RFC-020: Content-Addressed Storage

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

Abstract

This RFC specifies content-addressed storage (CAS) for the Library of Cyberspace: a storage model where data is addressed by its cryptographic hash rather than by location. Content addressing provides immutability guarantees, automatic deduplication, and tamper-evident storage.

Motivation

Traditional storage systems use location-based addressing:

- **DECtape**: Physical position on magnetic tape
- **Filesystems**: Path names (`/home/ddp/file.txt`)
- **Databases**: Row IDs, primary keys
- **URLs**: Server + path (`https://example.com/doc.pdf`)

Location-based addressing has fundamental problems:

1. **Mutability** - Same address can point to different content over time
2. **Link rot** - Addresses become invalid when content moves
3. **Duplication** - Identical content stored multiple times
4. **No verification** - Address doesn't prove content integrity

Content-addressed storage inverts this:

```
Location-based:  address → content (many-to-one, mutable)
Content-based:   content → address (one-to-one, immutable)
```

The address IS the content's cryptographic fingerprint. If the content changes, the address changes. If two files have the same address, they are byte-for-byte identical.

Content Addressing Model

Hash as Address

```
;; Content determines address
(define (content-address data)
  (sha256 data))
```

```scheme
;; Store by hash
(define (cas-store data)
  (let ((hash (content-address data)))
    (write-blob (hash->path hash) data)
    hash))

;; Retrieve by hash
(define (cas-retrieve hash)
  (let ((data (read-blob (hash->path hash))))
    (if (equal? hash (content-address data))
        data
        (error "Content tampered"))))
```

Properties

Property	Location-Based	Content-Addressed
Address stability	Unstable	Permanent
Content verification	External	Intrinsic
Deduplication	Manual	Automatic
Mutability	Mutable	Immutable
Link rot	Common	Impossible

Hash Function Requirements

The hash function MUST be:

1. **Collision-resistant** - Computationally infeasible to find two inputs with same hash
2. **Preimage-resistant** - Cannot derive content from hash
3. **Deterministic** - Same content always produces same hash
4. **Fast** - Practical for large objects

Specified hash: SHA-256 (32 bytes, 64 hex characters)

```scheme
;; Example content address
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
;; This is the SHA-256 of the empty string
```

Storage Architecture

Object Store

```
.cas/
├── objects/
│   ├── e3/
│   │   └── b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855
│   ├── a7/
│   │   └── ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
│   └── ...
├── refs/
│   ├── HEAD
│   └── tags/
└── index
```

Sharding: First two hex characters of hash form directory name (256 buckets).

Object Types

```
;; Blob - raw data
(cas-blob
  (hash "sha256:...")
  (size 1024)
  (data #${...}))

;; Tree - directory structure
(cas-tree
  (hash "sha256:...")
  (entries
    (("README.md" blob "sha256:abc...")
     ("src" tree "sha256:def...")
     ("lib" tree "sha256:ghi..."))))

;; Commit - snapshot with metadata
(cas-commit
  (hash "sha256:...")
  (tree "sha256:...")
  (parent "sha256:..." | #f)
  (author "ddp@eludom.net")
  (timestamp 1767700000)
  (message "Initial commit"))
```

Merkle Trees

Directories are trees of hashes. The root hash commits to all content:

```
                    root: sha256:abc...
                         /          \
              src: sha256:def     lib: sha256:ghi
               /        \                |
        main.scm     test.scm      util.scm
        sha256:111   sha256:222    sha256:333
```

Changing ANY file changes ALL parent hashes up to root.

Operations

Store

```
(define (cas-put content)
  "Store content, return hash"
  (let* ((hash (sha256 content))
         (path (hash->path hash)))
    (unless (file-exists? path)
      (write-blob path content))
    hash))
```

Deduplication: If hash already exists, storage is a no-op.

Retrieve

```
(define (cas-get hash)
  "Retrieve content by hash, verify integrity"
  (let* ((path (hash->path hash))
         (content (read-blob path)))
    (unless (equal? hash (sha256 content))
      (error "Content integrity failure" hash))
    content))
```

Self-verifying: Every retrieval confirms integrity.

Exists

```
(define (cas-exists? hash)
  "Check if content exists"
  (file-exists? (hash->path hash)))
```

Delete

```
(define (cas-gc roots)
  "Garbage collect unreachable objects"
  (let ((reachable (trace-reachable roots)))
    (for-each
      (lambda (hash)
        (unless (set-member? reachable hash)
          (delete-file (hash->path hash))))
      (all-objects))))
```

Note: Deletion requires garbage collection from known roots.

Naming Layer

Content addresses are not human-friendly. A naming layer maps names to hashes:

References

```scheme
;; Mutable reference to immutable content
(define (ref-set name hash)
  (write-file (ref-path name) hash))

(define (ref-get name)
  (read-file (ref-path name)))

;; Example
(ref-set "HEAD" "sha256:abc123...")
(ref-get "HEAD") ; => "sha256:abc123..."
```

Tags

```scheme
;; Immutable named reference
(define (tag-create name hash #!key message signature)
  (let ((tag-content
         `(tag
           (name ,name)
           (object ,hash)
           (message ,message)
           (signature ,signature))))
    (cas-put (serialize tag-content))))
```

Directory Entries

```scheme
;; Human name -> content hash
(define library-index
  '(("RFC-000" . "sha256:e3b0c44...")
    ("RFC-001" . "sha256:a7ffc6f...")
    ("RFC-002" . "sha256:b94d27b...")))
```

The Soup: Object Directory

Content-addressed storage provides retrieval by hash, but discovery requires an object directory. The **Soup** (inspired by NewtonOS) provides a unified view of all objects with rich metadata.

Philosophy

> "The soup is infinite." - Objects swim in a queryable sea of metadata.

The Soup inverts the traditional filesystem model:

- **Filesystem**: Navigate hierarchy to find objects
- **Soup**: Query attributes to discover objects

Object Enumeration

```
(define (soup)
  "List all objects in the vault with metadata"
  (append
    (soup-releases)      ; Signed releases
    (soup-archives)      ; Sealed archives
    (soup-keys)          ; Cryptographic keys
    (soup-audit-entries) ; Audit trail
    (soup-commits)))     ; Recent commits
```

Rich Metadata

Every object carries crypto metadata - the ciphers, hashes, and keys involved:

```
;; Archive object
(soup-object
  (name "1.0.0")
  (type archive)
  (size "1.2MB")
  (crypto (zstd sha256 "fe378a78...")))

;; Key object
(soup-object
  (name "vault-signing")
  (type key)
  (size "64B")
  (crypto (ed25519/256 public sign
           "sha256:a1b2c3d4..."    ; fingerprint
           "id:ddp@eludom.net"     ; identity
           "2026-01-07")))         ; creation date

;; Release object
(soup-object
  (name "0.1.0")
  (type release)
  (size "313B")
  (crypto (ed25519 sha512 "abc123...")))
```

Querying the Soup

```
;; Find all signed objects
(soup-query type: 'release)

;; Find objects using specific algorithm
(soup-query crypto: 'ed25519)

;; Find objects by size range
(soup-query min-size: 1000 max-size: 100000)

;; Find objects by content hash prefix
(soup-query hash-prefix: "fe378")
```

Display Format

The soup displays as a compact, human-readable listing:

```
$ seal-soup
vault-signing, 64B, ed25519/256, public, sign, sha256:a1b2c3d4...
0.1.0, 313B, signed, ed25519, sha512
1.0.0, 1.2MB, zstd, sha256, fe378a78...
audit/1, sealed, 2026-01-07T10:30:00Z
```

Soup as Index

The Soup can be serialized as a content-addressed index:

```
(define (soup-snapshot)
  "Create content-addressed snapshot of current soup"
  (let* ((entries (soup))
         (serialized (serialize entries))
         (hash (cas-put serialized)))
    (ref-set "soup/HEAD" hash)
    hash))
```

This enables: - **Time travel**: Load historical soup snapshots - **Replication**: Sync soup indexes between vaults - **Verification**: Prove soup state at a point in time

---

Integration with Library of Cyberspace

Vault Integration

The Vault (RFC-006) uses content addressing internally via Git:

```
;; Git objects ARE content-addressed
(define (git-hash-object content)
  (sha1 (string-append "blob " (number->string (blob-length content)) "\x00" content)))
```

CAS extends this with SHA-256 and Library-specific semantics.

Archive Storage

Sealed archives (RFC-018) can be stored by content address:

```
(define (archive-to-cas archive-path)
  (let* ((content (read-blob archive-path))
         (hash (cas-put content)))
    (ref-set (string-append "archives/" (archive-version archive-path)) hash)
    hash))
```

Replication

Content addressing enables efficient replication (RFC-001):

```
;; Only transfer objects receiver doesn't have
(define (replicate-to remote root-hash)
  (for-each
    (lambda (hash)
      (unless (remote-has? remote hash)
        (remote-put remote hash (cas-get hash))))
    (trace-reachable root-hash)))
```

SPKI Integration

Content hashes can be authorization subjects (RFC-004):

```
;; Grant permission to specific content
(spki-cert
  (issuer publisher-key)
  (subject (hash sha256 "abc123..."))
  (permission read)
  (validity (not-after "2027-01-01")))
```

Chunking for Large Objects

Large files are split into chunks for:

1. **Deduplication** at sub-file granularity
2. **Parallel transfer**
3. **Incremental updates**

Fixed-Size Chunking

```
(define chunk-size (* 64 1024))  ; 64KB

(define (chunk-fixed data)
  (let loop ((offset 0) (chunks '()))
    (if (>= offset (blob-length data))
        (reverse chunks)
        (loop (+ offset chunk-size)
              (cons (blob-copy data offset (min chunk-size (- (blob-length data) offset)))
                    chunks)))))
```

Content-Defined Chunking (Rabin fingerprinting)

```
;; Chunk boundaries determined by content
;; Survives insertions/deletions better than fixed-size
(define (chunk-rabin data)
  (let ((window-size 48)
        (min-chunk 2048)
        (max-chunk 65536)
        (mask #x0fff))  ; Average 4KB chunks
    ...))
```

Chunk Tree

```
(cas-chunked
  (hash "sha256:...")       ; Root hash
  (size 10485760)           ; 10MB original
  (chunks
    ("sha256:aaa..." 65536)
    ("sha256:bbb..." 65536)
    ("sha256:ccc..." 32768)
    ...))
```

Introspection

The Library is fully introspective: it stores, addresses, and reasons about itself.

Self-Hosting

The system contains its own description:

```
;; The RFCs are in the CAS
(cas-get (ref-get "rfc/020"))  ; => This document

;; The code is in the CAS
```

```
(cas-get (ref-get "src/vault.scm"))  ; => Implementation

;; The schema is in the CAS
(cas-get (ref-get "schema/soup-object"))  ; => Soup object definition
```

Meta-Objects

Objects that describe objects:

```
;; Schema for soup objects (itself a soup object)
(soup-object
  (name "schema/soup-object")
  (type schema)
  (size "412B")
  (crypto (sha256 "def456..."))
  (describes soup-object))

;; The soup can enumerate itself
(soup-query type: 'schema)  ; => All schemas including this one
```

Reflexive Queries

The soup answers questions about itself:

```
;; What types exist?
(soup-types)  ; => (archive release key audit commit schema ...)

;; What crypto algorithms are in use?
(soup-algorithms)  ; => (sha256 ed25519 zstd age ...)

;; What is the total size of the vault?
(soup-total-size)  ; => 47.3MB

;; How much deduplication?
(soup-dedup-ratio)  ; => 0.73 (27% space saved)
```

Provenance

Every object knows its origin:

```
(soup-object
  (name "rfc-020.pdf")
  (type blob)
  (size "89KB")
  (crypto (sha256 "abc123..."))
  (provenance
    (created-by "ddp@eludom.net")
    (created-at 1767700000)
    (derived-from "sha256:fff888...")  ; The markdown source
    (tool "pandoc 3.1")))
```

Provenance chains are themselves content-addressed:

```
;; Trace full history
(define (provenance-chain hash)
  (let ((obj (soup-get hash)))
    (if (soup-object-derived-from obj)
        (cons obj (provenance-chain (soup-object-derived-from obj)))
        (list obj))))
```

Audit of Audits

The audit trail audits itself:

```
;; Audit entry for an audit entry
(audit-entry
  (id 42)
  (actor vault-key)
  (action (audit-append 41))  ; Recorded adding entry 41
  (timestamp 1767700100))

;; The audit trail is in the soup
(soup-query type: 'audit)  ; => All audit entries as soup objects
```

Bootstrapping

The system can describe how to build itself:

```
;; Bootstrap manifest - everything needed to reconstruct
(bootstrap-manifest
  (hash "sha256:bootstrap...")
  (contents
    (("src/" tree "sha256:src...")
     ("rfcs/" tree "sha256:rfcs...")
     ("keys/" tree "sha256:keys...")
     ("schema/" tree "sha256:schema...")))
  (build-instructions
    "Load src/boot.scm, call (bootstrap)"))

;; Verify bootstrap integrity
(define (verify-bootstrap)
  (let ((manifest (cas-get (ref-get "bootstrap"))))
    (for-each verify-tree (manifest-contents manifest))))
```

The Library Contains Itself

```
│            LIBRARY OF CYBERSPACE        │
│  ╭───────────────────────────────────╮  │
│  │     Content-Addressed Storage     │  │
│  │  ╭─────────────────────────────╮  │  │
│  │  │    RFC-020 (this document)  │  │  │
│  │  │    describing CAS           │  │  │
│  │  │    stored in CAS            │  │  │
│  │  ╰─────────────────────────────╯  │  │
│  ╰───────────────────────────────────╯  │
╰─────────────────────────────────────────╯
```

Homoiconic storage: the description is the thing.

Tombstones

Objects are never truly deleted - they are marked with tombstones.

Soft Deletion

```
(define (cas-tombstone hash #!key reason actor)
  "Mark object as deleted without removing it"
  (let ((tombstone
         `(tombstone
           (object ,hash)
           (deleted-at ,(current-seconds))
           (deleted-by ,actor)
           (reason ,reason))))
    (audit-append action: `(tombstone ,hash) motivation: reason)
    (cas-put (serialize tombstone))))
```

Tombstone Properties

```
(soup-object
  (name "sha256:deadbeef...")
  (type tombstone)
  (size "0B")  ; Logical size is zero
  (status deleted)
  (reason "Superseded by sha256:newversion...")
  (recoverable #t))
```

Recovery

```
(define (cas-resurrect hash)
  "Remove tombstone, restore object visibility"
  (let ((tombstone (find-tombstone hash)))
    (when tombstone
      (audit-append action: `(resurrect ,hash))
      (cas-delete (tombstone-hash tombstone)))))
```

Garbage Collection with Tombstones

```
(define (cas-gc roots #!key preserve-tombstones)
  "GC respects tombstones by default"
  (let ((reachable (trace-reachable roots))
        (tombstoned (all-tombstoned-hashes)))
    (for-each
      (lambda (hash)
        (unless (or (set-member? reachable hash)
                    (and preserve-tombstones
                         (set-member? tombstoned hash)))
          (delete-file (hash->path hash))))
      (all-objects))))
```

Pinning

Pinned objects are protected from garbage collection.

Pin Operations

```
(define (cas-pin hash #!key recursive reason)
  "Pin object (and optionally its references)"
  (let ((pin `(pin
               (object ,hash)
               (pinned-at ,(current-seconds))
               (recursive ,recursive)
               (reason ,reason))))
    (write-file (pin-path hash) (serialize pin))
    (when recursive
      (for-each (lambda (ref) (cas-pin ref recursive: #t))
                (object-references hash)))))

(define (cas-unpin hash)
  "Remove pin, allow GC"
  (delete-file (pin-path hash)))

(define (cas-pinned? hash)
  "Check if object is pinned"
  (file-exists? (pin-path hash)))
```

Pin Manifest

```
;; All pins as soup objects
(soup-query type: 'pin)

;; Pin statistics
(soup-pin-count)      ; => 142 objects pinned
(soup-pinned-size)    ; => 23.4MB
```

Root Pins

Certain objects are implicitly pinned:

```
(define implicit-roots
  '("HEAD"          ; Current commit
    "refs/tags/*"    ; All tags
    "bootstrap"      ; System bootstrap
    "schema/*"))     ; All schemas
```

Bloom Filters

Compact probabilistic set membership for efficient replication.

Structure

```
(define (make-bloom-filter capacity error-rate)
  "Create bloom filter for approximate membership"
  (let* ((bits (bloom-optimal-bits capacity error-rate))
         (hashes (bloom-optimal-hashes capacity bits)))
    (bloom-filter
      (bit-vector (make-vector bits 0))
      (hash-count hashes)
      (capacity capacity)
      (error-rate error-rate))))
```

Operations

```
(define (bloom-add! filter hash)
  "Add hash to bloom filter"
  (for-each
    (lambda (i)
      (bit-vector-set! (bloom-bits filter) (bloom-index filter hash i) 1))
    (iota (bloom-hash-count filter))))

(define (bloom-contains? filter hash)
  "Check if hash MIGHT be in filter (false positives possible)"
  (every
    (lambda (i)
      (= 1 (bit-vector-ref (bloom-bits filter) (bloom-index filter hash i))))
    (iota (bloom-hash-count filter))))
```

Replication Protocol

```
;; Sender: "Here's what I have"
(define (bloom-inventory)
  (let ((filter (make-bloom-filter 10000 0.01)))
    (for-each (lambda (hash) (bloom-add! filter hash))
              (all-object-hashes))
    filter))

;; Receiver: "Send me what I'm missing"
(define (bloom-diff local-filter remote-filter)
  (filter (lambda (hash)
            (and (bloom-contains? remote-filter hash)
                 (not (cas-exists? hash))))
          (all-object-hashes)))

;; Exchange is O(bloom-size) not O(object-count)
```

Soup Integration

```
;; Bloom filter itself is a soup object
(soup-object
  (name "bloom/2026-01-07")
  (type bloom-filter)
  (size "12KB")
  (crypto (sha256 "bloom-hash..."))
  (capacity 10000)
  (error-rate 0.01)
  (population 4723))
```

Content Types

Typed objects with schema validation.

Type Registry

```
(define content-types
  '((blob        . "application/octet-stream")
    (text        . "text/plain")
    (markdown    . "text/markdown")
    (scheme      . "text/x-scheme")
    (sexp        . "application/x-sexp")
    (json        . "application/json")
    (pdf         . "application/pdf")
    (archive     . "application/x-sealed-archive")
    (key         . "application/x-spki-key")
    (certificate . "application/x-spki-cert")))
```

Typed Storage

```
(define (cas-put/typed content type #!key schema)
  "Store with type metadata"
  (let* ((hash (sha256 content))
         (meta `(typed-object
                 (hash ,hash)
                 (type ,type)
                 (size ,(blob-length content))
                 (schema ,schema))))
    (cas-put content)
    (cas-put (serialize meta))
    hash))
```

Schema Validation

```
(define (cas-validate hash)
  "Validate object against its schema"
  (let* ((meta (cas-get-meta hash))
         (schema-hash (typed-object-schema meta))
         (schema (cas-get schema-hash))
         (content (cas-get hash)))
    (schema-validate schema content)))

;; Schema is itself content-addressed
(soup-object
  (name "schema/soup-object")
  (type schema)
  (validates soup-object)
  (crypto (sha256 "schema-hash...")))
```

MIME Detection

```
(define (cas-detect-type content)
  "Detect content type from magic bytes"
  (cond
    ((pdf-magic? content) 'pdf)
    ((gzip-magic? content) 'archive)
    ((utf8-text? content)
     (cond
       ((sexp-syntax? content) 'sexp)
       ((markdown-syntax? content) 'markdown)
       (else 'text)))
    (else 'blob)))
```

Object Capabilities

Content addresses as capabilities: if you know the hash, you can retrieve it.

Hash as Capability

```
;; Knowledge of hash grants read access
(define (cas-get-if-known hash)
  "Capability-based retrieval"
  (if (valid-hash? hash)
      (cas-get hash)
      (error "Invalid capability")))

;; Hashes are unguessable (256-bit entropy)
;; Sharing a hash = sharing read access
```

Capability Attenuation

```
;; SPKI certificate granting access to specific content
(spki-cert
  (issuer vault-key)
  (subject reader-key)
  (permission (read (hash sha256 "specific-content...")))
  (validity (not-after "2027-01-01")))

;; Grant access to a subtree
(spki-cert
  (issuer vault-key)
  (subject reader-key)
  (permission (read (tree sha256 "subtree-root...")))
  (propagate #t))  ; Access to all referenced objects
```

Sealed Capabilities

```
;; Encrypted capability - only holder of key can use
(define (seal-capability hash recipient-key)
  (let ((cap `(capability
               (object ,hash)
               (granted-at ,(current-seconds)))))
```

```
    (age-encrypt (serialize cap) recipient-key)))

;; Recipient decrypts to obtain hash
(define (unseal-capability sealed-cap identity)
  (let ((cap (deserialize (age-decrypt sealed-cap identity))))
    (capability-object cap)))
```

Capability Revocation

```
;; Revocation via tombstone
(define (revoke-capability hash)
  (cas-tombstone hash reason: "Capability revoked"))

;; Or via SPKI CRL
(spki-crl
  (issuer vault-key)
  (revoked
    ((hash sha256 "revoked-content...")
     (reason "Superseded")
     (revoked-at 1767700000))))
```

Hash Migration

When cryptographic algorithms weaken, the system must migrate.

Multihash

```
;; Self-describing hash format
(define (multihash algo data)
  (let ((hash (case algo
                ((sha256) (sha256 data))
                ((sha384) (sha384 data))
                ((sha512) (sha512 data))
                ((blake3) (blake3 data)))))
    (list algo (blob-length hash) hash)))

;; Parse multihash
(define (multihash-algorithm mh) (car mh))
(define (multihash-digest mh) (caddr mh))
```

Dual-Hash Period

```
;; During migration, store under both hashes
(define (cas-put/migrate content old-algo new-algo)
  (let ((old-hash (hash-with old-algo content))
        (new-hash (hash-with new-algo content)))
    (cas-put-raw old-hash content)
    (cas-put-raw new-hash content)
    ;; Link old to new
    (ref-set (string-append "migrate/" old-hash) new-hash)
    new-hash))

;; Lookup follows migration links
(define (cas-get/migrate hash)
  (let ((migrated (ref-get (string-append "migrate/" hash))))
    (cas-get (or migrated hash))))
```

Migration Manifest

```
(migration-manifest
  (from-algorithm sha256)
  (to-algorithm sha384)
  (started-at 1767700000)
  (status in-progress)
  (migrated 4723)
  (remaining 1892)
  (mappings
    (("sha256:abc..." . "sha384:def...")
     ("sha256:123..." . "sha384:456...")
     ...)))
```

Verification During Migration

```
(define (verify-migration hash)
  "Verify object under both old and new hash"
  (let* ((content (cas-get hash))
         (old-hash (sha256 content))
         (new-hash (sha384 content)))
    (and (or (equal? hash old-hash)
             (equal? hash new-hash))
         (equal? (ref-get (string-append "migrate/" old-hash))
                 new-hash))))
```

Comparison with Related Systems

System	Hash	Chunking	Naming
Git	SHA-1	Pack files	refs/branches
IPFS	SHA-256/multihash	Rabin	IPNS, DNSLink
Perkeep	SHA-224	Rolling hash	Permanodes

System	Hash	Chunking	Naming
Library CAS	SHA-256	Configurable	Vault refs

Security Considerations

Hash Collision Attacks

SHA-256 provides 128-bit collision resistance. No practical attacks known.

If SHA-256 is ever broken:

```
;; Multihash for algorithm agility
(define (multihash algo data)
  (case algo
    ((sha256) (cons 'sha256 (sha256 data)))
    ((sha384) (cons 'sha384 (sha384 data)))
    ((blake3) (cons 'blake3 (blake3 data)))))
```

Length Extension Attacks

SHA-256 is vulnerable to length extension. For authentication, use HMAC:

```
(define (cas-mac key hash)
  (hmac-sha256 key hash))
```

Timing Attacks

Hash comparison MUST be constant-time:

```
(define (hash-equal? a b)
  (let ((result 0))
    (do ((i 0 (+ i 1)))
        ((= i 32) (zero? result))
      (set! result (bitwise-ior result
                     (bitwise-xor (blob-ref a i) (blob-ref b i)))))))
```

Performance Considerations

Caching

```
;; LRU cache for hot objects
(define cas-cache (make-lru-cache 1000))

(define (cas-get/cached hash)
  (or (lru-get cas-cache hash)
      (let ((content (cas-get hash)))
        (lru-put! cas-cache hash content)
        content)))
```

Parallel Hashing

Large files benefit from parallel hashing of chunks.

SSD Optimization

Random access pattern. Use write-ahead log for durability:

```
(define (cas-put/durable content)
  (let ((hash (sha256 content)))
    (wal-append hash content)
    (write-blob (hash->path hash) content)
    (wal-commit hash)
    hash))
```

Implementation Notes

Dependencies

Component	Library
SHA-256	message-digest, sha2
Blob I/O	CHICKEN I/O
Chunking	Custom

Error Handling

```
(define-condition-type &cas-error &error
  cas-error?
  (hash cas-error-hash))

(define-condition-type &cas-not-found &cas-error
  cas-not-found?)

(define-condition-type &cas-corrupt &cas-error
  cas-corrupt?)
```

References

1. Merkle, R. (1987). A Digital Signature Based on a Conventional Encryption Function

2. Git Internals - Git Objects
3. IPFS Content Addressing
4. RFC-006: Vault System Architecture
5. RFC-018: Sealed Archive Format

Changelog

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

Implementation Status: Draft **Dependencies:** sha2, message-digest **Integration:** Vault, Replication, SPKI