RFC-032: Rate Limiting and Quotas

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

Abstract

This RFC specifies rate limiting and quotas for the Library of Cyberspace: how vaults protect themselves from abuse while ensuring fair access for legitimate users. Limits are capability-aware and auditable.

Motivation

Shared resources require protection:

- **Denial of Service** - Malicious overload
- **Resource exhaustion** - Runaway processes
- **Fair sharing** - Equal access for all users
- **Cost control** - Prevent unbounded consumption

Limits must be:

- **Configurable** - Different limits for different principals
- **Graceful** - Degrade smoothly, don't cliff
- **Transparent** - Users know their limits
- **Auditable** - All limiting decisions logged

Rate Limiting

Token Bucket Algorithm

```
(define (make-token-bucket capacity fill-rate)
  "Create token bucket rate limiter"
  (let ((tokens capacity)
        (last-fill (current-time-ms)))

    (lambda (cost)
      ;; Refill tokens based on elapsed time
      (let* ((now (current-time-ms))
             (elapsed (- now last-fill))
             (new-tokens (+ tokens (* fill-rate (/ elapsed 1000)))))
        (set! tokens (min capacity new-tokens))
        (set! last-fill now)
```

```
        ;; Try to consume tokens
        (if (>= tokens cost)
            (begin
              (set! tokens (- tokens cost))
              #t)  ; Allowed
            #f))))) ; Denied

;; Example: 100 requests per second, burst of 200
(define request-limiter (make-token-bucket 200 100))
```

Sliding Window

```
(define (make-sliding-window size interval)
  "Sliding window rate limiter"
  (let ((events (make-deque)))

    (lambda ()
      ;; Remove old events
      (let ((cutoff (- (current-time-ms) interval)))
        (while (and (not (deque-empty? events))
                    (< (deque-front events) cutoff))
          (deque-pop-front! events)))

      ;; Check if under limit
      (if (< (deque-length events) size)
          (begin
            (deque-push-back! events (current-time-ms))
            #t)
          #f))))
```

Leaky Bucket

```
(define (make-leaky-bucket capacity leak-rate)
  "Leaky bucket for smoothing bursts"
  (let ((level 0)
        (last-leak (current-time-ms)))

    (lambda (amount)
      ;; Leak based on elapsed time
      (let* ((now (current-time-ms))
             (elapsed (- now last-leak))
             (leaked (* leak-rate (/ elapsed 1000))))
        (set! level (max 0 (- level leaked)))
        (set! last-leak now)

        ;; Try to add to bucket
```

```
        (if (<= (+ level amount) capacity)
            (begin
              (set! level (+ level amount))
              #t)
            #f)))))
```

Quota System

Quota Types

```
(define quota-types
  '((storage . ((unit . bytes)
                (period . #f)          ; No reset
                (default . 10737418240))) ; 10GB

    (bandwidth . ((unit . bytes)
                  (period . monthly)
                  (default . 107374182400))) ; 100GB/month

    (requests . ((unit . count)
                 (period . daily)
                 (default . 100000)))    ; 100k/day

    (objects . ((unit . count)
                (period . #f)
                (default . 1000000)))))  ; 1M objects
```

Quota Tracking

```
(define principal-quotas (make-hash-table))

(define (get-quota principal quota-type)
  "Get principal's quota for type"
  (let ((quotas (hash-table-ref principal-quotas principal '())))
    (or (assoc-ref quotas quota-type)
        (assoc-ref quota-types quota-type 'default))))

(define (get-usage principal quota-type)
  "Get principal's current usage"
  (let ((period (quota-period quota-type)))
    (soup-aggregate
      where: (and (type: 'usage-record)
                  (principal: principal)
                  (quota-type: quota-type)
                  (if period
```

```
                    (timestamp: (> (period-start period)))
                    #t))
      aggregate: (sum 'amount))))

(define (check-quota principal quota-type amount)
  "Check if operation would exceed quota"
  (let ((quota (get-quota principal quota-type))
        (usage (get-usage principal quota-type)))
    (<= (+ usage amount) quota)))
```

Quota Enforcement

```
(define (enforce-quota principal quota-type amount)
  "Enforce quota, raise error if exceeded"
  (unless (check-quota principal quota-type amount)
    (let ((quota (get-quota principal quota-type))
          (usage (get-usage principal quota-type)))
      (audit-append action: 'quota-exceeded
                    principal: principal
                    quota-type: quota-type
                    quota: quota
                    usage: usage
                    requested: amount)
      (error 'quota-exceeded
             `((type . ,quota-type)
               (quota . ,quota)
               (usage . ,usage)
               (requested . ,amount))))))

(define (record-usage principal quota-type amount)
  "Record quota usage"
  (soup-put
    `((type . usage-record)
      (principal . ,principal)
      (quota-type . ,quota-type)
      (amount . ,amount)
      (timestamp . ,(current-time)))))
```

Principal-Based Limits

Limit Tiers

```
(define limit-tiers
  '((anonymous . ((requests-per-second . 10)
                  (storage-bytes . 0)
```

```
                (bandwidth-bytes . 1073741824)))

    (authenticated . ((requests-per-second . 100)
                      (storage-bytes . 10737418240)
                      (bandwidth-bytes . 107374182400)))

    (premium . ((requests-per-second . 1000)
                (storage-bytes . 1099511627776)
                (bandwidth-bytes . 10995116277760)))

    (admin . ((requests-per-second . #f)  ; Unlimited
              (storage-bytes . #f)
              (bandwidth-bytes . #f)))))

(define (principal-tier principal)
  "Determine principal's limit tier"
  (cond
    ((admin-principal? principal) 'admin)
    ((premium-principal? principal) 'premium)
    ((authenticated? principal) 'authenticated)
    (else 'anonymous)))
```

Per-Principal Limiters

```
(define principal-limiters (make-hash-table))

(define (get-limiter principal)
  "Get or create rate limiter for principal"
  (or (hash-table-ref principal-limiters principal #f)
      (let* ((tier (principal-tier principal))
             (rps (assoc-ref (assoc-ref limit-tiers tier)
                             'requests-per-second)))
        (if rps
            (let ((limiter (make-token-bucket (* rps 2) rps)))
              (hash-table-set! principal-limiters principal limiter)
              limiter)
            (lambda (_) #t)))))  ; Unlimited

(define (rate-limit-principal principal)
  "Apply rate limit to principal"
  (let ((limiter (get-limiter principal)))
    (unless (limiter 1)
      (audit-append action: 'rate-limited principal: principal)
      (error 'rate-limited))))
```

Capability-Based Limits

```
;; Capabilities can include rate limits
(spki-cert
  (issuer vault-admin)
  (subject api-client)
  (capability
    (action read)
    (object vault-content)
    (rate-limit
      (requests-per-second 50)
      (burst 100)))
  (validity (not-after "2027-01-01")))

(define (capability-rate-limit cap)
  "Extract rate limit from capability"
  (assoc-ref cap 'rate-limit))
```

Resource-Specific Limits

Storage Limits

```
(define (enforce-storage-limit principal size)
  "Enforce storage quota before write"
  (let* ((quota (get-quota principal 'storage))
         (used (principal-storage-used principal)))
    (when (and quota (> (+ used size) quota))
      (error 'storage-quota-exceeded
             `((quota . ,quota)
               (used . ,used)
               (requested . ,size))))))

(define (principal-storage-used principal)
  "Calculate principal's storage usage"
  (soup-aggregate
    where: (owner: principal)
    aggregate: (sum 'size)))
```

Bandwidth Limits

```
(define (enforce-bandwidth-limit principal bytes direction)
  "Enforce bandwidth quota"
  (let* ((quota (get-quota principal 'bandwidth))
         (used (principal-bandwidth-used principal)))
    (when (and quota (> (+ used bytes) quota))
```

```
      (error 'bandwidth-quota-exceeded
             `((quota . ,quota)
               (used . ,used)
               (requested . ,bytes)
               (direction . ,direction))))))
```

Connection Limits

```
(define max-connections-per-principal 100)
(define principal-connections (make-hash-table))

(define (enforce-connection-limit principal)
  "Limit concurrent connections"
  (let ((count (hash-table-ref principal-connections principal 0)))
    (when (>= count max-connections-per-principal)
      (error 'connection-limit-exceeded
             `((limit . ,max-connections-per-principal)
               (current . ,count))))))

(define (track-connection principal direction)
  (let ((count (hash-table-ref principal-connections principal 0)))
    (hash-table-set! principal-connections principal
      (case direction
        ((open) (+ count 1))
        ((close) (max 0 (- count 1)))))))
```

Backpressure

Request Queuing

```
(define request-queue (make-bounded-queue 1000))

(define (queue-request request)
  "Queue request with backpressure"
  (if (queue-full? request-queue)
      (begin
        (audit-append action: 'request-rejected reason: 'queue-full)
        (error 'service-unavailable))
      (queue-push! request-queue request)))
```

Load Shedding

```
(define (should-shed-load?)
  "Determine if load shedding needed"
  (or (> (cpu-usage) 0.9)
```

```
      (> (memory-usage) 0.9)
      (> (queue-length request-queue) 800)))

(define (shed-load request)
  "Selectively reject requests under load"
  (cond
    ;; Always allow admin requests
    ((admin-principal? (request-principal request))
     (process-request request))
    ;; Shed low-priority requests
    ((and (should-shed-load?)
          (low-priority-request? request))
     (audit-append action: 'load-shed request: (request-id request))
     (error 'service-unavailable))
    (else
     (process-request request))))
```

Retry-After

```
(define (rate-limit-response retry-after)
  "Generate rate limit response"
  `((status . 429)
    (headers . ((Retry-After . ,retry-after)
                (X-RateLimit-Limit . ,(current-limit))
                (X-RateLimit-Remaining . ,(remaining-tokens))
                (X-RateLimit-Reset . ,(reset-time))))))
```

Monitoring

Limit Metrics

```
(define (record-limit-metrics)
  "Record rate limiting metrics"
  (metric-set! 'ratelimit.requests.allowed allowed-count)
  (metric-set! 'ratelimit.requests.denied denied-count)
  (metric-set! 'ratelimit.requests.queued (queue-length request-queue))
  (metric-set! 'quota.storage.used total-storage-used)
  (metric-set! 'quota.storage.available total-storage-available))
```

Limit Alerts

```
(define limit-alerts
  '((high-rejection-rate
     (condition (> (/ denied-count (+ allowed-count denied-count)) 0.1))
     (severity warning)
```

```
     (message "High rate limit rejection rate"))

    (quota-near-limit
     (condition (> (/ used quota) 0.9))
     (severity warning)
     (message "Principal approaching quota limit"))))
```

Configuration

Dynamic Limits

```
;; Limits can be adjusted at runtime
(define (set-limit principal limit-type value)
  "Set custom limit for principal"
  (soup-put
    `((type . principal-limit)
      (principal . ,principal)
      (limit-type . ,limit-type)
      (value . ,value)
      (set-by . ,(current-principal))
      (set-at . ,(current-time)))
    type: 'configuration))

(define (get-effective-limit principal limit-type)
  "Get principal's effective limit"
  (or (soup-query-one type: 'principal-limit
                      principal: principal
                      limit-type: limit-type)
      (tier-limit (principal-tier principal) limit-type)))
```

Limit Inheritance

```
;; Groups can have limits that members inherit
(define (group-limit group limit-type)
  (soup-query-one type: 'group-limit
                  group: group
                  limit-type: limit-type))

(define (effective-limit principal limit-type)
  "Resolve limit with inheritance"
  (or (get-effective-limit principal limit-type)
      (let ((groups (principal-groups principal)))
        (find (lambda (g) (group-limit g limit-type)) groups))
      (default-limit limit-type)))
```

Security

Limit Bypass Prevention

```
;; Prevent limit bypass via identity rotation
(define (track-ip-limits ip)
  "Track limits by IP in addition to principal"
  (let ((limiter (get-ip-limiter ip)))
    (unless (limiter 1)
      (error 'ip-rate-limited))))

;; Combine principal and IP limits
(define (combined-rate-limit principal ip)
  (and (rate-limit-principal principal)
       (track-ip-limits ip)))
```

Audit Trail

```
(define (audit-limit-event event-type principal details)
  (audit-append
    action: event-type
    principal: principal
    details: details
    timestamp: (current-time)))
```

References

1. Token Bucket Algorithm
2. Leaky Bucket Algorithm
3. RFC-021: Capability Delegation
4. RFC-031: Monitoring

Changelog

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

Implementation Status: Draft **Dependencies:** soup, spki, monitoring **Integration:** API layer, network protocol, resource management