RFC-026: Garbage Collection

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

Abstract

This RFC specifies garbage collection for the Library of Cyberspace: how vaults identify and reclaim storage from unreferenced objects while preserving pinned content, respecting tombstones, and maintaining audit trails. Content-addressed storage requires careful GC to avoid data loss.

Motivation

Content-addressed storage accumulates objects forever unless actively pruned:

- **Orphaned objects** - No longer referenced by any root
- **Superseded versions** - Old versions after updates
- **Failed uploads** - Partial or abandoned writes
- **Temporary objects** - Intermediate computation results

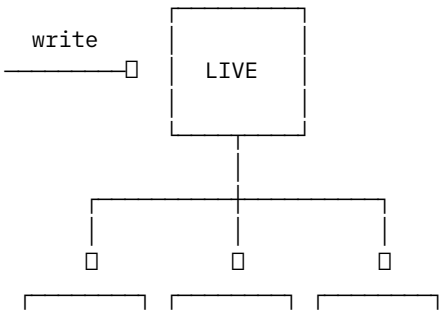
But deletion is dangerous:

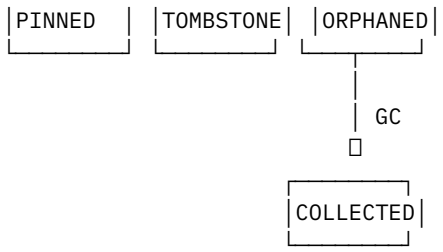
- **Hash as capability** - Someone may hold the hash
- **Lazy replication** - Remote vaults may need it later
- **Audit requirements** - May need historical data
- **Resurrection** - Deleted objects may be re-added

GC must be conservative and auditable.

Object Lifecycle

States





State Transitions

```
(define (object-state hash)
  (cond
    ((pinned? hash) 'pinned)
    ((tombstoned? hash) 'tombstone)
    ((referenced? hash) 'live)
    (else 'orphaned)))

(define (can-collect? hash)
  (and (eq? (object-state hash) 'orphaned)
       (not (in-grace-period? hash))
       (not (pending-replication? hash))))
```

Reference Counting

Direct References

```
;; Track incoming references
(define ref-counts (make-hash-table))

(define (add-reference from-hash to-hash)
  (let ((count (hash-table-ref ref-counts to-hash 0)))
    (hash-table-set! ref-counts to-hash (+ count 1))
    (audit-append action: `(add-ref ,from-hash ,to-hash))))

(define (remove-reference from-hash to-hash)
  (let ((count (hash-table-ref ref-counts to-hash 0)))
    (hash-table-set! ref-counts to-hash (max 0 (- count 1)))
    (audit-append action: `(remove-ref ,from-hash ,to-hash))))

(define (reference-count hash)
  (hash-table-ref ref-counts hash 0))
```

Root References

```
;; GC roots - objects that are always reachable
(define gc-roots (make-hash-set))

(define (add-gc-root hash reason)
  (hash-set-add! gc-roots hash)
  (audit-append action: `(add-root ,hash ,reason)))

(define (remove-gc-root hash)
  (hash-set-remove! gc-roots hash)
  (audit-append action: `(remove-root ,hash)))

(define (gc-root? hash)
  (hash-set-member? gc-roots hash))
```

Implicit Roots

```
;; Some objects are implicitly rooted
(define (implicit-root? hash)
  (or (pinned? hash)
      (soup-object-type? hash 'certificate)  ; Certs are roots
      (soup-object-type? hash 'audit-entry)  ; Audit is sacred
      (recent-write? hash)))                 ; Grace period
```

Mark and Sweep

Mark Phase

```
(define (mark-reachable)
  "Mark all objects reachable from roots"
  (let ((marked (make-hash-set)))
    (define (mark hash)
      (unless (hash-set-member? marked hash)
        (hash-set-add! marked hash)
        (for-each mark (object-references hash))))

    ;; Mark from explicit roots
    (for-each mark (hash-set->list gc-roots))

    ;; Mark from implicit roots
    (for-each (lambda (hash)
                (when (implicit-root? hash)
                  (mark hash)))
              (all-object-hashes))
```

```
    marked))
```

Sweep Phase

```
(define (sweep marked)
  "Collect unmarked objects"
  (let ((collected '()))
    (for-each
      (lambda (hash)
        (unless (hash-set-member? marked hash)
          (when (can-collect? hash)
            (set! collected (cons hash collected))
            (collect-object! hash))))
      (all-object-hashes))
    collected))

(define (collect-object! hash)
  "Remove object from storage"
  (let ((obj (cas-get hash)))
    (audit-append
      action: 'gc-collect
      hash: hash
      size: (object-size obj)
      age: (object-age obj))
    (cas-delete! hash)))
```

Full GC

```
(define (gc-full)
  "Perform full garbage collection"
  (let ((start (current-time)))
    (audit-append action: 'gc-start type: 'full)

    (let* ((marked (mark-reachable))
           (collected (sweep marked)))

      (audit-append
        action: 'gc-complete
        type: 'full
        duration: (- (current-time) start)
        marked: (hash-set-size marked)
        collected: (length collected)
        bytes-freed: (sum (map object-size collected)))

      collected)))
```

Incremental GC

Generational Collection

```
;; Objects partitioned by age
(define generations
  '((young . 86400)      ; < 1 day
    (middle . 2592000)   ; < 30 days
    (old . #f)))         ; >= 30 days

(define (object-generation hash)
  (let ((age (object-age hash)))
    (cond
      ((< age 86400) 'young)
      ((< age 2592000) 'middle)
      (else 'old))))

(define (gc-generation gen)
  "Collect only specified generation"
  (let ((candidates (filter (lambda (h)
                              (eq? (object-generation h) gen))
                            (all-object-hashes))))
    (gc-candidates candidates)))
```

Write Barrier

```
;; Track modified objects for incremental GC
(define modified-set (make-hash-set))

(define (cas-put-with-barrier data)
  (let ((hash (cas-put data)))
    (hash-set-add! modified-set hash)
    hash))

(define (gc-incremental)
  "Collect recently modified objects if orphaned"
  (let ((candidates (hash-set->list modified-set)))
    (hash-set-clear! modified-set)
    (gc-candidates candidates)))
```

Concurrent GC

```
;; GC runs concurrently with mutations
(define gc-lock (make-mutex))
(define gc-running? #f)
```

```
(define (gc-concurrent)
  "Run GC without stopping the world"
  (when (mutex-try-lock! gc-lock)
    (set! gc-running? #t)
    (let ((snapshot (snapshot-roots)))
      ;; Mark phase uses snapshot
      (let ((marked (mark-from-snapshot snapshot)))
        ;; Sweep only clearly dead objects
        (sweep-conservative marked)))
    (set! gc-running? #f)
    (mutex-unlock! gc-lock)))
```

Pinning

Pin Management

```
(define pins (make-hash-table))

(define (pin! hash reason #!key duration)
  "Protect object from GC"
  (hash-table-set! pins hash
    `((reason . ,reason)
      (pinned-at . ,(current-time))
      (expires . ,(and duration (+ (current-time) duration)))
      (pinned-by . ,(current-principal))))
  (audit-append action: `(pin ,hash ,reason)))

(define (unpin! hash)
  "Allow object to be collected"
  (hash-table-delete! pins hash)
  (audit-append action: `(unpin ,hash)))

(define (pinned? hash)
  (let ((pin (hash-table-ref pins hash #f)))
    (and pin
         (or (not (assoc-ref pin 'expires))
             (> (assoc-ref pin 'expires) (current-time))))))
```

Transitive Pinning

```
(define (pin-tree! root-hash reason)
  "Pin object and all objects it references"
  (let ((visited (make-hash-set)))
    (define (pin-recursive hash)
```

```
      (unless (hash-set-member? visited hash)
        (hash-set-add! visited hash)
        (pin! hash reason)
        (for-each pin-recursive (object-references hash))))
    (pin-recursive root-hash)))
```

Tombstones

Tombstone Handling

```
;; Tombstones are never collected
(define (tombstoned? hash)
  (let ((obj (soup-get hash)))
    (and obj (eq? (soup-object-type obj) 'tombstone))))

;; Tombstones prevent resurrection
(define (cas-put-checked data)
  (let ((hash (content-hash data)))
    (when (tombstoned? hash)
      (error "Cannot resurrect tombstoned object" hash))
    (cas-put data)))
```

Tombstone Expiry

```
;; Optional: tombstones can expire
(define (tombstone-expired? hash)
  (let ((tomb (soup-get hash)))
    (and tomb
         (assoc-ref (soup-object-metadata tomb) 'expires)
         (< (assoc-ref (soup-object-metadata tomb) 'expires)
            (current-time)))))

(define (gc-expired-tombstones)
  "Remove expired tombstones"
  (for-each
    (lambda (hash)
      (when (and (tombstoned? hash) (tombstone-expired? hash))
        (collect-object! hash)))
    (all-object-hashes)))
```

Grace Periods

Write Grace Period

```
;; Recently written objects are protected
(define *write-grace-period* 86400)  ; 24 hours

(define (recent-write? hash)
  (let ((obj (soup-get hash)))
    (and obj
         (< (- (current-time) (soup-object-created obj))
            *write-grace-period*))))
```

Replication Grace Period

```
;; Objects pending replication are protected
(define pending-replication (make-hash-set))

(define (mark-pending-replication hash vaults)
  (hash-set-add! pending-replication hash)
  (audit-append action: `(pending-replication ,hash ,vaults)))

(define (clear-pending-replication hash)
  (hash-set-remove! pending-replication hash))

(define (pending-replication? hash)
  (hash-set-member? pending-replication hash))
```

Distributed GC

Coordinated Collection

```
;; Multi-vault GC requires coordination
(define (distributed-gc vaults)
  "Coordinate GC across vault federation"
  ;; Phase 1: Gather root sets
  (let ((root-sets (map vault-roots vaults)))
    ;; Phase 2: Compute global reachability
    (let ((global-marked (union-all root-sets)))
      ;; Phase 3: Local sweep (conservative)
      (for-each (lambda (vault)
                  (vault-sweep vault global-marked))
                vaults))))
```

Remote Reference Tracking

```
;; Track references from remote vaults
(define remote-refs (make-hash-table))

(define (add-remote-reference vault-id hash)
  (let ((refs (hash-table-ref remote-refs hash '())))
    (hash-table-set! remote-refs hash (cons vault-id refs))))

(define (remove-remote-reference vault-id hash)
  (let ((refs (hash-table-ref remote-refs hash '())))
    (hash-table-set! remote-refs hash (delete vault-id refs))))

(define (has-remote-references? hash)
  (not (null? (hash-table-ref remote-refs hash '()))))
```

Lease-Based Collection

```
;; Remote vaults lease objects
(define leases (make-hash-table))

(define (grant-lease hash vault-id duration)
  (let ((expires (+ (current-time) duration)))
    (hash-table-set! leases hash
      (cons (cons vault-id expires)
            (hash-table-ref leases hash '())))))

(define (lease-active? hash)
  (let ((hash-leases (hash-table-ref leases hash '())))
    (any (lambda (lease)
           (> (cdr lease) (current-time)))
         hash-leases)))
```

GC Scheduling

Triggers

```
;; GC triggered by various conditions
(define (should-gc?)
  (or (> (storage-usage-percent) 80)      ; Disk pressure
      (> (orphan-estimate) 10000)         ; Many orphans
      (> (time-since-last-gc) 86400)))    ; Daily minimum

(define (gc-schedule)
  "Run appropriate GC based on conditions"
```

```
  (cond
    ((> (storage-usage-percent) 95)
      (gc-full))                          ; Emergency: full GC
    ((> (storage-usage-percent) 80)
      (gc-generation 'young))             ; Pressure: young gen
    (else
      (gc-incremental))))                 ; Normal: incremental
```

Background GC

```
(define gc-thread #f)

(define (start-gc-daemon interval)
  (set! gc-thread
    (thread-start!
      (make-thread
        (lambda ()
          (let loop ()
            (thread-sleep! interval)
            (when (should-gc?)
              (gc-schedule))
            (loop)))))))
```

Safety Mechanisms

Dry Run

```
(define (gc-dry-run)
  "Report what would be collected without collecting"
  (let* ((marked (mark-reachable))
         (would-collect (filter (lambda (h)
                                  (not (hash-set-member? marked h)))
                                (all-object-hashes))))
    `((would-collect . ,(length would-collect))
      (bytes . ,(sum (map object-size would-collect)))
      (samples . ,(take would-collect 10)))))
```

Collection Log

```
;; Every collection is logged with recovery info
(define (collect-with-log! hash)
  (let ((obj (cas-get hash)))
    ;; Log enough to reconstruct if needed
    (audit-append
      action: 'gc-collect
```

```
      hash: hash
      size: (object-size obj)
      type: (soup-object-type obj)
      references: (object-references hash)
      metadata: (soup-object-metadata obj))
    (cas-delete! hash)))
```

Recovery

```
;; Recover recently collected object from audit log
(define (gc-recover hash)
  "Attempt to recover collected object"
  (let ((entry (find (lambda (e)
                       (and (eq? (audit-action e) 'gc-collect)
                            (equal? (audit-hash e) hash)))
                     (recent-audit-entries))))
    (if entry
        (error "Object collected, metadata preserved in audit"
               (audit-metadata entry))
        (error "Object not found in recent collections"))))
```

Metrics

GC Statistics

```
(define gc-stats
  `((collections . 0)
    (bytes-freed . 0)
    (objects-freed . 0)
    (total-time . 0)
    (last-gc . #f)))

(define (update-gc-stats collected duration)
  (set! gc-stats
    `((collections . ,(+ 1 (assoc-ref gc-stats 'collections)))
      (bytes-freed . ,(+ (sum (map object-size collected))
                         (assoc-ref gc-stats 'bytes-freed)))
      (objects-freed . ,(+ (length collected)
                           (assoc-ref gc-stats 'objects-freed)))
      (total-time . ,(+ duration (assoc-ref gc-stats 'total-time)))
      (last-gc . ,(current-time)))))
```

Monitoring

```
;; Expose GC metrics
(define (gc-metrics)
  `((storage-used . ,(storage-used))
    (storage-total . ,(storage-total))
    (object-count . ,(object-count))
    (orphan-estimate . ,(orphan-estimate))
    (pinned-count . ,(hash-table-size pins))
    (gc-stats . ,gc-stats)))
```

Security Considerations

GC as Side Channel

```
;; GC timing can leak information about object references
;; Use constant-time operations where possible
(define (constant-time-mark hash)
  (let ((refs (object-references hash)))
    ;; Always process same number of refs
    (for-each mark (pad-list refs *max-refs*))))
```

Denial of Service

```
;; Prevent GC starvation attacks
(define *max-pins-per-principal* 10000)

(define (pin-with-limit! hash reason)
  (let ((count (principal-pin-count (current-principal))))
    (when (> count *max-pins-per-principal*)
      (error "Pin limit exceeded"))
    (pin! hash reason)))
```

References

1. The Garbage Collection Handbook - Jones, Hosking, Moss
2. On-the-Fly Garbage Collection - Dijkstra et al.
3. RFC-020: Content-Addressed Storage
4. RFC-003: Cryptographic Audit Trail

Changelog

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

Implementation Status: Draft **Dependencies:** cas, soup, audit **Integration:** Storage management, replication, vault maintenance