                 (vault . ,(vault-id))
                 (trace-id . ,(and (current-span) (span-trace-id (current-span))))
                 ,@(plist->alist attributes))))
    (emit-log entry)))

(define (log-debug message . attrs) (apply log 'debug message attrs))
(define (log-info message . attrs) (apply log 'info message attrs))
(define (log-warn message . attrs) (apply log 'warn message attrs))
(define (log-error message . attrs) (apply log 'error message attrs))

;; Example
(log-info "Object stored"
          'hash hash
          'size (bytevector-length data)
          'duration-ms elapsed)
```

Log Levels

```
(define log-levels '((trace . 0) (debug . 1) (info . 2)
                     (warn . 3) (error . 4) (fatal . 5)))

(define current-log-level (make-parameter 'info))

(define (log-enabled? level)
  (>= (assoc-ref log-levels level)
      (assoc-ref log-levels (current-log-level))))
```

Log Sinks

```
;; Multiple output destinations
(define log-sinks '())

(define (add-log-sink! sink)
  (set! log-sinks (cons sink log-sinks)))

(define (emit-log entry)
  (when (log-enabled? (assoc-ref entry 'level))
    (for-each (lambda (sink)
                (sink-write sink entry))
              log-sinks)))

;; Sink implementations
(define (make-file-sink path)
  (lambda (entry)
    (with-output-to-file path
      (lambda () (write-json entry) (newline))
      'append)))

(define (make-soup-sink)
  (lambda (entry)
    (soup-put entry type: 'log-entry)))
```

Health Checks

Health Endpoints

```
(define (health-check)
  "Comprehensive health check"
  `((status . ,(if (all-healthy?) 'healthy 'unhealthy))
    (checks . ((storage . ,(check-storage))
               (network . ,(check-network))
               (replication . ,(check-replication))
               (keys . ,(check-keys))))))

(define (check-storage)
  `((status . ,(if (storage-accessible?) 'healthy 'unhealthy))
    (used . ,(storage-used))
    (available . ,(storage-available))
    (percent . ,(storage-percent-used))))

(define (check-network)
  `((status . ,(if (network-healthy?) 'healthy 'unhealthy))
    (peers . ,(connected-peer-count))
    (latency-ms . ,(average-peer-latency))))

(define (check-replication)
  `((status . ,(if (replication-healthy?) 'healthy 'unhealthy))
    (lag-seconds . ,(replication-lag))
    (pending . ,(pending-replication-count))))
```

Readiness vs Liveness

```
;; Liveness: is the process running?
(define (liveness-check)
  '((status . alive)))

;; Readiness: can we serve requests?
(define (readiness-check)
  (if (and (storage-accessible?)
           (keys-available?)
           (not (vault-readonly?)))
      '((status . ready))
      '((status . not-ready))))
```

Alerting

Alert Rules

```
(define alert-rules
  `((storage-critical
     (condition (> (metric-value 'storage.percent.used) 95))
     (severity critical)
     (message "Storage critically low"))

    (storage-warning
     (condition (> (metric-value 'storage.percent.used) 80))
     (severity warning)
     (message "Storage running low"))

    (error-rate-high
     (condition (> (metric-value 'errors.rate) 10))
     (severity warning)
     (message "High error rate detected"))

    (replication-lag
     (condition (> (metric-value 'replication.lag.seconds) 300))
     (severity warning)
     (message "Replication lag exceeds 5 minutes"))

    (peer-disconnected
     (condition (< (metric-value 'network.peers.connected) 1))
     (severity critical)
     (message "No peers connected"))))
```

Alert Evaluation

```
(define (evaluate-alerts)
  "Check all alert conditions"
  (filter-map
    (lambda (rule)
      (when (eval-condition (assoc-ref rule 'condition))
        (fire-alert rule)))
    alert-rules))

(define (fire-alert rule)
  (let ((alert `((name . ,(car rule))
                 (severity . ,(assoc-ref (cdr rule) 'severity))
                 (message . ,(assoc-ref (cdr rule) 'message))
                 (timestamp . ,(current-time)))))
    (audit-append action: 'alert-fired alert: alert)
    (notify-alert alert)
    alert))
```

Dashboards

Metric Export

```
;; Prometheus format
(define (export-prometheus)
  (with-output-to-string
    (lambda ()
      (for-each
        (lambda (metric)
          (let ((name (car metric))
                (data (cdr metric)))
            (format #t "# HELP ~a ~a~%" name (assoc-ref data 'help))
            (format #t "# TYPE ~a ~a~%" name (assoc-ref data 'type))
            (format #t "~a ~a~%" name (assoc-ref data 'value))))
        (hash-table->alist metrics-registry)))))
```

Status Page

```
(define (status-page)
  "Generate human-readable status"
  `(vault-status
    (health . ,(health-check))
    (uptime . ,(vault-uptime))
    (version . ,(vault-version))
    (metrics . ((storage . ,(storage-metrics))
                (operations . ,(operation-metrics))
                (network . ,(network-metrics))))
    (recent-errors . ,(recent-errors 10))))
```

Privacy

Metric Sanitization

```
;; Never expose sensitive data in metrics
(define (sanitize-metric-labels labels)
  (map (lambda (label)
         (case (car label)
           ((hash) (cons 'hash "[REDACTED]"))
           ((principal) (cons 'principal (hash-principal (cdr label))))
           (else label)))
       labels))

(define (safe-metric name value #!key labels)
  (metric-set! name value labels: (sanitize-metric-labels labels)))
```

Aggregate Metrics Only

```
;; Expose only aggregates, not individual operations
(define (operation-metrics)
  `((total-operations . ,(metric-value 'cas.operations.total))
    (operations-per-second . ,(metric-value 'cas.operations.rate))
    (average-latency-ms . ,(metric-value 'cas.latency.average))
    (p99-latency-ms . ,(metric-value 'cas.latency.p99))))
```

Storage

Metric Retention

```
;; Metrics stored in soup with TTL
(define *metric-retention* (* 30 24 3600))  ; 30 days

(define (store-metric-snapshot)
  "Periodic metric snapshot"
  (let ((snapshot `((timestamp . ,(current-time))
                    (metrics . ,(export-all-metrics)))))
    (soup-put snapshot
      type: 'metric-snapshot
      ttl: *metric-retention*)))
```

```
(define (gc-old-metrics)
  "Remove expired metric snapshots"
  (let ((expired (soup-query type: 'metric-snapshot
                             timestamp: (< (- (current-time) *metric-retention*)))))
    (for-each soup-delete! expired)))
```

Trace Sampling

```
;; Sample traces to control storage
(define *trace-sample-rate* 0.1)  ; 10%

(define (should-sample-trace?)
  (< (random 1.0) *trace-sample-rate*))

(define (record-trace! trace)
  (when (or (trace-has-error? trace)
            (should-sample-trace?))
    (soup-put trace type: 'trace)))
```

References

1. OpenTelemetry - Observability framework
2. Prometheus - Monitoring system
3. RFC-003: Cryptographic Audit Trail
4. RFC-028: Error Handling

Changelog

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

Implementation Status: Draft **Dependencies:** soup, audit **Integration:** Operations, alerting, dashboards