RFC-024: Network Protocol

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

Abstract

This RFC specifies the network protocol for the Library of Cyberspace: how vaults discover each other, authenticate, exchange objects, and synchronize state. The protocol is transport-agnostic, capability-authenticated, and optimized for content-addressed data.

Motivation

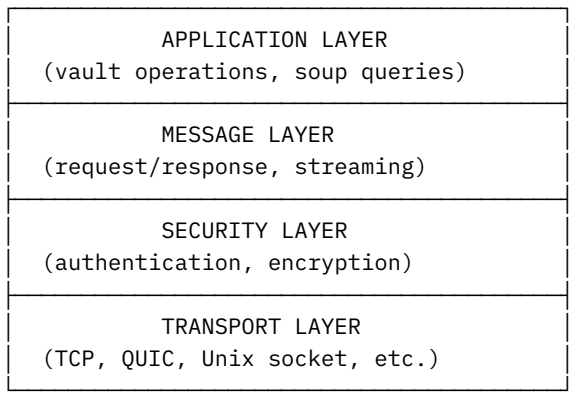
Vaults must communicate to:

- **Replicate** - Distribute objects across vaults
- **Federate** - Link independent vaults into networks
- **Subscribe** - Receive updates from publishers
- **Query** - Search the distributed soup

The protocol must handle:

- **Intermittent connectivity** - Vaults may be offline
- **Untrusted networks** - All communication authenticated
- **Large objects** - Efficient chunked transfer
- **Partial sync** - Resume interrupted transfers

Protocol Layers



Transport Layer

Supported Transports

```
(define transports
  '((tcp "Traditional TCP/IP")
    (quic "UDP-based, multiplexed")
    (unix "Local Unix domain socket")
    (tor "Onion-routed TCP")
    (i2p "Garlic-routed")
    (bluetooth "Local mesh")
    (sneakernet "Physical media exchange")))
```

Connection Establishment

```
(define (connect-vault address transport)
  "Establish connection to remote vault"
  (let* ((socket (transport-connect transport address))
         (connection (make-connection socket transport)))

    ;; Perform handshake
    (connection-handshake connection)

    ;; Return authenticated connection
    connection))
```

Multiplexing

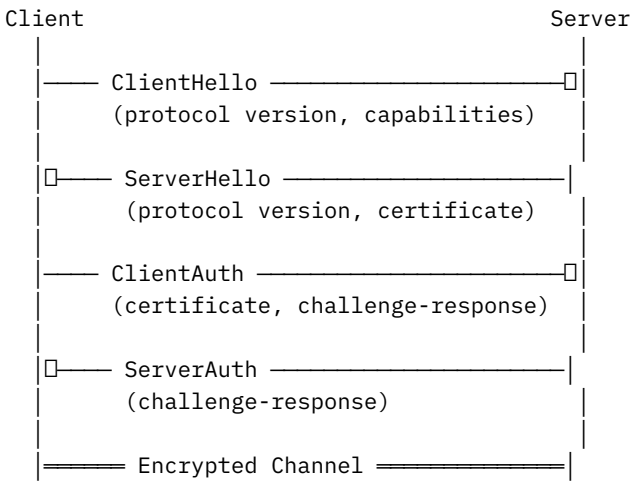
Multiple streams over single connection:

```
(define (open-stream connection stream-type)
  "Open multiplexed stream within connection"
  (let ((stream-id (allocate-stream-id connection)))
    (send-frame connection
      (frame type: 'stream-open
             stream-id: stream-id
             stream-type: stream-type))
    (make-stream connection stream-id stream-type)))
```

Security Layer

Handshake Protocol

Mutual authentication using SPKI certificates:



Handshake Implementation

```
(define (connection-handshake connection)
  "Perform mutual authentication handshake"

  ;; Send ClientHello
  (send-message connection
    `(client-hello
      (version ,protocol-version)
      (capabilities ,(local-capabilities))))

  ;; Receive ServerHello
  (let ((server-hello (receive-message connection)))
    (verify-protocol-version (server-hello-version server-hello))

    ;; Verify server certificate
    (let ((server-cert (server-hello-certificate server-hello)))
      (unless (verify-certificate server-cert)
        (error "Server certificate invalid"))

      ;; Generate session key using X25519
      (let* ((ephemeral-keypair (x25519-keypair))
             (shared-secret (x25519-shared (keypair-secret ephemeral-keypair)
                                           (cert-public-key server-cert))))

        ;; Send ClientAuth
        (send-message connection
          `(client-auth
            (certificate ,(local-certificate))
            (ephemeral-public ,(keypair-public ephemeral-keypair))
```

```
            (signature ,(sign-challenge server-hello))))

        ;; Receive ServerAuth
        (let ((server-auth (receive-message connection)))
          (unless (verify-server-signature server-auth server-cert)
            (error "Server authentication failed"))

          ;; Derive session keys
          (derive-session-keys connection shared-secret))))))
```

Session Encryption

```
(define (derive-session-keys connection shared-secret)
  "Derive symmetric keys for session"
  (let* ((key-material (hkdf shared-secret
                             info: "library-session"
                             length: 64))
         (client-key (subbytes key-material 0 32))
         (server-key (subbytes key-material 32 64)))

    (set-connection-keys! connection
      (if (connection-is-client? connection)
          (keys send: client-key receive: server-key)
          (keys send: server-key receive: client-key)))))

(define (encrypt-message connection message)
  "Encrypt message with session key"
  (let ((nonce (connection-next-nonce connection)))
    (chacha20-poly1305-encrypt
      (connection-send-key connection)
      nonce
      (serialize message))))

(define (decrypt-message connection ciphertext)
  "Decrypt message with session key"
  (let ((nonce (connection-next-nonce connection)))
    (deserialize
      (chacha20-poly1305-decrypt
        (connection-receive-key connection)
        nonce
        ciphertext))))
```

Message Layer

Message Format

```
(library-message
  (version 1)
  (type request|response|stream|error)
  (id <message-id>)
  (in-reply-to <message-id>|#f)
  (capability <spki-cert>|#f)
  (body <payload>))
```

Request Types

```
(define request-types
  '(;; Object operations
    (get-object hash)
    (put-object hash data)
    (has-object hash)

    ;; Bulk operations
    (get-objects hashes)
    (get-tree root-hash)
    (sync-objects bloom-filter)

    ;; Soup queries
    (soup-query query)
    (soup-subscribe query)

    ;; Vault operations
    (get-refs pattern)
    (get-head)
    (get-releases)

    ;; Discovery
    (get-capabilities)
    (get-peers)))
```

Request/Response

```
(define (vault-request connection request #!key capability timeout)
  "Send request, wait for response"
  (let ((message-id (generate-message-id)))

    ;; Send request
    (send-message connection
      (library-message
```

```
        version: 1
        type: 'request
        id: message-id
        capability: capability
        body: request))

    ;; Wait for response
    (let ((response (receive-response connection message-id timeout)))
      (if (eq? (message-type response) 'error)
          (error (message-body response))
          (message-body response)))))
```

Streaming

For large transfers:

```
(define (stream-object connection hash)
  "Stream large object in chunks"
  (let ((stream (open-stream connection 'object-transfer)))

    ;; Send chunks
    (let ((data (cas-get hash)))
      (let loop ((offset 0))
        (when (< offset (blob-length data))
          (let ((chunk (blob-copy data offset chunk-size)))
            (stream-send stream chunk)
            (loop (+ offset chunk-size))))))

      ;; End stream
      (stream-close stream))))

(define (receive-stream connection stream callback)
  "Receive streamed data"
  (let loop ((chunks '()))
    (let ((chunk (stream-receive stream)))
      (if (stream-ended? stream)
          (apply bytes-append (reverse chunks))
          (begin
            (callback chunk)
            (loop (cons chunk chunks)))))))
```

Object Exchange

Get Object

```
;; Request
(get-object
  (hash "sha256:abc123..."))

;; Response (small object)
(object-data
  (hash "sha256:abc123...")
  (size 1024)
  (data #${...}))

;; Response (large object - streaming)
(object-stream
  (hash "sha256:abc123...")
  (size 10485760)
  (stream-id 42))
```

Put Object

```
;; Request
(put-object
  (hash "sha256:abc123...")
  (data #${...}))

;; Response
(put-result
  (hash "sha256:abc123...")
  (status stored|duplicate|rejected)
  (reason #f|"quota exceeded"))
```

Batch Operations

```
;; Request multiple objects
(get-objects
  (hashes ("sha256:aaa..." "sha256:bbb..." "sha256:ccc...")))

;; Response
(objects-batch
  (objects
    ((hash "sha256:aaa..." data #${...})
     (hash "sha256:bbb..." data #${...})
     (hash "sha256:ccc..." status: missing))))
```

Tree Transfer

Efficient Merkle tree sync:

```
;; Request tree
(get-tree
  (root "sha256:root...")
  (depth 3)  ; How deep to fetch
  (exclude ("sha256:already-have...")))

;; Response
(tree-data
  (root "sha256:root...")
  (objects
    ((hash "sha256:root..." data #${...})
     (hash "sha256:child1..." data #${...})
     (hash "sha256:child2..." data #${...}))))
```

Synchronization

Bloom Filter Exchange

Efficient "what do you have?" queries:

```
;; Step 1: Exchange bloom filters
(define (sync-init local-vault remote-connection)
  (let ((local-bloom (vault-bloom-filter local-vault)))

    ;; Send our bloom filter
    (send-message remote-connection
      `(sync-bloom
        (filter ,local-bloom)
        (population ,(bloom-population local-bloom))))

    ;; Receive their bloom filter
    (let ((remote-bloom (receive-bloom remote-connection)))

      ;; Compute what they might want
      (let ((to-send (filter (lambda (hash)
                               (not (bloom-contains? remote-bloom hash)))
                             (vault-all-hashes local-vault))))
        to-send))))

;; Step 2: Exchange missing objects
(define (sync-exchange to-send to-receive connection)
  (parallel
```

```
    ;; Send what they need
    (for-each (lambda (hash)
                (stream-object connection hash))
              to-send)
    ;; Receive what we need
    (for-each (lambda (hash)
                (receive-object connection hash))
              to-receive)))
```

Incremental Sync

```
(define (incremental-sync local-vault remote-connection since)
  "Sync changes since timestamp or commit"

  ;; Get remote changes
  (let ((remote-changes (vault-request remote-connection
                          `(changes-since ,since))))

    ;; Get local changes
    (let ((local-changes (vault-changes-since local-vault since)))

      ;; Bidirectional merge
      (sync-merge local-vault remote-connection
                  local-changes remote-changes))))
```

Conflict Resolution

```
(define (sync-merge local-vault remote changes-local changes-remote)
  "Merge changes with conflict resolution"

  (for-each
    (lambda (hash)
      (cond
        ;; Only local has it - push
        ((and (member hash changes-local)
              (not (member hash changes-remote)))
         (push-object local-vault remote hash))

        ;; Only remote has it - pull
        ((and (member hash changes-remote)
              (not (member hash changes-local)))
         (pull-object local-vault remote hash))

        ;; Both have it - verify same
        ((and (member hash changes-local)
              (member hash changes-remote))
```

```
        ;; Content-addressed, so same hash = same content
        #t)))
    (union changes-local changes-remote)))
```

Discovery

Peer Discovery

```
;; Well-known peers (bootstrap)
(define (discover-bootstrap)
  (map parse-address
       (read-file "/etc/library/bootstrap-peers")))

;; mDNS/DNS-SD for local network
(define (discover-local)
  (mdns-browse "_library._tcp.local"))

;; DHT for global discovery
(define (discover-dht target-hash)
  (dht-find-providers target-hash))

;; Peer exchange
(define (discover-pex connection)
  (vault-request connection '(get-peers)))
```

Capability Advertisement

```
;; Query vault capabilities
(get-capabilities)

;; Response
(vault-capabilities
  (protocol-version 1)
  (features (bloom-sync tree-transfer streaming soup-query))
  (max-object-size 104857600)  ; 100MB
  (storage-available 10737418240)  ; 10GB
  (public-key "ed25519:abc...")
  (services
    ((soup-query rate-limit: 100/minute)
     (object-transfer rate-limit: 10MB/second))))
```

Soup Queries

Remote Query

```
;; Query remote soup
(soup-query
  (type certificate)
  (issuer "vault-admin")
  (limit 100))

;; Response
(soup-results
  (count 47)
  (objects
    ((name "cert/alice" type certificate ...)
     (name "cert/bob" type certificate ...)
     ...)))
```

Subscription

```
;; Subscribe to soup changes
(soup-subscribe
  (query (type release))
  (callback-url "library://my-vault/callbacks/releases"))

;; Subscription confirmation
(subscription
  (id "sub-123")
  (query (type release))
  (expires "2026-02-07T00:00:00Z"))

;; Push notification (when matching object appears)
(soup-notification
  (subscription-id "sub-123")
  (object (name "release/1.2.0" type release ...)))
```

Error Handling

Error Types

```
(define error-types
  '((not-found "Object not found")
    (unauthorized "Capability required")
    (forbidden "Capability insufficient")
    (quota-exceeded "Storage quota exceeded")
    (rate-limited "Too many requests")
```

```
    (timeout "Operation timed out")
    (invalid-request "Malformed request")
    (internal-error "Server error")
    (unavailable "Service temporarily unavailable")))
```

Error Response

```
(library-message
  (type error)
  (id <message-id>)
  (in-reply-to <request-id>)
  (body
    (error-response
      (code not-found)
      (message "Object sha256:abc... not found")
      (retry-after #f))))
```

Retry Logic

```
(define (vault-request-with-retry connection request #!key max-retries)
  "Request with exponential backoff retry"
  (let loop ((attempt 0) (delay 1000))
    (let ((result (vault-request connection request)))
      (if (and (error-response? result)
               (retryable-error? result)
               (< attempt max-retries))
          (begin
            (sleep delay)
            (loop (+ attempt 1) (* delay 2)))
          result))))
```

Security Considerations

Capability Verification

```
(define (verify-request-capability request connection)
  "Verify request has sufficient capability"
  (let ((required-cap (request->capability request))
        (presented-cap (message-capability request)))

    (unless presented-cap
      (error 'unauthorized "Capability required"))

    (unless (valid-certificate? presented-cap)
      (error 'unauthorized "Invalid capability certificate"))
```

```
    (unless (capability-grants? presented-cap required-cap)
      (error 'forbidden "Insufficient capability"))

    ;; Verify capability chain back to trusted root
    (unless (verify-capability-chain presented-cap)
      (error 'forbidden "Capability chain invalid"))))
```

Denial of Service Protection

```
(define (rate-limit connection request-type)
  "Apply rate limiting"
  (let ((limit (get-rate-limit request-type))
        (current (connection-request-count connection request-type)))
    (when (> current limit)
      (error 'rate-limited
             (format "Rate limit exceeded: ~a/~a" current limit)))))

(define (size-limit request)
  "Enforce size limits"
  (when (> (request-size request) max-request-size)
    (error 'invalid-request "Request too large")))
```

Replay Prevention

```
(define (verify-nonce connection message)
  "Prevent replay attacks"
  (let ((nonce (message-nonce message)))
    (when (nonce-seen? connection nonce)
      (error 'invalid-request "Replay detected"))
    (record-nonce connection nonce)))
```

Implementation Notes

Wire Format

Messages serialized as canonical S-expressions:

```
(define (serialize-message message)
  "Canonical S-expression serialization"
  (canonical-sexp->bytes message))

(define (deserialize-message bytes)
  "Parse S-expression"
  (bytes->canonical-sexp bytes))
```

Compression

Optional compression for large payloads:

```
(define (maybe-compress data)
  (if (> (blob-length data) compression-threshold)
      (values (zstd-compress data) 'zstd)
      (values data 'none)))
```

Connection Pooling

```
(define connection-pool (make-hash-table))

(define (get-connection address)
  "Get pooled connection or create new"
  (or (hash-table-ref connection-pool address #f)
      (let ((conn (connect-vault address)))
        (hash-table-set! connection-pool address conn)
        conn)))
```

References

1. QUIC Protocol - RFC 9000
2. Noise Protocol Framework
3. RFC-020: Content-Addressed Storage
4. RFC-021: Capability Delegation
5. IPFS Bitswap Protocol
6. libp2p Specifications

Changelog

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

Implementation Status: Draft **Dependencies:** transport, crypto, cas **Integration:** Replication, federation, distributed soup