15-7-25 | Distributed Database Architecture

"Cryptographic sovereignty meets sacred economics through append-only truth"

Core Design Philosophy

The Synchronicity Engine operates on **append-only event logs** that create an **eventually consistent** distributed database using OrbitDB. The architecture prioritizes **simplicity, fraud resistance, and sacred economics** over traditional cryptocurrency patterns.

Fundamental Principles

- **Attention is the only scarce resource** - everything else derives from focused intention
- **Append-only logs prevent manipulation** - history cannot be rewritten
- **Cryptographic proof of stewardship** - only current stewards can transfer artifacts
- **Subscription-based access** - prevents spam and funds sovereign infrastructure
- **Community reputation system** - good actors pay less, bad actors pay more
- **Zero transfer fees** - gratitude flows freely without friction

---

Core Event Log Architecture

1. Attention Switch Event Log (Per User)

The Foundation of All Value

```
interface AttentionSwitchEvent {
  eventId: string;                    // UUID for this specific event
  userId: string;                     // cryptographic identity of user
  index: number;                    // monotonically increasing natural number (0, 1, 2, 3...)
  intentionId: string;              // what intention received attention
  timestamp: number;                  // when attention switched (milliseconds since epoch)
  signature: string;                // cryptographic signature proving authenticity
  previousEventHash?: string;         // hash of previous event (blockchain-like integrity)
}
```

Key Properties:

- **Indexed by natural numbers** - enables duration calculation via `(timestamp[index+1] - timestamp[index])`
- **Append-only** - events cannot be deleted or modified
- **Cryptographically signed** - only the user can add to their own log
- **Chained integrity** - each event references the hash of the previous event

Duration Calculation:

```
function calculateAttentionDuration(
  userId: string,
  startIndex: number,
  endIndex?: number,
  currentTime = Date.now()
): number {
  const userLog = getUserAttentionLog(userId);
  const startEvent = userLog[startIndex];
  const endEvent = endIndex ? userLog[endIndex] : null;

  const startTime = startEvent.timestamp;
  const endTime = endEvent?.timestamp ?? currentTime;

  return endTime - startTime;
}
```

2. Stewardship Transfer Event Log (Per Artifact)

Cryptographic Proof of Ownership

```
interface StewardshipTransferEvent {
  eventId: string;
  artifactId: string;              // which artifact is being transferred
  fromSteward: string;             // previous steward's cryptographic identity
  toSteward: string;               // new steward's cryptographic identity
  transferType: "gift" | "offering_fulfillment" | "service_recognition";
  relatedEventId?: string;         // offering acceptance or proof of service
  timestamp: number;
  signature: string;               // signature from current steward (proves they
initiated)
  witnessSignatures?: string[];    // optional community verification
}
```

Key Properties:

- **Only current steward can transfer** - cryptographically enforced
- **Immutable ownership chain** - complete history of all transfers
- **Transfer reasons tracked** - enables reputation and abuse detection
- **Community witnessing** - optional multi-sig verification for high-value transfers

3. Bid Event Log (Global)

Temporary Asset Escrow

```
interface BidEvent {
  eventId: string;
  bidderId: string;                // who is making the bid
  offeringId: string;              // what they're bidding on
  artifactIds: string[];           // artifacts being offered as bid
  bidType: "token_basket" | "service_offer" | "artifact_access";
  escrowDuration: number;          // how long artifacts are locked (max 30
days)
  message?: string;                // optional personal message
  timestamp: number;
  signature: string;                  // bidder's signature
}
```

Key Properties:

- **Temporary escrow** - artifacts locked but not transferred until acceptance
- **Time-bounded** - bids automatically expire to prevent indefinite locking
- **Reversible** - bidder can withdraw before acceptance
- **Message channel** - enables personal connection in sacred economy

Core Data Structures

Artifacts (Immutable Records)

```
interface Artifact {
  artifactId: string;                 // permanent identifier
  artifactType: "token_of_gratitude" | "physical_resource" | "offering" | "intention";

  // For Tokens of Gratitude
  tokenData?: {
    forgedFrom: AttentionSpan[];      // which attention spans created this token
    totalDuration: number;            // calculated from attention spans
    dedicatedTo: string;              // intention this token honors
    forgedBy: string;                 // who created this token
    forgedAt: number;                 // timestamp of creation
  };

  // For Physical Resources
  resourceData?: {
    name: string;                     // "Solar Dehydrator" | "Permaculture Library"
    description: string;
    location?: GeographicPoint;
    accessType: "shared" | "lendable" | "gift";
    usageInstructions?: string;
  };
  // For Offerings
  offeringData?: {
    title: string;                // "Weekend Permaculture Workshop"
    description: string;
    timeWindow?: TimeWindow;
    location?: string;
    slotsAvailable: number;
    requirements?: string[];            // what bidders need to provide
  };

  // Universal properties
  currentSteward: string;               // who currently holds this artifact
  createdBy: string;                 // original creator
  createdAt: number;                   // creation timestamp
  status: "active" | "locked_in_bid" | "transferred" | "completed";
}

interface AttentionSpan {
  userId: string;
  intentionId: string;
  startIndex: number;                  // index in user's attention log
  endIndex?: number;                    // if null, span is still active
  quality?: number;                   // 1-10 rating of attention depth (optional)
}

interface TimeWindow {
  start: string;                   // ISO datetime
  end: string;                       // ISO datetime
  timezone: string;                  // "America/Los_Angeles"
}
```

Zero-Knowledge Reputation System

```
interface UserReputation {
  userId: string;
  reputationScore: number;                // 0-1000 (higher = more trusted)
  subscriptionTier: "bronze" | "silver" | "gold" | "platinum";
  monthlyDonationRequired: number;    // decreases with higher reputation

  // Zero-knowledge proofs of good behavior
  zkProofs: {
    attentionConsistency: string;     // proves attention logs are consistent
    transferHonesty: string;          // proves fair stewardship transfers
    communityContribution: string;    // proves service to collective
  };

  // Reputation factors (calculated from event logs)
  factors: {
    attentionQuality: number;         // depth and consistency of intention focus
    transferFairness: number;         // history of fair stewardship transfers
    communityService: number;         // offerings and gifts provided
    networkStability: number;         // uptime and sync reliability
  };

  lastUpdated: number;
}
```

Anti-Fraud Architecture

1. Subscription-Based Access Control

```
interface SubscriptionPayment {
  userId: string;
  amount: number;                     // in local currency or tokens
  paymentMethod: "fiat_donation" | "community_service" | "token_contribution";
  validUntil: number;                 // timestamp when subscription expires

  // Community service alternative
  serviceContribution?: {
    offeringsProvided: string[];      // offering IDs provided to community
    attentionContributed: number;     // total attention given to community intentions
    reputationBonus: number;         // reduction in required payment
  };

  signature: string;
}
```

Subscription Tiers & Pricing:

- **Bronze** (New Users): $25/month - basic access
- **Silver** (Good Reputation): $15/month - proven track record
- **Gold** (Community Leaders): $5/month - significant contributions
- **Platinum** (Stewards): $0/month - sustained service to collective

2. Cryptographic Integrity Checks

```
// Each event must pass cryptographic verification
function verifyEventIntegrity(event: AttentionSwitchEvent): boolean {
  // 1. Signature verification
  if (!verifySignature(event.signature, event.userId, eventHash(event))) {
    return false;
  }

  // 2. Chain integrity
  if (event.index > 0 && !event.previousEventHash) {
    return false;
  }

  // 3. Monotonic timestamp (within reasonable bounds)
  const previousEvent = getPreviousEvent(event.userId, event.index - 1);
  if (previousEvent && event.timestamp <= previousEvent.timestamp) {
    return false;
  }

  // 4. Reasonable time bounds (no events more than 24 hours apart)
  if (previousEvent && (event.timestamp - previousEvent.timestamp) > 86400000) {
    return false; // Likely attention switching fraud
  }
  return true;
}
```

3. Zero-Knowledge Reputation Proofs

```
// Users can prove good behavior without revealing private data
interface ZKProof {
  proofType: "attention_consistency" | "transfer_honesty" | "community_service";

  // Zero-knowledge proof that user's attention logs show consistent patterns
  attentionConsistencyProof?: {
    claim: "User has focused attention for >2 hours daily for 30 days";
    proof: string;                    // zk-SNARK proof
    publicInputs: {
      userId: string;
      timeRange: [number, number];
      threshold: number;
    };
  };

  // Proof of fair transfers without revealing specific amounts
  transferHonestyProof?: {
    claim: "User has completed >95% of stewardship transfers fairly";
    proof: string;
    publicInputs: {
      userId: string;
      transferCount: number;
      fairnessThreshold: number;
    };
  };

  // Proof of community contribution
  communityServiceProof?: {
    claim: "User has provided >X hours of community offerings";
    proof: string;
    publicInputs: {
      userId: string;
      serviceHours: number;
      timeRange: [number, number];
    };
  };
}
```

Offering & Bidding Flow

1. Creating an Offering

```
async function createOffering(
  stewardId: string,
  title: string,
  description: string,
  slotsAvailable: number,
  requirements?: string[]
): Promise<string> {
  const offeringId = generateId();

  const offeringArtifact: Artifact = {
    artifactId: offeringId,
    artifactType: "offering",
    offeringData: {
      title,
      description,
      slotsAvailable,
      requirements
    },
    currentSteward: stewardId,
    createdBy: stewardId,
    createdAt: Date.now(),
    status: "active"
  };

  await artifactsDB.put(offeringArtifact);
  return offeringId;
}
```

2. Placing a Bid

```
async function placeBid(
  bidderId: string,
  offeringId: string,
  artifactIds: string[],
  message?: string
): Promise<string> {
  // Verify bidder owns all artifacts being bid
  for (const artifactId of artifactIds) {
    const artifact = await artifactsDB.get(artifactId);
    if (artifact.currentSteward !== bidderId) {
      throw new Error(`Bidder does not own artifact ${artifactId}`);
    }
  }

  const bidEvent: BidEvent = {
    eventId: generateId(),
    bidderId,
    offeringId,
    artifactIds,
    bidType: "token_basket",
    escrowDuration: 86400000 * 7, // 7 days
    message,
    timestamp: Date.now(),
    signature: await signEvent(bidderId, bidEventHash)
  };

  // Lock artifacts in escrow
  for (const artifactId of artifactIds) {
    const artifact = await artifactsDB.get(artifactId);
    artifact.status = "locked_in_bid";
    await artifactsDB.put(artifact);
  }
  await bidEventsDB.add(bidEvent);
  return bidEvent.eventId;
}
```

3. Accepting Bids

```
async function acceptBid(
  stewardId: string,
  bidEventId: string
): Promise<void> {
  const bidEvent = await bidEventsDB.get(bidEventId);
  const offering = await artifactsDB.get(bidEvent.offeringId);

  // Verify steward owns the offering
  if (offering.currentSteward !== stewardId) {
    throw new Error("Only offering steward can accept bids");
  }

  // Transfer bid artifacts to offering steward
  for (const artifactId of bidEvent.artifactIds) {
    await transferStewardship(
      artifactId,
      bidEvent.bidderId,
      stewardId,
      "offering_fulfillment",
      bidEventId
    );
  }

  // Transfer offering to bidder
  await transferStewardship(
    bidEvent.offeringId,
    stewardId,
    bidEvent.bidderId,
    "offering_fulfillment",
    bidEventId
  );
  // Mark offering as completed
  offering.status = "completed";
  await artifactsDB.put(offering);
}
```

OrbitDB Configuration

Database Structure

```
const databases = {
  // Event logs (append-only)
  attentionSwitches: {
    type: "eventlog",
    accessController: "self", // only user can write to their own log
    indexBy: ["userId", "index"]
  },

  stewardshipTransfers: {
    type: "eventlog",
    accessController: "steward-only", // custom AC requiring steward signat
ure
    indexBy: ["artifactId", "timestamp"]
  },

  bidEvents: {
    type: "eventlog",
    accessController: "write-once", // immutable after creation
    indexBy: ["offeringId", "bidderId", "timestamp"]
  },

  subscriptionPayments: {
    type: "eventlog",
    accessController: "payment-gateway", // only authorized payment proce
ssor
    indexBy: ["userId", "validUntil"]
  },
  // Document stores (mutable but versioned)
  artifacts: {
    type: "documents",
    accessController: "steward-update", // only current steward can update
    indexBy: ["currentSteward", "artifactType", "status"]
  },

  userProfiles: {
    type: "documents",
    accessController: "self-update",
    indexBy: ["userId", "subscriptionTier"]
  },

  reputationScores: {
    type: "documents",
    accessController: "reputation-oracle", // calculated by trusted nodes
    indexBy: ["userId", "lastUpdated"]
  }
};
```

Custom Access Controllers

```
// Only current steward can transfer artifacts
class StewardOnlyAccessController {
  async canAppend(entry: any): Promise<boolean> {
    const artifact = await getArtifact(entry.payload.artifactId);
    const currentSteward = artifact.currentSteward;

    // Verify signature from current steward
    return verifySignature(
      entry.payload.signature,
      currentSteward,
      entryHash(entry.payload)
    );
  }
}

// Users can only write to their own attention log
```

```
class SelfOnlyAccessController {
  async canAppend(entry: any): Promise<boolean> {
    const userId = entry.payload.userId;
    const signerId = entry.identity;

    return userId === signerId;
  }
}
```

Fraud Prevention Measures

1. Attention Manipulation Prevention

```
// Detect suspicious attention patterns
function detectAttentionFraud(userId: string): FraudAlert[] {
  const alerts: FraudAlert[] = [];
  const userLog = getUserAttentionLog(userId);

  // Check for impossibly regular patterns
  const intervals = userLog.map((event, i) =>
    i > 0 ? event.timestamp - userLog[i-1].timestamp : 0
  ).slice(1);

  const avgInterval = intervals.reduce((a, b) => a + b) / intervals.length;
  const variance = intervals.reduce((sum, interval) =>
    sum + Math.pow(interval - avgInterval, 2), 0
  ) / intervals.length;

  if (variance < 1000) { // Less than 1 second variance
    alerts.push({
      type: "robotic_attention_pattern",
      severity: "high",
      description: "Attention switches show robotic regularity"
    });
  }

  // Check for impossible durations
  const longSessions = intervals.filter(interval => interval > 14400000); // >4
hours
  if (longSessions.length > userLog.length * 0.1) {
    alerts.push({
      type: "impossible_attention_duration",
      severity: "medium",
      description: "Too many impossibly long attention sessions"
    });
  }

  return alerts;
}
```

2. Economic Attack Prevention

```
// Prevent circular trading and value inflation
function detectCircularTrading(transferEvents: StewardshipTransferEvent
[]): boolean {
  const graph = buildTransferGraph(transferEvents);

  // Detect cycles where artifacts return to original owners quickly
  for (const cycle of detectCycles(graph)) {
    const timeSpan = cycle[cycle.length - 1].timestamp - cycle[0].timestamp;
    if (timeSpan < 86400000) { // Less than 24 hours
      return true; // Suspicious circular trading
    }
  }

  return false;
}

// Rate limiting for high-value transfers
function checkTransferRateLimit(userId: string): boolean {
  const recentTransfers = getRecentTransfers(userId, 3600000); // Last hou
r
  const highValueTransfers = recentTransfers.filter(t =>
    calculateArtifactValue(t.artifactId) > 10000 // >10 hours of attention
  );
  return highValueTransfers.length <= 3; // Max 3 high-value transfers per hour
}
```

3. Reputation-Based Security

```
// Progressive security based on reputation
function getSecurityLevel(userId: string): SecurityLevel {
  const reputation = getUserReputation(userId);

  if (reputation.reputationScore > 800) {
    return {
      level: "trusted",
      transferLimit: null, // No limits
      witnessRequired: false,
      subscriptionDiscount: 0.8 // 80% discount
    };
  } else if (reputation.reputationScore > 500) {
    return {
      level: "verified",
      transferLimit: 50000, // 50 hours worth of tokens
      witnessRequired: false,
      subscriptionDiscount: 0.5 // 50% discount
    };
  } else {
    return {
      level: "probationary",
      transferLimit: 10000, // 10 hours worth of tokens
      witnessRequired: true, // Require community witness
      subscriptionDiscount: 0 // No discount
    };
  }
}
```

Network Synchronization

Eventually Consistent Architecture

```
// Handle conflicts in distributed environment
class EventualConsistencyManager {
  async resolveConflict(
    localEvent: AttentionSwitchEvent,
    remoteEvent: AttentionSwitchEvent
  ): Promise<AttentionSwitchEvent> {
    // Timestamp wins for attention events
    if (localEvent.timestamp !== remoteEvent.timestamp) {
      return localEvent.timestamp < remoteEvent.timestamp ? localEvent : remoteEvent;
    }

    // Hash comparison for exact timestamp collisions
    const localHash = eventHash(localEvent);
    const remoteHash = eventHash(remoteEvent);

    return localHash < remoteHash ? localEvent : remoteEvent;
  }

  async mergeLogs(localLog: AttentionSwitchEvent[], remoteLog: AttentionSwitchEvent[]): Promise<AttentionSwitchEvent[]> {
    const merged = [...localLog, ...remoteLog];

    // Sort by index, then timestamp
    merged.sort((a, b) => {
      if (a.index !== b.index) return a.index - b.index;
      return a.timestamp - b.timestamp;
    });

    // Remove duplicates and resolve conflicts
    const deduplicated = [];
    for (let i = 0; i < merged.length; i++) {
      const current = merged[i];
      const next = merged[i + 1];

      if (!next || current.index !== next.index) {
        deduplicated.push(current);
      } else {
        deduplicated.push(await this.resolveConflict(current, next));
        i++; // Skip next item
      }
    }

    return deduplicated;
  }
}
```

This architecture creates a **fraud-resistant, eventually consistent** system where:

1. **Attention is the only source of value** - prevents artificial inflation
2. **Append-only logs prevent manipulation** - history cannot be rewritten
3. **Cryptographic stewardship** - only owners can transfer artifacts
4. **Subscription-based access** - prevents spam and funds infrastructure
5. **Zero-knowledge reputation** - privacy-preserving trust system
6. **Community witnessing** - high-value transfers can require verification
7. **Economic rate limiting** - prevents circular trading and wash sales

The system reimagines crypto security around **sacred economics** rather than speculative trading, creating a foundation for authentic abundance creation and grateful exchange.