Mehr Protocol

Complete Specification

Version 1.0 — Design Complete, Pre-Implementation

Generated February 25, 2026

Table of Contents

Hardware

Reference Designs

Device Capabilities by Tier

Development

Roadmap

Design Decisions

Open Questions

Full Specification

Mehr Network

A Decentralized Capability Marketplace Over Transport-Agnostic Mesh

Mehr is a decentralized network where every resource — bandwidth, compute, storage, connectivity — is a discoverable, negotiable, verifiable, payable capability. Nodes participate at whatever level their hardware allows. Nothing is required except a cryptographic keypair.

Vision

Strengthen Communities

The internet was supposed to connect people. Instead, it routed everything through distant data centers owned by a handful of corporations. Mehr reverses this: communication within a community is **free, direct, and unstoppable**. Trusted neighbors relay for each other at zero cost. The economic layer only activates when traffic crosses trust boundaries — just like the real world.

Democratize Communication

A village with no ISP should still be able to communicate. A country under internet shutdown should still have a mesh. A community that can't afford $30/month per household should be able to share one uplink across a neighborhood. Mehr makes communication infrastructure a commons, not a product.

One Decentralized Computer

Every device on the network — from a $30 solar relay to a GPU workstation — contributes what it can. Storage, compute, bandwidth, and connectivity are pooled into a single capability marketplace. Your phone delegates AI inference to a neighbor's GPU. Your Raspberry Pi stores data for the mesh. No single point of failure, no single point of control. The network **is** the computer.

Share Hardware, Save Money

Most hardware sits idle most of the time. A home internet connection averages less than 5% utilization. A desktop GPU sits unused 22 hours a day. Mehr turns idle capacity into shared infrastructure: you earn when others use your resources, and you pay when you use theirs. The result is that communities need far less total hardware to achieve the same capabilities.

Why Mehr?

The internet depends on centralized infrastructure: ISPs, cloud providers, DNS registrars, certificate authorities. When any of these fail — through censorship, natural disaster, or economic exclusion — people lose connectivity entirely.

Mehr is designed for a world where:

- A village with no internet can still communicate internally over LoRa radio
- A country with internet shutdowns can maintain mesh connectivity between citizens
- A community can run its own local network and bridge to the wider internet through any available uplink
- Every device — from a $30 solar-powered relay to a GPU workstation — contributes what it can and pays for what it needs

Core Principles

1. Transport Agnostic

Any medium that can move bytes is a valid link. The protocol never assumes IP, TCP, or any specific transport. It works from 500 bps radio to 10 Gbps fiber. A single node can bridge between multiple transports simultaneously.

2. Capability Agnostic

Nodes are not classified into fixed roles. A node advertises what it can do. What it cannot do, it delegates to a neighbor and pays for the service. Hardware determines capability; the market determines role.

3. Partition Tolerant

Network fragmentation is not an error state — it is expected operation. A village on LoRa **is** a partition. A country with internet cut **is** a partition. Every protocol layer functions correctly during partitions and converges correctly when partitions heal.

4. Anonymous by Default

Packets carry no source address. A relay node knows which neighbor handed it a packet, but not whether that neighbor originated it or is relaying it from someone else. Identity is a cryptographic keypair — not a name, not an IP address, not an account. Human-readable names are optional and self-assigned. You can use the network, earn MHR, host content, and communicate without ever revealing who you are.

5. Free Local, Paid Routed

Direct neighbors communicate for free. You pay only when your packets traverse other people's infrastructure. This mirrors real-world economics — talking to your neighbor costs nothing, sending a letter across the country does.

6. Layered Separation

Each layer depends only on the layer below it. Applications never touch transport details. Payment never touches routing internals. Security is not bolted on — it is structural.

Protocol Stack Overview

Mehr is organized into seven layers, each building on the one below. Click any layer to read its full specification.

How It Works — A Simple Example

1. **Alice** has a Raspberry Pi with a LoRa radio and WiFi. She's in a rural area with no internet.
2. **Bob** has a gateway node 5 km away with a cellular modem providing internet access.
3. **Carol** is somewhere on the internet and wants to message Alice.

Here's what happens:

- Carol's message is encrypted end-to-end for Alice's public key
- It routes through the internet to Bob's gateway
- Bob relays it over LoRa to Alice (earning a small MHR fee)
- Alice's device decrypts and displays the message
- Bob's relay cost is paid automatically through a bilateral payment channel

No central server. No accounts. No subscriptions. Just cryptographic identities and a marketplace for capabilities.

Next Steps

- **Understand the protocol**: Start with Physical Transport and work up the stack
- **Explore the economics**: Learn how MHR tokens and stochastic relay rewards enable decentralized resource markets
- **See the real-world impact**: Understand how Mehr affects existing economics and how participants earn
- **See the hardware**: Check out the reference designs for building Mehr nodes
- **Read the full spec**: The complete protocol specification covers every detail

Frequently Asked Questions

Plain-language answers. No jargon.

The Basics

What is Mehr?

Mehr is a network that lets devices talk to each other directly — without relying on a phone company, an ISP, or a cloud server. Your phone, laptop, or a cheap radio module can join the mesh and communicate with anyone nearby. Think of it like a community-owned telephone system that nobody controls.

How do I join?

Install the app (or flash a device) and create an account. That's it. Your account is just a cryptographic key — no email, no phone number, no sign-up form. You're on the network immediately.

What device do I need?

Anything from a $30 solar-powered radio relay to a smartphone to a full computer. The network adapts to what your device can do. Low-power devices relay text messages; powerful devices can host websites and run computations.

Is it free?

Talking to your friends and neighbors: always free. You mark people as "trusted" (like adding a contact), and all communication between trusted people costs nothing.

Reaching strangers or distant nodes: costs a tiny amount of MHR (the network's internal token). You earn MHR automatically by helping relay other people's traffic — so for most people, the system is self-sustaining. You earn by participating, and you spend by using.

Do I need to buy tokens?

No. You earn MHR by relaying traffic for others, which happens automatically in the background. The more your device helps the network, the more you earn. You can start using the network with zero tokens — free communication between trusted peers works immediately.

Finding Things

How do I find what's happening in my city?

Your device automatically discovers nearby nodes and their content. Here's how, from closest to furthest:

1. **Friends' feeds**: You follow people. Their posts show up on your device automatically, newest first. No algorithm choosing what you see — just a chronological feed.

2. **Neighborhood activity**: Your community has a label (like `mumbai-mesh` or `portland-mesh`). Subscribe to it and you'll see all local activity — forum posts, marketplace listings, wiki edits, new websites — from everyone in that community.

3. **Nearby nodes**: Your device constantly hears announcements from nearby nodes. You can browse what services and content exist within a few hops, like walking down a street and reading shop signs.

4. **Search**: If you want something specific that isn't nearby, you can search the wider network. It's slower and may cost a small amount, but you can find anything that anyone has published.

How do I find my friends?

By their name. Everyone can pick a name scoped to their community — like `alice@portland-mesh` or `ravi@mumbai-mesh`. Type it in and you're connected. You can also use private nicknames ("Mom", "Work") that only exist on your device.

If you're physically near someone, your devices will discover each other automatically over radio or WiFi — no names needed.

Is there a "feed" like Instagram or Twitter?

Yes, but better. You follow people and see their posts in chronological order. There is:

- **No algorithm** deciding what you see
- **No ads** injected into your feed
- **No central server** storing your social graph
- **No one** who can ban you from the network

On a good connection you get full images and video. On a weak radio link, you get text and tiny image previews. The app adapts automatically.

Can I browse websites?

Yes. People host websites on Mehr just like on the regular internet, but without needing a server or domain name. You visit them by name (`mysite@portland-mesh`) or by direct link. Popular sites load fast because copies are cached throughout the network automatically.

Communication

How do I message someone?

Open the messaging app, pick a contact, type your message. It's end-to-end encrypted — only you and the recipient can read it. If they're offline, the network holds the message and delivers it when they come back online (like email, but encrypted).

Can I make voice calls?

Yes, on connections with enough bandwidth. WiFi and cellular links support real-time voice. On slow radio links, voice isn't practical — use text messaging instead.

Can I send photos and videos?

Yes. The app adapts to your connection:

Connection	What you can send/receive
WiFi or cellular	Photos, videos, full media
Moderate radio link	Compressed images, text
Slow radio (LoRa)	Text only, with tiny image previews

You never need to think about this — the app handles it automatically.

What happens when I'm moving around?

Your device automatically handles roaming. It constantly listens for nearby nodes on all its radios (WiFi, Bluetooth, LoRa) and connects to the best one available — no manual switching required.

- **Walk into a cafe with a Mehr WiFi node?** Your device connects in under a second.
- **Walk out of WiFi range?** Traffic shifts to LoRa automatically. Apps adapt (images become text previews).
- **Visit the same cafe tomorrow?** Your device remembers it and reconnects instantly — no setup.
- **On a voice call while moving?** The call hands off between nodes with less than a second of interruption. Quality may change (WiFi → LoRa = high-fidelity → walkie-talkie) but the call

doesn't drop.

Think of it like your phone switching between cell towers — except there's no phone company, no SIM card, and no monthly bill.

Community

How do communities form?

You mark people as trusted. They mark you as trusted. When a group of people all trust each other, that's a community. Nobody "creates" it or "runs" it — it emerges from real-world relationships.

Optionally, you can all label yourselves with the same community name (like `portland-mesh`) so newcomers can find you.

Can I run a local forum?

Yes. A forum is just a shared space where community members post. A moderator contract enforces whatever rules your community agrees on. Different forums can have different rules — there's no platform-wide content policy.

Can I sell things on a local marketplace?

Yes. Post a listing (text, photos, price), and it's visible to your neighborhood. Buyers contact you directly. Payment can happen in person, through an external service, or through MHR escrow (the network holds the payment until both sides confirm the deal).

Can I host a website or blog?

Yes, and it's much simpler than traditional hosting:

Traditional web	Mehr
Rent a server	Not needed — content lives in the mesh
Buy a domain name ($10–50/year)	Pick a name for free (`myblog@mumbai-mesh`)
Get an SSL certificate	Not needed — everything is encrypted and verified automatically
Pay for traffic spikes	Visitors pay their own relay costs, not you

You pay only for storage (tiny amounts of MHR), and popular content gets cheaper because it's cached everywhere.

Privacy and Safety

Is it private?

Yes. Messages are end-to-end encrypted. Social posts can be public or neighborhood-only. There is no central server with a copy of your messages, your contacts, or your browsing history. Your identity is a cryptographic key — you never need to provide your real name.

Can someone spy on my messages?

No. End-to-end encryption means only the sender and recipient can read a message. Relay nodes carry encrypted blobs they cannot decrypt. Even your direct neighbors don't know if a packet originated from you or if you're just relaying it for someone else.

Can someone shut down the network?

No single point of failure. There's no server to seize, no company to shut down, no domain to block. As long as any two devices can reach each other — by radio, WiFi, Bluetooth, or anything else — the network works.

Economy

How does money work on Mehr?

MHR is the network's internal token. Think of it like arcade tokens — they're valuable inside the arcade (network services), but they're not meant for trading on an exchange.

- **You earn MHR** by relaying traffic, storing data, or providing other services
- **You spend MHR** when your messages cross through untrusted infrastructure
- **Talking to friends is always free** — MHR only matters at trust boundaries

What's it worth in real money?

MHR has no official exchange rate with any fiat currency. Its value comes from what it buys on the network — relay time, storage space, compute. The prices of these services float based on supply and demand, like any market. But you never need to convert MHR to dollars — it's a closed-loop system.

Can I get rich from MHR?

That's not the point. MHR is designed to be spent on services, not hoarded or traded. There's no ICO, no pre-mine, no exchange listing. You earn it by contributing, you spend it by consuming. The goal is a functioning economy, not a speculative asset.

Compared to What I Use Now

How is this different from the regular internet?

	Regular Internet	Mehr
Works without ISP	No	Yes — radio, WiFi, anything
Works during internet shutdown	No	Yes — local mesh continues
Free local communication	No — you pay your ISP	Yes — trusted peers are free
Your data on a corporate server	Yes (Google, Meta, etc.)	No — data stays on your devices and your community's mesh
Can be censored	Yes — ISPs, DNS, app stores	Extremely difficult — no central control point
Needs an account	Email, phone number, ID	Just a cryptographic key (anonymous)

Can Mehr replace my internet connection?

It depends on where you live.

In a **dense area** (apartment building, neighborhood, campus) where many nodes run WiFi, the mesh delivers 10–300 Mbps per hop — comparable to cable internet. Add a few shared internet uplinks (Starlink, fiber, cellular) and the community mesh handles distribution. Most people would save 50–75% on connectivity costs. You can browse, stream, video call — normal internet use.

In a **rural or remote area** with only LoRa radio coverage, Mehr delivers 0.3–50 kbps — enough for text messaging, basic social feeds, and push-to-talk voice, but not video streaming or modern web browsing. Here, Mehr isn't replacing your internet — it's providing communication where there was none, or sharing one expensive satellite connection across an entire village.

Your situation	What Mehr does
Dense urban, many WiFi nodes	Replaces individual ISP subscriptions — share uplinks, save money
Suburban, mixed WiFi + LoRa	Supplements your connection — free local communication, shared backup uplink

| Rural, LoRa only | Provides communication where there is none — text, voice, local services |
| No infrastructure at all | Only option that works — $30 solar radio nodes, no towers needed |

The key insight: Mehr doesn't compete with Starlink or cellular on raw speed. It **uses** them as transport — one Starlink dish becomes a shared community gateway. The mesh handles the local distribution and economics. Everyone gets internet access; the gateway operator earns; residents save.

How is this different from Signal or WhatsApp?

Signal and WhatsApp need internet access and rely on central servers for delivery. Mehr works without internet, stores messages across the mesh (not one company's servers), and the network itself is decentralized. Nobody can block your access because there's nothing to block.

How is this different from Bitcoin?

Bitcoin is money designed for global financial transactions. MHR is an internal utility token for paying network services. They share some concepts (cryptographic keys, no central authority) but serve completely different purposes. MHR is more like "bus tokens for the network" than a cryptocurrency.

Layer 0: Physical Transport

Mehr requires a transport layer that provides transport-agnostic networking over any medium supporting at least a half-duplex channel with ≥5 bps throughput and ≥500 byte MTU. The transport layer is a swappable implementation detail — Mehr defines the interface it needs, not the implementation.

Transport Requirements

The transport layer must provide:

- **Any medium is a valid link**: LoRa, WiFi, Ethernet, serial, packet radio, fiber, free-space optical
- **Multiple simultaneous interfaces**: A node can bridge between transports automatically
- **Announce-based routing**: No manual configuration of addresses, subnets, or routing tables
- **Mandatory encryption**: All traffic is encrypted; unencrypted packets are dropped as invalid
- **Sender anonymity**: No source address in packets
- **Constrained-link operation**: Functional at ≥5 bps

Current Implementation: Reticulum

The current transport implementation uses the Reticulum Network Stack, which satisfies all requirements above and is proven on links as slow as 5 bps. Mehr extends it with CompactPathCost annotations on announces and an economic layer above.

Reticulum is an implementation choice, not an architectural dependency. Mehr extensions are carried as opaque payload data within Reticulum's announce DATA field — a clean separation that allows the transport to be replaced with a clean-room implementation in the future without affecting any layer above.

Participation Levels

Not all nodes need to understand Mehr extensions. Three participation levels coexist on the same mesh:

Level	Node Type	Understands	Earns MHR	Marketplace
L0	Transport-only	Wire protocol only	No	No
L1	Mehr Relay	L0 + CompactPathCost + stochastic rewards	Yes (relay only)	No
L2	Full Mehr	Everything	Yes	Yes

L0 nodes relay packets and forward announces (including Mehr extensions as opaque bytes) but do not parse economic extensions, earn rewards, or participate in the marketplace. They are zero-cost hops from Mehr's perspective. This ensures the mesh works even when some nodes run the transport layer alone.

L1 nodes are the minimum viable Mehr implementation — they parse CompactPathCost, run the VRF relay lottery, and maintain payment channels. This is the target for ESP32 firmware.

L2 nodes implement the full protocol stack including capability marketplace, storage, compute, and application services.

Implementation Strategy

Platform	Implementation
Raspberry Pi, desktop, phone	Rust implementation (primary)
ESP32, embedded	Rust `no_std` implementation (L1 minimum)

All implementations speak the same wire protocol and interoperate on the same network.

Supported Transports

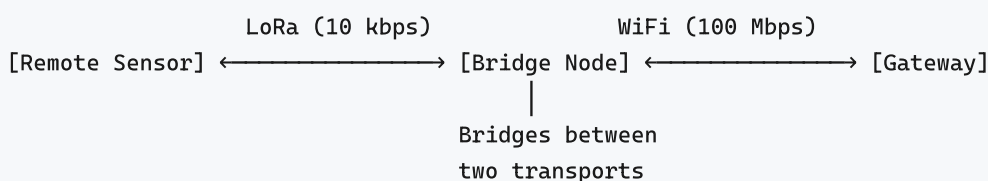| Transport	Typical Bandwidth	Typical Range	Duplex	Notes
LoRa (ISM band)	0.3-50 kbps	2-15 km	Half	Unlicensed, low power, high range. RNode as reference hardware.
WiFi Ad-hoc	10-300 Mbps	50-200 m	Full	Ubiquitous, short range
WiFi P2P (directional)	100-800 Mbps	1-10 km	Full	Point-to-point backbone links
Cellular (LTE/5G)	1-100 Mbps	Via carrier	Full	Requires carrier subscription
Ethernet	100 Mbps-10 Gbps	Local	Full	Backbone, data center
Serial (RS-232, AX.25)	1.2-56 kbps	Varies	Half	Legacy radio, packet radio
Fiber	1-100 Gbps	Long haul	Full	Backbone
Bluetooth/BLE	1-3 Mbps	10-100 m	Full	Wearables, phone-to-phone

A node can have **multiple interfaces active simultaneously**. The network layer selects the best interface for each destination based on cost, latency, and reliability.

Multi-Interface Bridging

A node with both LoRa and WiFi interfaces automatically bridges between the two networks. Traffic arriving on LoRa can be forwarded over WiFi and vice versa.

The bridge node is where bandwidth characteristics change dramatically — and where the capability marketplace becomes valuable. A bridge node can:

- Accept low-bandwidth LoRa traffic from remote sensors
- Forward it over high-bandwidth WiFi to a local network
- Earn relay rewards for the bridging service
- Advertise its bridging capability to nearby nodes

```
                    LoRa (10 kbps)              WiFi (100 Mbps)
   [Remote Sensor] ←——————————————→ [Bridge Node] ←——————————————→ [Gateway]
                                          |
                                     Bridges between
                                     two transports
```

Bandwidth Ranges and Their Implications

The 20,000x range between the slowest and fastest supported transports (500 bps to 10 Gbps) has profound implications for protocol design:

- **All protocol overhead must be budgeted.** Gossip, routing updates, and economic state consume bandwidth that could carry user data. On a 1 kbps LoRa link, every byte matters.
- **Data objects carry minimum bandwidth requirements.** A 500 KB image declares `min_bandwidth: 10000` (10 kbps). LoRa nodes never attempt to transfer it — they only propagate its hash and metadata.
- **Applications adapt to link quality.** The protocol provides link metrics; applications decide what to send based on available bandwidth.

NAT Traversal

Residential nodes behind NATs (common for WiFi and Ethernet interfaces) are handled at the transport layer. The Reticulum transport uses its link establishment protocol to traverse NATs — an outbound connection from behind the NAT establishes a bidirectional link without requiring port forwarding or STUN/TURN servers.

For nodes that cannot establish outbound connections (rare), the announce mechanism still propagates their presence. Traffic destined for a NATed node is routed through a neighbor that

does have a direct link — functionally equivalent to standard relay forwarding. No special NAT-awareness is needed at the Mehr protocol layers above transport.

What Mehr Adds Above Transport

The transport layer provides packet delivery, routing, and encryption. Mehr adds everything above:

Extension	Purpose
CompactPathCost on announces	Enables economic routing — cheapest, fastest, or balanced path selection
Stochastic relay rewards	Incentivizes relay operators without per-packet payment overhead
Capability advertisements	Makes compute, storage, and connectivity discoverable and purchasable
CRDT economic ledger	Tracks balances without consensus or blockchain
Trust graph	Enables free communication between trusted peers
Congestion control	CSMA/CA, per-neighbor fair sharing, priority queuing, backpressure

These extensions ride on top of the transport's existing gossip and announce mechanisms, staying within the protocol's bandwidth budget.

Layer 1: Network Protocol

The network protocol handles identity, addressing, routing, and state propagation across the mesh. It uses Reticulum as the transport foundation and extends it with cost-aware routing and economic state gossip.

Identity and Addressing

Mehr uses Reticulum's identity model. Every node has a cryptographic identity generated locally with no registrar:

```
NodeIdentity {
    keypair: Ed25519Keypair,           // 256-bit, generated locally
    public_key: Ed25519PublicKey,       // 32 bytes
    destination_hash: [u8; 16],        // truncated hash of public key
    x25519_public: X25519PublicKey,     // derived via RFC 7748 birational map
}
```

Destination Hash

The destination hash is the node's address — 16 bytes (128 bits), derived from the public key. This provides:

- **Flat address space**: No hierarchy, no subnets, no allocation authority
- **Self-assigned**: Any node can generate an address without asking permission
- **Negligible collision probability**: 2^128 possible addresses
- **Pseudonymous**: The hash is not linked to a real-world identity unless the owner publishes that association

A single node can generate **multiple destination hashes** for different purposes (personal identity, service endpoints, anonymous identities). Each is derived from a separate Ed25519 keypair.

Packet Format

Mehr uses the Reticulum packet format:

```
[HEADER 2 bytes] [ADDRESSES 16/32 bytes] [CONTEXT 1 byte] [DATA 0-465 bytes]
```

Header flags encode: propagation type (broadcast/transport), destination type (single/group/plain/link), and packet type (data/announce/link request/proof). Maximum overhead

per packet: 35 bytes.

Critical property (inherited from Reticulum): The source address is **NOT** in the header. Packets carry only the destination. Sender anonymity is structural.

Mehr Extension: Compact Path Cost

Mehr extends announces with a constant-size cost summary that each relay updates in-place as it forwards the announce:

```
CompactPathCost {
    cumulative_cost: u16,     // log₂-encoded µMHR/byte (2 bytes)
    worst_latency_ms: u16,    // max latency on any hop in path (2 bytes)
    bottleneck_bps: u8,       // log₂-encoded min bandwidth on path (1 byte)
    hop_count: u8,            // number of relays traversed (1 byte)
}
// Total: 6 bytes (constant, regardless of path length)
```

Each relay updates the running totals as it forwards:

- `cumulative_cost += my_cost_per_byte` (re-encoded to log scale)
- `worst_latency_ms = max(existing, my_measured_latency)`
- `bottleneck_bps = min(existing, my_bandwidth)`
- `hop_count += 1`

Log encoding for cost: `encoded = round(16 × log₂(value + 1))`. A u16 covers the full practical cost range with ~6% precision per step.

Log encoding for bandwidth: `encoded = round(8 × log₂(bps))`. A u8 covers 1 bps to ~10 Tbps with ~9% precision.

The CompactPathCost is carried in the announce DATA field using a TLV envelope:

```
MehrExtension {
    magic: u8 = 0x4E,            // 'N' – identifies Mehr extension presence
    version: u8,                 // extension format version
    path_cost: CompactPathCost,  // 6 bytes
    extensions: [{               // future extensions via TLV pairs
        type: u8,
        length: u8,
        data: [u8; length],
    }],
}
// Minimum size: 8 bytes (magic + version + path_cost)
```

Nodes that don't understand the `0x4E` magic byte forward the DATA field as opaque payload. Mehr-aware nodes parse and update it.

Why No Per-Relay Signatures

Earlier designs signed each relay's cost annotation individually (~84 bytes per relay hop). This is unnecessary for three reasons:

1. **Routing decisions are local.** You select a next-hop neighbor. You only need to trust your neighbor's cost claim — and your neighbor is already authenticated by the link-layer encryption.
2. **Trust is transitive at each hop.** Your neighbor trusts *their* neighbor (link-authenticated), who trusts *their* neighbor, and so on. No node needs to verify claims from relays it has never communicated with.
3. **The market enforces honesty.** A relay that inflates path costs gets routed around. A relay that deflates costs loses money on every packet. Economic incentives are a cheaper and more robust enforcement mechanism than cryptographic proofs for cost claims.

The announce itself remains signed by the destination node (proving authenticity of the route). The path cost summary is trusted transitively through link-layer authentication at each hop — analogous to how BGP trusts direct peers, not every AS along the path.

Routing

Routing is destination-based with cost annotations, formalized as **greedy forwarding on a small-world graph**. Each node maintains a routing table:

```
RoutingEntry {
    destination: DestinationHash,
    next_hop: InterfaceID + LinkAddress, // which interface, which neighbor

    // From CompactPathCost (6 bytes in announce)
    cumulative_cost: u16,              // log₂-encoded µMHR/byte
    worst_latency_ms: u16,             // max latency on path
    bottleneck_bps: u8,                // log₂-encoded min bandwidth
    hop_count: u8,                     // relay count

    // Locally computed
    reliability: u8,                   // 0-255 (0=unknown, 255=perfect) — avoids FP on
 ESP32

    last_updated: Timestamp,
    expires: Timestamp,
}
```

Small-World Routing Model

Mehr routing is based on the **Kleinberg small-world model**, adapted for a physical mesh with heterogeneous transports. This provides a formal basis for routing scalability.

The Network as a Small-World Graph

The destination hash space `[0, 2^128)` forms a **ring**. The circular distance between two addresses is:

```
ring_distance(a, b) = min(|a − b|, 2^128 − |a − b|)
```

The physical mesh naturally provides two types of links, matching Kleinberg's model:

- **Short-range links** (lattice edges): LoRa, WiFi ad-hoc, BLE — these connect geographically nearby nodes, forming an approximate 2D lattice determined by physical proximity.
- **Long-range links** (Kleinberg contacts): Directional WiFi, cellular, internet gateways, fiber — these connect distant nodes, providing shortcuts across the ring.

Kleinberg's result proves that greedy forwarding achieves **O(log² N) expected hops** when long-range link probability follows `P(u→v) ∝ 1/d(u,v)^r` with clustering exponent `r` equal to the network dimension. The distribution of real-world backbone links (many local WiFi, fewer city-to-city, even fewer intercontinental) naturally approximates this harmonic distribution.

Greedy Forwarding with Cost Weighting

At each hop, the current node selects the neighbor that minimizes a scoring function:

```
score(neighbor) = α · norm_ring_distance(neighbor, destination)
                + β · norm_cumulative_cost(neighbor)
                + γ · norm_worst_latency(neighbor)
```

Where `norm_*` normalizes each metric to `[0, 1]` across the candidate set. Normalization is performed on **decoded** values (not the log-encoded wire representation):

```
Decoding (for normalization):
  decoded_cost = (2 ^ (encoded / 16.0)) − 1    // inverse of log₂ encoding
  decoded_bw   = 2 ^ (encoded / 8.0)           // inverse of bandwidth encoding

  norm_cumulative_cost = decoded_cost(neighbor) / max_decoded_cost_in_candidate_set
  norm_worst_latency   = neighbor.worst_latency_ms / max_latency_in_candidate_set
```

This preserves the true cost ratios. Log-encoded values compress dynamic range for wire efficiency but must not be used directly in scoring — otherwise a 1000x cost difference would appear as only

~2x.

The weights α, β, γ are derived from the per-packet `PathPolicy`:

```
PathPolicy: enum {
    Cheapest,                          // α=0.1, β=0.8, γ=0.1
    Fastest,                           // α=0.1, β=0.1, γ=0.8
    MostReliable,                      // maximize delivery probability
    Balanced(cost_weight, latency_weight, reliability_weight),
}
```

Pure greedy routing (α=1, β=0, γ=0) guarantees **O(log² N) expected hops**. Cost and latency weighting trades path length for economic efficiency — a path may take more hops if each hop is cheaper or faster.

Applications specify their preferred policy:

- **Voice traffic** uses `Fastest` — latency matters most
- **Bulk storage replication** uses `Cheapest` — cost efficiency matters most
- **Default** is `Balanced` — a weighted combination of all factors

With N nodes where each has O(1) long-range links (typical for relay nodes), expected path length is **O(log² N)**. Backbone nodes with O(log N) connections reduce this to **O(log N)**.

Why Mehr Does Not Need Location Swapping

Unlike Freenet/Hyphanet, which uses location swapping to arrange nodes into a navigable topology, Mehr does not need this mechanism:

1. **Destination hashes are self-assigned** — each node's position on the ring is fixed by its Ed25519 keypair.
2. **Announcements build routing tables** — when a node announces itself, it creates routing table entries across the mesh that function as navigable links.
3. **Multi-transport bridges are natural long-range contacts** — a node bridging LoRa to WiFi to internet inherently provides the long-range shortcuts that make the graph navigable.

The announcement propagation itself creates the navigable topology. Each announcement that reaches a distant node via a backbone link creates exactly the kind of long-range routing table entry that Kleinberg's model requires.

Path Discovery

Path discovery works via announcements:

1. A node announces its destination hash to the network, signed with its Ed25519 key

2. The announcement propagates through the mesh via greedy forwarding, with each relay updating the CompactPathCost running totals in-place (no per-relay signatures — link-layer authentication is sufficient)

3. Receiving nodes record the path (or multiple paths) and select based on the scoring function above

4. Multiple paths are retained and scored — the best path per policy is used, with fallback to alternatives on failure

Announce Propagation Rules

Announces are **event-driven with periodic refresh**, not purely periodic:

```
Announce triggers:
  – First boot / new identity: immediate announce
  – Interface change: announce on new interface within 1 gossip round
  – Cost change > 25%: re-announce with updated CompactPathCost
  – Periodic refresh: every 30 minutes (1,800 seconds)
  – Forced refresh: on peer request (pull-based for constrained links)
```

Hop limit: Announces carry a `max_hops` field (u8, default 128). Each relay decrements by 1; announces at 0 are not forwarded. This prevents unbounded propagation in large meshes while ensuring $O(\log^2 N)$ reachability.

Expiry: Routing entries expire at `last_updated + announce_interval × 3` (default 90 minutes). If no refresh is received, the entry is marked stale (still usable at lower priority) for one additional interval, then evicted. On memory pressure, LRU eviction removes the least-recently-used stale entries first, then lowest-reliability active entries.

Link failure detection: If a direct neighbor misses 3 consecutive gossip rounds (3 minutes) without response, the link is marked down. All routing entries using that neighbor as next-hop are immediately marked stale (not deleted — the neighbor may return). After 10 missed rounds, entries are evicted.

Gossip Protocol

All protocol-level state propagation uses a common gossip mechanism:

```
GossipRound (every 60 seconds with each neighbor):

1. Exchange state summaries (bloom filters of known state)
2. Identify deltas (what I have that you don't, and vice versa)
3. Exchange deltas (compact, only what's new)
4. Apply received state via CRDT merge rules
```

Gossip Bloom Filter

State summaries use a compact bloom filter to identify deltas without exchanging full state:

```
GossipFilter {
    bits: [u8; N],          // N scales with known state entries
    hash_count: 3,          // 3 independent hash functions (Blake3-derived)
    target_fpr: 1%,         // 1% false positive rate (tolerant — FP only causes redundant
delta)
}
```

Known state entries	Filter size	FPR
100	120 bytes	~1%
1,000	1.2 KB	~1%
10,000	12 KB	~1%

On constrained links (below 10 kbps), the filter is capped at 256 bytes — entries beyond the filter capacity are omitted (pull-only mode for Tiers 3-4 handles this). False positives are harmless: they cause a delta item to not be requested, but the item will be caught in the next round when the bloom filter is regenerated.

New node joining: A node with empty state sends an all-zeros bloom filter. The neighbor detects maximum divergence and sends a prioritized subset of state (Tier 1 first, then Tier 2, etc.) spread across multiple gossip rounds to avoid link saturation.

A single gossip round multiplexes all protocol state:

- Routing announcements (with cost annotations)
- Ledger state (settlements, balances)
- Trust graph updates
- Capability advertisements
- DHT metadata
- Pub/sub notifications

Bandwidth Budget

Total protocol overhead targets **≤10% of available link bandwidth**, allocated by priority tier:

```
Gossip Bandwidth Budget (per link):

  Tier 1 (critical):  Routing announcements       — up to 3%
  Tier 2 (economic):  Payment + settlement state   — up to 3%
```

```
    Tier 3 (services):   Capabilities, DHT, pub/sub    — up to 2%
    Tier 4 (social):     Trust graph, names            — up to 2%
```

On constrained links (< 10 kbps), the budget adapts automatically:

- Tiers 3–4 switch to **pull-only** (no proactive gossip — only respond to requests)
- Payment batching interval increases from 60 seconds to **5 minutes**
- Capability advertisements limited to **Ring 0 only** (direct neighbors)

Link type	Routing	Payment	Services	Trust/Social	Total
1 kbps LoRa	~1.5%	~0.5%	pull-only	pull-only	~2%
50 kbps LoRa	~2%	~2%	~1%	~1%	~6%
10+ Mbps WiFi	~1%	~1%	~2%	~2%	~6%

This tiered model ensures constrained links are never overwhelmed by protocol overhead, while higher-bandwidth links gossip more aggressively for faster convergence.

Congestion Control

User data has three layers of congestion control. Protocol gossip is handled separately by the bandwidth budget.

Link-Level Collision Avoidance (CSMA/CA)

On half-duplex links (LoRa, packet radio), mandatory listen-before-talk:

```
LinkTransmit(packet):
  1. CAD scan (LoRa Channel Activity Detection, ~5ms)
  2. If channel busy:
       backoff = random(1, 2^attempt) × slot_time
       slot_time = max_packet_airtime for this link
                 (~200ms at 1 kbps for 500-byte MTU)
  3. Max 7 backoff attempts → drop packet, signal congestion upstream
  4. If channel clear → transmit
```

On full-duplex links (WiFi, Ethernet), the transport handles collision avoidance natively — this layer is a no-op.

Per-Neighbor Token Bucket

Each outbound link enforces fair sharing across neighbors:

```
LinkBucket {
    link_id: InterfaceID,
    capacity_tokens: u32,        // link_bandwidth_bps × window_sec / 8
    tokens: u32,                 // current available (1 token = 1 byte)
    refill_rate: u32,            // bytes/sec = measured_bandwidth × (1 -
protocol_overhead)
    per_neighbor_share: Map<NodeID, u32>,
}


Bandwidth measurement:
  measured_bandwidth = exponential moving average of successfully-transmitted bytes/sec
  Half-life: 60 seconds (adapts within ~3 half-lives = 3 minutes)
  On transport change (e.g., LoRa → WiFi): reset EMA to the new link's nominal rate,
    then converge from there
  refill_rate = measured_bandwidth × 0.90  (10% reserved for protocol overhead)
  per_neighbor_share = refill_rate / num_active_neighbors  (user data only;
    protocol gossip has its own budget per the bandwidth tiers above)
```

Fair share is `refill_rate / num_active_neighbors` by default. Neighbors with active payment channels get share weighted proportionally to channel balance — paying for bandwidth earns proportional priority.

When a neighbor exceeds its share, packets are queued (not dropped). If the queue exceeds a depth threshold, a backpressure signal is sent.

Priority Queuing

Four priority levels for user data, scheduled with strict priority and starvation prevention (P3 guaranteed at least 10% of user bandwidth):

Priority	Traffic Type	Examples	Queue Policy
P0	Real-time	Voice (Codec2), interactive control	Tail-drop at 500ms deadline
P1	Interactive	Messaging, DHT lookups, link establishment	FIFO, 5s max queue time
P2	Standard	Social posts, pub/sub, MHR-Name	FIFO, 30s max queue time
P3	Bulk	Storage replication, large file transfer	FIFO, unbounded patience

Within a priority level, round-robin across neighbors. On half-duplex links, preemption occurs at packet boundaries only.

Backpressure Signaling

When an outbound queue exceeds 50% capacity, a 1-hop signal is sent to upstream neighbors:

```
CongestionSignal {
    severity: enum {          // 2 bits
```

```
        Moderate,              // reduce sending rate by 25%
        Severe,                // reduce by 50%, reroute P2/P3 traffic
        Saturated,             // stop P2/P3, throttle P1, P0 only
    },
    scope: enum {              // 2 bits
        ThisLink,              // only the link to the signaling neighbor is congested
        AllOutbound,           // all outbound links on this node are congested
    },
    estimated_drain_ms: u16,   // estimated time until queue drains
}
// Byte 0: [severity (2 bits) | scope (2 bits) | reserved (4 bits)]
// Bytes 1-2: estimated_drain_ms (u16, little-endian)
// Total: 3 bytes
```

The signal does not identify which internal interface is congested — upstream peers only need to know whether to reduce traffic through this node (`ThisLink` for targeted rerouting, `AllOutbound` if the node itself is overloaded). Internal link topology is not exposed.

```
<a id="protocol-network-protocol--dynamic-cost-response"></a>

### Dynamic Cost Response

Congestion increases the effective cost of a link. When queue depth exceeds 50%:
```

effective_cost = base_cost × (1 + (queue_depth / queue_capacity)^2)

```
The quadratic term ensures gentle increase at moderate load and sharp increase near
saturation. The updated cost propagates in the next gossip round's CompactPathCost, causing
upstream nodes to naturally reroute traffic to less-congested paths. This is a local
decision — no protocol extension beyond normal cost updates.

<a id="protocol-network-protocol--time-model"></a>

## Time Model

Mehr does not require global clock synchronization. Time is handled through three
mechanisms:

<a id="protocol-network-protocol--logical-clocks"></a>

### Logical Clocks

Packet headers carry a **Lamport timestamp** incremented at each hop. Used for ordering
events and detecting stale routing entries. If a node receives a routing announcement with
a lower logical timestamp than one already in its table for the same destination, the older
announcement is discarded.

<a id="protocol-network-protocol--neighbor-relative-time"></a>
```

Neighbor-Relative Time

During link establishment, nodes exchange their local monotonic clock values. Each node maintains a `clock_offset` per neighbor. Relative time between any two direct neighbors is accurate to within RTT/2.

Used for: agreement expiry, routing entry TTL, payment channel batching intervals.

Epoch-Relative Time

Epochs define coarse time boundaries. "Weekly" means approximately **10,000 settlement batches after the previous epoch** — not wall-clock weeks. The epoch trigger is settlement count, not elapsed time.

The "30-day grace period" for epoch finalization is defined as **4 epochs after activation**, tolerating clock drift of up to 50% without protocol failure.

All protocol `Timestamp` fields are `u64` values representing milliseconds on the node's local monotonic clock (not wall-clock). Conversion to neighbor-relative or epoch-relative time is performed at the protocol layer.

<div class="page-break"></div>

Layer 2: Security

Security in Mehr is structural, not bolted on. Every layer of the protocol incorporates cryptographic protections. There is no trusted infrastructure — no certificate authorities, no trusted servers, no DNS.

Threat Model

Mehr assumes the worst:

- **Open network**: Any node can join. Nodes may be malicious.
- **Hostile observers**: All link-layer traffic may be monitored (especially radio).
- **No trusted infrastructure**: No certificate authorities, no trusted servers, no DNS.
- **State-level adversaries**: Governments may control internet gateways and operate nodes within the mesh.

Mehr does **not** attempt to defend against:

- **Global traffic analysis**: A sufficiently powerful adversary monitoring all links simultaneously can correlate traffic patterns. [Opt-in onion routing](#protocol-security--onion-routing-opt-in) mitigates this for individual packets but does not defeat a global

```
    adversary.
    - **Physical compromise**: If an adversary physically captures a node, they obtain its
    private key and all local state.

    <a id="protocol-security--encryption-model"></a>

    ## Encryption Model

    <a id="protocol-security--link-layer-encryption-hop-by-hop"></a>

    ### Link-Layer Encryption (Hop-by-Hop)

    Every link between two nodes is encrypted using a session key derived from X25519 Diffie-
    Hellman key exchange:
```

Link establishment:

1. Alice and Bob exchange X25519 ephemeral public keys

2. Both derive shared_secret = X25519(my_ephemeral, their_ephemeral)

3. session_key = Blake2b(shared_secret || alice_pub || bob_pub)

4. All traffic on this link encrypted with ChaCha20-Poly1305(session_key) Nonce: 64-bit counter (zero-padded to 96 bits), incremented per packet Counter is per session_key — reset to 0 on each key rotation No nonce reuse risk: key rotation occurs well before 2^{64} packets

5. Keys rotated periodically (every 1 hour of local monotonic time, or max(1 MB, bandwidth_bps × 60s) of data, whichever first — this scales the data threshold to ~1 minute of link capacity, preventing excessive rotation on fast links). "1 hour" is measured by each node's local monotonic clock independently — no synchronization needed. Either side of the link can initiate rotation; the peer accepts and derives a new session key via fresh ephemeral key exchange

```
    This prevents passive observers from reading packet contents or metadata beyond the
    cleartext header fields needed for routing.

    <a id="protocol-security--end-to-end-encryption-data-payloads"></a>

    ### End-to-End Encryption (Data Payloads)

    Data packets are encrypted by the sender for the destination using the destination's public
    key. Relay nodes **cannot** read the payload:
```

E2E encryption for a message to Bob:

1. Alice generates ephemeral X25519 keypair

2. shared_secret = X25519(alice_ephemeral, bob_x25519_public)

3. payload_key = Blake2b(shared_secret || alice_ephemeral_pub)

4. encrypted_payload = ChaCha20-Poly1305(payload_key, plaintext)

5. Packet contains: alice_ephemeral_pub || encrypted_payload

6. Bob derives the same payload_key and decrypts

```
This provides **forward secrecy per message** — each message uses a unique ephemeral key.
Compromise of one message's key does not compromise any other message.


<a id="protocol-security--what-relay-nodes-can-see"></a>


### What Relay Nodes Can See


| Visible | Hidden |
├─────────┼─────────┤
| Destination hash | Source address |
| Hop count | Payload contents |
| Packet size | Application-layer data |
| Timing | Sender identity |


<a id="protocol-security--authentication"></a>


## Authentication

Node identity is **self-certifying**. A node proves it owns a destination hash by signing
with the corresponding Ed25519 private key. No certificates, no PKI, no trust hierarchy.


- **Payment channels**: Both parties sign every state update. Forgery requires the other
party's private key.
- **Capability agreements**: Both provider and consumer sign. Neither can forge the other's
commitment.
- **Announcements**: Path announcements are signed by the announcing node. Relay nodes
update the [CompactPathCost](#protocol-network-protocol--mehr-extension-compact-path-cost)
running totals (not individually signed — link-layer authentication at each hop is
sufficient). Malicious relays can lie about costs, but the economic model disincentivizes
this — overpriced nodes are routed around, underpriced nodes lose money.


<a id="protocol-security--privacy"></a>


## Privacy

<a id="protocol-security--sender-anonymity"></a>


### Sender Anonymity

Packets do not carry source addresses. A relay node knows which neighbor sent it a packet,
but not whether that neighbor originated the packet or is relaying it from someone else.


<a id="protocol-security--recipient-privacy"></a>


### Recipient Privacy

Destination hashes are pseudonymous. A hash is not linked to a real-world identity unless
```

```
    the user chooses to publish that association (e.g., via MHR-Name).


    <a id="protocol-security--traffic-analysis-resistance"></a>


    ### Traffic Analysis Resistance


    Basic protections (always active):
    - Link-layer encryption prevents content inspection
    - Variable-rate padding on LoRa links obscures traffic patterns
    - No source address in packet headers


    <a id="protocol-security--onion-routing-opt-in"></a>


    ### Onion Routing (Opt-In)


    For high-threat environments where an adversary monitors multiple links simultaneously,
    per-packet layered encryption is available as an opt-in privacy upgrade via
    `PathPolicy.ONION_ROUTE`:
```

Onion-routed packet (3-hop default):

1. Sender selects 3 intermediate relays (at least 1 outside trust neighborhood)

2. Wraps message in 3 encryption layers (outermost = first relay)

3. Each layer: 16-byte nonce + 16-byte Poly1305 tag = 32 bytes overhead

4. Each relay decrypts one layer, reads next-hop destination, forwards

5. Final relay decrypts innermost layer and delivers to destination

Overhead: 32 bytes × 3 hops = 96 bytes Usable payload on LoRa (465 max): 369 bytes (~79% efficiency)

```
    Key properties:
    - **Stateless**: No circuit establishment, no relay-side state. Each packet is
    independently routable
    - **Opt-in**: Enabled per-packet via `PathPolicy.ONION_ROUTE` with configurable hop count
    (default 3)
    - **Cover traffic**: Optional constant-rate dummy packets (1/minute, off by default) for
    timing analysis resistance on high-threat links
    - **Not for voice**: The payload overhead and additional latency make onion routing
    unsuitable for real-time voice. Recommended for text messaging in high-threat scenarios


    <a id="protocol-security--key-rotation"></a>


    ### Key Rotation


    - **Long-lived keys** (Ed25519 identity): Used only for signing, never for encryption
    - **Ephemeral keys**: Used for encryption, discarded after use
    - Compromise of an ephemeral key does not compromise past or future communications


    <a id="protocol-security--sybil-resistance"></a>
```

```
## Sybil Resistance

An attacker can generate unlimited identities (Sybil attack). Mehr mitigates this through
economic mechanisms rather than identity verification:

1. **Payment channel deposits**: Opening a channel requires visible balance. Sybil nodes
with no balance cannot participate in the economy.
2. **Reputation accumulation**: Reputation is earned through verified service delivery over
time. New identities start with zero reputation. Creating many identities dilutes rather
than concentrates reputation.
3. **Trust graph**: A Sybil attacker needs real social relationships to gain trust. Trusted
peers [vouch economically](#economics-trust-neighborhoods--trust-based-credit) — they
absorb the debts of nodes they trust, making trust costly to extend.
4. **Proof of service**: Stochastic relay rewards use a [VRF-based lottery](#economics-
payment-channels--how-stochastic-rewards-work) that produces exactly one verifiable outcome
per (relay, packet) pair. A node can only earn by actually delivering packets, and cannot
grind for favorable lottery outcomes.
5. **Transitive credit limits**: Even if a Sybil node gains one trust relationship,
transitive credit is capped at 10% per hop and rate-limited for new relationships.

<a id="protocol-security--reputation"></a>

## Reputation

Reputation is a **locally computed, per-neighbor score** — not a global value. There is no
network-wide reputation database. Each node maintains its own view of how reliable its
peers are.

<a id="protocol-security--reputation-state"></a>

### Reputation State
```

PeerReputation { node_id: NodeID, relay_score: u16, // 0-10000 (fixed-point, 2 decimal places) storage_score: u16, // 0-10000 compute_score: u16, // 0-10000 total_interactions: u32, // number of completed agreements failed_interactions: u32,// number of failed/disputed agreements first_seen_epoch: u64, // how long we've known this peer last_updated: Timestamp, }

```
<a id="protocol-security--how-reputation-is-earned"></a>

### How Reputation Is Earned

Each completed capability agreement adjusts the relevant score:
```

On agreement completion: if successful: score += (10000 - score) / 100 // diminishing returns — harder to gain at higher scores if failed: score -= score / 10 // 10% penalty per failure — fast to lose

Initialization:

- New peer (no interactions, no referral): score = 0 (unknown)

- New peer with trusted referral: score = min(5000, referrer_score × 0.3)

- Referral → first-hand transition: after 5 successful interactions, first-hand score fully replaces the referral score

- Referral expiry: 500 gossip rounds (~8 hours) without refresh

```
<a id="protocol-security--how-reputation-is-used"></a>

### How Reputation Is Used

- **Credit line sizing**: Nodes extend larger credit lines to higher-reputation peers
- **Capability selection**: When multiple providers offer the same capability, the consumer
considers reputation alongside cost and latency
- **Storage agreement duration**: Nodes with higher storage scores get offered longer
storage contracts
- **Epoch consensus**: Epoch proposals from higher-reputation nodes are preferred when
competing proposals conflict

<a id="protocol-security--properties"></a>

### Properties

- **Local only**: Each node computes its own reputation scores. No gossip of reputation
values — this prevents reputation manipulation by flooding the network with fake
endorsements
- **First-hand primary**: Scores are based primarily on direct interactions. First-hand
experience always takes precedence over third-party information
- **No global score**: There is no way to ask "what is node X's reputation?" There is only
"what is my experience with node X?" This makes reputation Sybil-resistant — an attacker
can't inflate a score without actually providing good service to the scoring node

<a id="protocol-security--trust-weighted-referrals"></a>

### Trust-Weighted Referrals

When a node has no direct experience with a peer, it can query trusted neighbors for their
first-hand scores. Referrals help new nodes bootstrap but are tightly bounded to limit
manipulation:

- **1-hop only**: Only direct trusted peers can provide referrals. No transitive gossip — a
referral from a friend-of-a-friend is not accepted. This limits the manipulation surface to
corruption of your direct trusted peers
- **Weight formula**: `referral_weight = trust_score_of_referrer / max_score × 0.3` — even
a maximally trusted referrer's opinion carries only 30% of direct experience weight
- **Capped at 50%**: A referred reputation score cannot exceed 5000 (50% of max). A
referral alone cannot make a peer fully trusted — direct interaction is required to reach
higher scores
- **Overwritten by experience**: Referral scores are advisory. After the first few direct
interactions, first-hand experience overwrites the referral entirely
- **Expiry**: Referral scores expire after 500 gossip rounds (~8 hours at 60-second
```

```
intervals) without refresh from the referrer
- **Anti-collusion**: Since only 1-hop referrals are accepted and each is capped, a
colluding cluster must corrupt your direct trusted peers to manipulate scores — which
already breaks the trust model regardless of reputation


<a id="protocol-security--key-management"></a>


## Key Management


| Operation | Method |
├───────────┼────────┤
| **Generation** | Ed25519 keypair from cryptographically secure random source on first
boot |
| **Storage** | Private key encrypted at rest (ChaCha20-Poly1305 with user passphrase, or
hardware secure element) |
| **Recovery** | Social recovery via Shamir's Secret Sharing — split key into N shares,
recover with K-of-N |
| **Revocation** | No global revocation mechanism (intentional — no infrastructure that can
be coerced into revoking keys) |


The absence of a revocation mechanism is a deliberate tradeoff. A user who loses their key
loses their identity and balance, but no authority can forcibly revoke anyone's identity.


<a id="protocol-security--key-compromise-advisory"></a>


### Key Compromise Advisory


While there is no revocation, a node that detects its key has been compromised can
broadcast an advisory:
```

KeyCompromiseAdvisory { compromised_key: Ed25519PublicKey, new_key: Ed25519PublicKey, // optional migration target sequence: u64, // monotonic counter — prevents replay of old advisories evidence: enum { SignedByBoth(sig_old, sig_new), // proves control of both keys SignedByOldOnly(sig_old), // can only prove old key ownership }, timestamp: u64, }

```
The `sequence` field is monotonically increasing per compromised key. Receiving nodes only
accept an advisory if its sequence is strictly greater than any previously seen advisory
for the same `compromised_key`. This prevents an attacker from replaying an old advisory to
override a newer one.

This is **advisory, not authoritative**. Receiving nodes may:
- Flag the old identity as potentially compromised
- Require re-authentication for high-value operations
- Accept the new key if evidence includes both signatures (strongest proof)

**Conflict resolution**: An attacker holding the stolen key could issue a counter-advisory
claiming the legitimate owner's *new* key is compromised. Receiving nodes resolve
conflicting advisories as follows:

1. **`SignedByBoth` always wins over `SignedByOldOnly`** — proving control of both keys is
```

strictly stronger evidence than proving control of only one
2. **Multiple `SignedByOldOnly` advisories cancel out** — if two different advisories both signed only by the old key claim different new keys, both are suspect. Receiving nodes flag the old identity as compromised but accept neither new key automatically.
3. **Trust-weighted resolution** — if the advisory is vouched for by trusted peers (who can attest to knowing the real owner), it is weighted more heavily

This does not prevent an attacker from continuing to use the stolen key. It provides a mechanism for the legitimate owner to signal compromise and begin migration — strictly better than no mechanism at all. The `SignedByBoth` evidence type is the only reliable migration path; users should generate their new keypair **before** the old one is exposed whenever possible.

Cryptographic Primitives Summary

Purpose	Algorithm	Key Size
Identity / Signing	Ed25519	256-bit (32-byte public key)
Key Exchange	X25519 (Curve25519 DH)	256-bit
Identity Hashing	Blake2b	256-bit
Content Hashing	Blake3	256-bit
Symmetric Encryption	ChaCha20-Poly1305	256-bit key, 96-bit nonce
Address Derivation	Blake2b truncated	128-bit (16-byte destination hash)
Relay Lottery (VRF)	ECVRF-ED25519-SHA512-TAI ([RFC 9381](https://www.rfc-editor.org/rfc/rfc9381))	Reuses Ed25519 keypair; 80-byte proof

Hash Algorithm Split

Mehr uses two hash algorithms for distinct purposes:

- **Blake2b** — Identity derivation and key derivation. Chosen for compatibility with the Ed25519/X25519 ecosystem and its proven security margin. Used in: `destination_hash`, `session_key` derivation.
- **Blake3** — Content addressing and general hashing. Chosen for speed (3x faster than Blake2b on general data) and built-in Merkle tree support for streaming verification. Used in: `DataObject` hash, contract hash, DHT keys, `ChannelState` hash.

Both produce 256-bit outputs. The protocol never mixes them — identity operations use Blake2b, data operations use Blake3.

Ed25519 to X25519 Conversion

X25519 public keys are derived from Ed25519 public keys using the birational map defined in **RFC 7748 Section 4.1**: the Ed25519 public key (a compressed Edwards point) is converted to Montgomery form by computing `u = (1 + y) / (1 - y) mod p`, where `y` is the Edwards y-coordinate and `p = 2^255 - 19`. This is a standard, well-analyzed transformation used by libsodium, OpenSSL, and other major cryptographic libraries.

```
<div class="page-break"></div>


<a id="economics-mhr-token"></a>

# MHR Token

MHR is the unit of account for the Mehr network. It is not a speculative asset — it is the
internal currency for purchasing capabilities from nodes outside your trust network.


<a id="economics-mhr-token--properties"></a>

## Properties
```

MHR Properties: Smallest unit: 1 µMHR (micro-MHR) Initial distribution: Proof-of-service mining only (no ICO, no pre-mine) Supply ceiling: 2^64 µMHR (~18.4 × 10^18 µMHR, asymptotic — never reached)

```
<a id="economics-mhr-token--supply-model"></a>

### Supply Model

MHR has an **asymptotic supply ceiling** with **decaying emission**:

| Phase | Epoch Range | Emission Per Epoch |
|───────|─────────────|────────────────────|
| Bootstrap | 0-99,999 | 10^12 µMHR (1,000,000 MHR) |
| Halving 1 | 100,000-199,999 | 5 × 10^11 µMHR |
| Halving 2 | 200,000-299,999 | 2.5 × 10^11 µMHR |
| Halving N | N × 100,000 - (N+1) × 100,000 - 1 | 10^12 × 2^(-N) µMHR |
| Tail | When halved reward is below floor | 0.1% of circulating supply / estimated epochs
per year |
```

Emission formula: epoch_reward(e) = max(10^12 >> (e / 100_000), // discrete halving (bit-shift) circulating_supply * 0.001 / E_year // tail floor)

E_year = trailing 1,000-epoch moving average of epoch frequency Halving is epoch-counted, not wall-clock (partition-safe) At ~1 epoch per 10 minutes: 100,000 epochs ≈ 1.9 years

```
The theoretical ceiling is 2^64 µMHR, but it is never reached — tail emission
asymptotically approaches it. The initial reward of 10^12 µMHR/epoch yields ~1.5% of the
supply ceiling minted in the first halving period, providing strong bootstrap incentive.
Discrete halving every 100,000 epochs is epoch-counted (no clock synchronization needed)
and trivially computable via bit-shift on integer-only hardware.

The tail ensures ongoing proof-of-service rewards exist indefinitely, funding relay and
```

storage operators. In practice, lost keys (estimated 1–2% of supply annually) offset tail emission, keeping effective circulating supply roughly stable after year ~10.

Typical Costs

Operation	Cost
───────────	──────
Expected relay cost per packet	~5 μMHR
Relay lottery payout (on win)	~500 μMHR (5 μMHR ÷ 1/100 win probability)
Expected cost: 1 KB message, 5 hops	~75 μMHR (~3 packets × 5 μMHR × 5 hops)
1 hour of storage (1 MB)	~50 μMHR
1 minute of compute (contract execution)	~30–100 μMHR

The relay lottery pays out infrequently but in larger amounts. Expected value per packet is the same: `500 μMHR × 1/100 = 5 μMHR`. See [Stochastic Relay Rewards](#economics-payment-channels) for the full mechanism.

Economic Architecture

Mehr has a simple economic model: **free between friends, paid between strangers.**

Free Tier (Trust-Based)

- Traffic between [trusted peers](#economics-trust-neighborhoods) is **always free**
- No tokens, no channels, no settlements needed
- A local mesh where everyone trusts each other has **zero economic overhead**

Paid Tier (MHR)

- Traffic crossing trust boundaries triggers [stochastic relay rewards](#economics-payment-channels)
- Relay nodes earn MHR probabilistically — same expected income, far less overhead
- Settled via [CRDT ledger](#economics-crdt-ledger)

Genesis and Bootstrapping

The bootstrapping problem — needing MHR to use services, but needing to provide services to earn MHR — is solved by separating free-tier operation from the paid economy:

Free-Tier Operation (No MHR Required)

- **Trusted peer communication is always free** — no tokens needed

```
- **A local mesh works with zero tokens in circulation**
- The protocol is fully functional without any MHR — just limited to your trust network


<a id="economics-mhr-token--proof-of-service-mining-mhr-genesis"></a>


### Proof-of-Service Mining (MHR Genesis)

The [stochastic relay lottery](#economics-payment-channels) serves a dual purpose: it
determines who earns and how much, while the **funding source** depends on the economic
context:


1. **Minting (subsidy)**: Each epoch, the emission schedule determines how much new MHR is
minted. This is distributed proportionally to relay nodes based on their accumulated VRF
lottery wins during that epoch — proof that they actually forwarded packets. Minting
dominates during bootstrap and decays over time per the emission schedule.


2. **Channel debit (market)**: When a relay wins the lottery and has an open [payment
channel](#economics-payment-channels) with the upstream sender, the reward is debited from
that channel. The sender pays directly for routing. This becomes the dominant mechanism as
MHR enters circulation and channels become widespread.


Both mechanisms coexist. As the economy matures, channel-funded relay payments naturally
replace minting as the primary income source for relays, while the decaying emission
schedule ensures the transition is smooth.
```

Relay compensation per epoch: Epoch mint pool: $\max(10^{12} \gg (epoch / 100\_000), tail\_floor)$ → new supply created (not transferred from a pool) → halves every 100,000 epochs; floors at 0.1% annual inflation

Relay R's mint share: epoch_mint_pool × (R_wins / total_wins_in_epoch) → proportional to verified VRF lottery wins → a relay with 10% of the epoch's wins gets 10% of the mint pool

Channel revenue: sum of lottery wins debited from sender channels → direct payment, no new supply created

Total relay income = mint share + channel revenue

```
<a id="economics-mhr-token--bootstrap-sequence"></a>


### Bootstrap Sequence

1. Nodes form local meshes (free between trusted peers, no tokens)
2. Gateway nodes bridge to wider network
3. Non-trusted traffic triggers stochastic relay lottery (VRF-based)
4. Lottery wins accumulate as service proofs; epoch minting distributes MHR to relays
5. Relay nodes open payment channels and begin spending MHR on services
6. Senders with MHR fund relay costs via channel debits; minting share decreases
7. Market pricing emerges from supply/demand


<a id="economics-mhr-token--trust-based-credit"></a>
```

Trust-Based Credit

Trusted peers can [vouch for each other](#economics-trust-neighborhoods--trust-based-credit) by extending transitive credit. Each node configures the credit line it extends to its direct trusted peers (e.g., "I'll cover up to 1000 µMHR for Alice"). A friend-of-a-friend gets 10% of that direct limit — backed by the vouching peer's MHR balance. If a credited node defaults, the voucher absorbs the debt. This provides an on-ramp for new users without needing to earn MHR first.

Free direct communication works immediately with no tokens at all. MHR is only needed when your packets traverse untrusted infrastructure.

Why One Global Currency

MHR is a single global unit of account, not a per-community token. This is a deliberate design choice.

The Alternative: Per-Community Currencies

If each isolated community minted its own token, connecting two communities would require a currency exchange — someone to set an exchange rate, provide liquidity, and settle trades. On a mesh network of 50-500 nodes, there is not enough trading volume to sustain a functioning exchange market. The complexity (order books, matching, dispute resolution) vastly exceeds what constrained devices can support.

How One Currency Works Across Partitions

When two communities operate in isolation:

1. **Internally**: Both communities communicate free between trusted peers — no MHR needed
2. **Independently**: Each community mints MHR via proof-of-service, proportional to actual relay work. The [CRDT ledger](#economics-crdt-ledger) tracks balances independently on each side
3. **On reconnection**: The CRDT ledger merges automatically (CRDTs guarantee convergence). Both communities' MHR is valid because it was earned through real work, not printed arbitrarily

MHR derives its value from **labor** (relaying, storage, compute), not from community membership. One hour of relaying in Community A is roughly equivalent to one hour in Community B. Different hardware costs are reflected in **market pricing** — nodes set their own per-byte charges — not in separate currencies.

Price Discovery Without Fiat

> MHR has no fiat exchange rate by design. Its "purchasing power" floats based on supply and demand for services:

Abundant relay capacity + low demand → relay prices drop (in µMHR) Scarce relay capacity + high demand → relay prices rise (in µMHR)

Users don't need to know what 1 µMHR is worth in dollars. They only need to know: "Can I afford this service?" — and the answer is usually yes, because they earn MHR by providing services to others. The economy is circular, not pegged to an external reference.

Economic Design Goals

- **No speculation**: MHR is for purchasing services, not for trading. There is no fiat exchange rate by design.
- **No pre-mine**: All MHR enters circulation through proof-of-service
- **Hoarding-resistant**: MHR has no external exchange value, so accumulating it has no purpose beyond purchasing network services. Tail emission (0.1% annual) mildly dilutes idle holdings. Lost keys (~1-2% annually) permanently remove supply. The economic incentive is to spend MHR on services, not to sit on it.
- **Partition-safe**: The economic layer works correctly during network partitions and converges when they heal
- **Minimal overhead**: [Stochastic rewards](#economics-payment-channels) reduce economic bandwidth overhead by ~10x compared to per-packet payment
- **Communities first**: Trusted peer communication is free. The economic layer only activates at trust boundaries.

<div class="page-break"></div>

Stochastic Relay Rewards

Relay nodes are compensated through **probabilistic micropayments** rather than per-packet accounting. This dramatically reduces payment overhead on constrained radio links while providing the same expected income over time.

Why Not Per-Packet Payment?

Per-packet payment requires a channel state update for every batch of relayed packets. Even batched, this consumes significant bandwidth on LoRa links. The insight: relay rewards don't need to be deterministic — they can be probabilistic, like mining, achieving the same expected value with far less overhead.


```
## How Stochastic Rewards Work

Each relayed packet is checked against a **VRF-based lottery**. The relay computes a
Verifiable Random Function output over the packet, producing a deterministic but
unpredictable result that anyone can verify:
```

Relay reward lottery (VRF-based):

1. Relay computes: (vrf_output, vrf_proof) = VRF_prove(relay_private_key, packet_hash)

2. Check: vrf_output < difficulty_target

3. If win: reward = per_packet_cost × (1 / win_probability)

4. Expected value per packet = reward × probability = per_packet_cost ✓

5. Verification: VRF_verify(relay_public_key, packet_hash, vrf_output, vrf_proof)

```
**Why VRF, not a random nonce?** If the relay chose its own nonce, it could grind through
values until it found a winner for every packet, extracting the maximum reward every time.
The VRF produces exactly **one valid output** per (relay key, packet) pair — the relay
cannot influence the lottery outcome. The proof lets any party verify the result without
the relay's private key.

The VRF used is **ECVRF-ED25519-SHA512-TAI** ([RFC 9381](https://www.rfc-
editor.org/rfc/rfc9381)), which reuses the relay's existing Ed25519 keypair. VRF proof size
is 80 bytes, included only in winning lottery claims (not in every packet).

<a id="economics-payment-channels--example"></a>

### Example

| Parameter | Value |
|───────────|───────|
| Per-packet relay cost | 5 µMHR |
| Win probability | 1/100 |
| Reward per win | 500 µMHR |
| Expected value per packet | 5 µMHR (same) |
| Channel updates needed | 1 per ~100 packets (vs. every batch) |

A relay handling 10 packets/minute triggers a channel update approximately once every 10
minutes — a **10x reduction** in payment overhead compared to per-minute batching.

<a id="economics-payment-channels--adaptive-difficulty"></a>

### Adaptive Difficulty

The win probability adjusts based on traffic volume. Each relay computes its own difficulty
locally based on its observed traffic rate — no global synchronization needed:
```

Difficulty adjustment: target_updates_per_minute = 0.1 (one channel update per ~10 minutes) observed_packets_per_minute = trailing 5-minute moving average

win_probability = target_updates_per_minute / observed_packets_per_minute win_probability = clamp(win_probability, 1/10000, 1/5) // bounds

difficulty_target = MAX_VRF_OUTPUT × win_probability

Traffic tiers (approximate): High-traffic links (>100 packets/min): ~1/1000 probability, larger rewards Medium-traffic links (10-100 packets/min): ~1/100 probability Low-traffic links (<10 packets/min): ~1/10 probability, smaller rewards

Reward on win = per_packet_cost × (1 / win_probability) Expected value per packet = per_packet_cost (always, regardless of difficulty)

```
Low-traffic links use higher win probability to reduce variance — a relay handling only a
few packets per hour will still receive rewards regularly. The difficulty is computed
independently by each relay per-link, so different links on the same node may have
different difficulties.

<a id="economics-payment-channels--bilateral-payment-channels"></a>

## Bilateral Payment Channels

Rewards are settled through bilateral channels between direct neighbors. Unlike Lightning-
style multi-hop payment routing, Mehr uses simple per-hop channels:

- Only two parties need to coordinate
- Both parties are direct neighbors (by definition)
- No global coordination needed

<a id="economics-payment-channels--channel-state"></a>

### Channel State
```

ChannelState { channel_id: [u8; 16], // truncated Blake3 hash (16 bytes) party_a: [u8; 16], // destination hash (16 bytes) party_b: [u8; 16], // destination hash (16 bytes) balance_a: u64, // party A's current balance (8 bytes) balance_b: u64, // party B's current balance (8 bytes) sequence: u64, // monotonically increasing (8 bytes) sig_a: Ed25519Signature, // party A's signature (64 bytes) sig_b: Ed25519Signature, // party B's signature (64 bytes) } // Total: 16 + 16 + 16 + 8 + 8 + 8 + 64 + 64 = 200 bytes

```
<a id="economics-payment-channels--channel-lifecycle"></a>

### Channel Lifecycle
```

```
1. **Open**: Both parties agree on initial balances. Both sign the opening state (`sequence
= 0`).
2. **Update**: On each lottery win, the balance shifts by the reward amount and `sequence`
increments by 1. Both parties sign the updated state. Channel updates are infrequent — only
triggered by wins.
3. **Settle**: Either party can request settlement. Both sign a `SettlementRecord` whose
`final_sequence` matches the current channel `sequence`. The record is gossiped to the
network and applied to the [CRDT ledger](#economics-crdt-ledger). The channel remains open
after settlement — subsequent lottery wins continue from the settled point.
4. **Dispute**: If one party submits an old state, the counterparty can submit a higher-
sequence state within a **2,880 gossip round challenge window** (~48 hours at 60-second
rounds). The higher sequence always wins.
5. **Abandonment**: If a channel has no updates for **4 epochs**, either party can
unilaterally close with the last mutually-signed state. This prevents permanent fund
lockup.

<a id="economics-payment-channels--settlement-timing"></a>

### Settlement Timing

Lottery wins accumulate as local channel state updates (balance shifts + sequence
increments). Settlements to the CRDT ledger are **not** created per-win — they are created
when either party decides to finalize:
```

Settlement triggers:

- Either party requests cooperative settlement

- Channel dispute (one party publishes an old state)

- Channel abandonment (4 epochs of inactivity)

- Periodic finalization (recommended: once per epoch)

Between settlements, interim balances are NOT gossiped. Only the two parties track the current ChannelState locally.

```
This preserves the stochastic lottery's bandwidth savings: a relay handling 10
packets/minute triggers ~6 local channel updates per hour, but settlements to the CRDT
ledger happen much less frequently.

<a id="economics-payment-channels--sequence-number-semantics"></a>

### Sequence Number Semantics

The `sequence` field is a monotonically increasing version number:

- Each update increments `sequence` by 1; both parties must sign the same sequence
- A `SettlementRecord` references `final_sequence` — the sequence of the state being
settled
- After settlement, the channel continues with `sequence > final_sequence`
- Dispute resolution: higher `sequence` always wins, regardless of settlement history
```

- Replay protection: the CRDT ledger rejects settlements where `final_sequence` is not greater than the last settled sequence for the same `channel_id`

Multi-Hop Payment

When Alice sends a packet through Bob → Carol → Dave, each relay independently runs the VRF lottery:

Alice ——→ Bob ——→ Carol ——→ Dave │ │ lottery? lottery?

A lottery win triggers compensation through one or both mechanisms:

1. **Channel debit** (if a channel exists with the upstream sender): Bob's win debits Alice's channel with Bob; Carol's win debits Bob's channel with Carol. This is the steady-state mechanism once MHR is circulating.
2. **Mining proof** (always): The VRF proof is accumulated as a service proof entitling the relay to a share of the epoch's [minting reward](#economics-mhr-token--proof-of-service-mining-mhr-genesis). This is the dominant income source during bootstrap and provides a baseline subsidy that decays over time.

Most packets trigger no channel update at all. Each hop is independent — no end-to-end payment coordination.

Efficiency on Constrained Links

Metric	Value
————————	————————
State update size	200 bytes
Average updates per hour (1/100 prob, 10 pkts/min)	~6
Bandwidth overhead at 1 kbps LoRa	~0.3%
Compared to per-minute batching	**~8x reduction**

The stochastic model fits within [Tier 2 (economic)](#protocol-network-protocol--bandwidth-budget) of the gossip bandwidth budget even on the most constrained links.

Trusted Peers: Free Relay

Nodes relay traffic for [trusted peers](#economics-trust-neighborhoods) for free — no lottery, no channel updates. The stochastic reward system only activates for traffic between non-trusted nodes. This mirrors the real world: you help your neighbors for free, but charge strangers for using your infrastructure.

<div class="page-break"></div>

```
<a id="economics-crdt-ledger"></a>

# CRDT Ledger

The global balance sheet in Mehr is a CRDT-based distributed ledger. Not a blockchain. No
consensus protocol. No mining. CRDTs (Conflict-free Replicated Data Types) provide
automatic, deterministic convergence without coordination — exactly what a partition-
tolerant network requires.


<a id="economics-crdt-ledger--why-not-a-blockchain"></a>

## Why Not a Blockchain?

Blockchains require global consensus: all nodes must agree on the order of transactions.
This is fundamentally incompatible with Mehr's partition tolerance requirement. When a
village mesh is disconnected from the wider network for days or weeks, it must still
process payments internally. CRDTs make this possible.


<a id="economics-crdt-ledger--account-state"></a>

## Account State
```

AccountState { node_id: NodeID, total_earned: GCounter, // grow-only, per-node entries, merge = pointwise max total_spent: GCounter, // grow-only, same structure // Balance = earned - spent (derived, never stored) settlements: GSet, // dedup set }

```
<a id="economics-crdt-ledger--how-gcounters-work"></a>

### How GCounters Work

A GCounter (grow-only counter) is a CRDT that can only increase. Each node maintains its
own entry, and merging takes the pointwise maximum:

- Node A says "Node X has earned 100" and Node B says "Node X has earned 150"
- Merge result: "Node X has earned 150" (the higher value wins)
- This works regardless of the order updates arrive

Balance is always derived: `balance = total_earned - total_spent`. It is never stored
directly.


<a id="economics-crdt-ledger--settlement-flow"></a>

## Settlement Flow
```

SettlementRecord { channel_id: [u8; 16], party_a: [u8; 16], party_b: [u8; 16], amount_a_to_b: i64, // net transfer (negative = B pays A) final_sequence: u64, // channel state sequence at settlement sig_a: Ed25519Signature, sig_b: Ed25519Signature, } // settlement_hash = Blake3(channel_id ||

party_a || party_b || amount || sequence) // Signatures are over the settlement_hash (sign-then-hash, not hash-then-sign)

Settlement flow:

1. Alice and Bob settle their payment channel (SettlementRecord signed by both)

2. SettlementRecord is gossiped to the network

3. Each receiving node validates:

   ○ Both signatures verify against the settlement_hash

   ○ settlement_hash is not already in the GSet (dedup)

   ○ Neither party's derived balance goes negative after applying

   ○ If any check fails: silently drop (do not gossip)

4. If valid and new:

   ○ Increment party_a's spent / party_b's earned (or vice versa)

   ○ Add settlement_hash to GSet

   ○ Gossip forward to neighbors

5. Convergence: O(log N) gossip rounds

```
Settlement validation is performed by **every node** that receives the record. This is
cheap (two Ed25519 signature verifications + one Blake3 hash + one GSet lookup) and ensures
no node relies on a single validator. Invalid settlements are dropped silently — no
penalty, no gossip.


<a id="economics-crdt-ledger--gossip-bandwidth"></a>


### Gossip Bandwidth

With [stochastic relay rewards](#economics-payment-channels), settlements happen far less
frequently than under per-packet payment — channel updates only trigger on lottery wins.
This dramatically reduces the volume of settlement records the CRDT ledger must gossip.

- Baseline gossip: proportional to settlement frequency (~100–200 bytes per settlement)
- On constrained links (< 10 kbps): batching interval increases, reducing overhead further
- Fits within **Tier 2 (economic)** of the [gossip bandwidth budget](#protocol-network-
protocol--bandwidth-budget)


<a id="economics-crdt-ledger--double-spend-prevention"></a>


## Double-Spend Prevention

Double-spend prevention is **probabilistic, not perfect**. Perfect prevention requires
global consensus, which contradicts partition tolerance. Mehr mitigates double-spending
through multiple layers:

1. **Channel deposits**: Both parties must have visible balance to open a channel
```

2. **Credit limits**: Based on locally-known balance
3. **Reputation staking**: Long-lived nodes get higher credit limits
4. **Fraud detection**: Overdrafts are flagged network-wide; the offending node is blacklisted
5. **Economic disincentive**: For micropayments, blacklisting makes cheating unprofitable — the cost of losing your identity and accumulated reputation exceeds any single double-spend gain

Partition Minting and Supply Convergence

When the network is partitioned, each partition independently runs the emission schedule and mints MHR proportional to local relay work. On merge, the GCounter merge (pointwise max per account) preserves individual balance correctness — no one loses earned MHR. However, **total minted supply across all partitions exceeds what a single-partition emission schedule would have produced.**

Example: Epoch 5 emission schedule: 1000 MHR total Partition A (60% of nodes): mints 1000 MHR to its relays Partition B (40% of nodes): mints 1000 MHR to its relays On merge: total minted in epoch 5 = 2000 MHR (not 1000)

This is an accepted tradeoff of partition tolerance — the alternative (coordinated minting) requires global consensus, which is incompatible with the design. The overminting is bounded:

1. **Proportional to partition count**: Two partitions produce at most 2x; three produce at most 3x. Prolonged fragmentation into many partitions is rare in practice.
2. **Detectable on merge**: When partitions heal, nodes can observe that multiple epoch proposals exist for the same epoch number. Post-merge epochs resume normal single-emission-schedule minting.
3. **Self-correcting over time**: The emission schedule decays geometrically. A one-time overmint during a partition is a fixed quantity that becomes negligible relative to total supply. The asymptotic ceiling is unchanged — it is just approached slightly faster.
4. **Offset by lost keys**: The estimated 1-2% annual key loss rate dwarfs partition minting overshoot in most scenarios.

The protocol does not attempt to "claw back" overminted supply. The cost of the mechanism (requiring consensus) exceeds the cost of the problem (minor temporary supply inflation during rare partitions).

Relay Compensation Tracking

Relay minting rewards are computed during epoch finalization. Each relay accumulates VRF lottery win proofs during the epoch and includes them in its epoch acknowledgment:

RelayWinSummary { relay_id: NodeID, win_count: u32, // number of VRF lottery wins this epoch sample_proofs: Vec, // subset of proofs (up to 10) for spot-checking total_wins_hash: Blake3Hash, //

Blake3 of all win proofs (verifiable if challenged) }

```
The epoch proposer aggregates win summaries from gossip and includes total win counts in
the epoch snapshot. Mint share for each relay is `epoch_reward × (relay_wins /
total_wins)`. Full proof sets are not gossiped (too large) — only summaries with spot-check
samples. Any node can challenge a relay's win count during the 4-epoch grace period by
requesting the full proof set. Fraudulent claims result in the relay's minting share being
redistributed and the relay's reputation being penalized.


<a id="economics-crdt-ledger--epoch-compaction"></a>


## Epoch Compaction


The settlement GSet grows without bound. The Epoch Checkpoint Protocol solves this by
periodically snapshotting the ledger state.
```

Epoch { epoch_number: u64, timestamp: u64,

```
// Frozen account balances at this epoch
account_snapshot: Map<NodeID, (total_earned, total_spent)>,

// Bloom filter of ALL settlement hashes included
included_settlements: BloomFilter,

// Active set: defines the 67% threshold denominator
active_set_hash: Blake3Hash,     // hash of sorted NodeIDs active in last 2 epochs
active_set_size: u32,            // number of nodes in the active set

// Acknowledgment tracking
ack_count: u32,
ack_threshold: u32,              // 67% of active_set_size
status: enum { Proposed, Active, Finalized, Archived },
```

}

```
<a id="economics-crdt-ledger--epoch-triggers"></a>


### Epoch Triggers


An epoch is triggered when **any** of these conditions is met:


| Trigger | Threshold | Purpose |
|─────────|───────────|─────────|
| **Settlement count** | ≥ 10,000 batches | Standard trigger for large meshes |
| **GSet memory** | ≥ 500 KB | Protects constrained devices (ESP32 has ~520 KB usable RAM)
|
| **Small partition** | ≥ max(200, active_set_size × 10) settlements AND ≥ 1,000 gossip
rounds since last epoch | Prevents stagnation in small partitions |
```

```
The small-partition trigger ensures a 20-node village doesn't wait months for an epoch. At
200 settlements (the minimum), the GSet is ~6.4 KB — well within ESP32 capacity. The 1,000
gossip round floor (roughly 17 hours at 60-second intervals) prevents epochs from firing
too rapidly in tiny partitions with bursty activity.


<a id="economics-crdt-ledger--epoch-proposer-selection"></a>


### Epoch Proposer Selection


Eligibility requirements adapt to partition size:


1. The node has processed ≥ min(10,000, current epoch trigger threshold) settlement batches
since the last epoch
2. The node has direct links to ≥ min(3, active_set_size / 2) active nodes
3. No other epoch proposal for this `epoch_number` has been seen


In a 20-node partition, a node needs only 3 direct links (not 10) and 200 processed
settlements (not 10,000) to propose.


**Conflict resolution**: If multiple proposals for the same `epoch_number` arrive, nodes
ACK the one with the **highest settlement count** (most complete state). Ties broken by
lowest proposer `destination_hash`.


**Active set divergence** (post-partition): Two partitions may propose epochs with
different `active_set_hash` values because they've seen different settlement participants.
Resolution:
```

Active set conflict handling:

1. If your local settlement count is within 5% of the proposal's count: ACK the proposal's
   active_set_hash (defer to proposer — close enough)
2. If your local settlement count exceeds the proposal's by >5%: NAK the proposal. Wait 3 gossip
   rounds for further convergence, then propose your own epoch if no better proposal arrives
3. After partition merge: the epoch with the highest settlement count is accepted by all nodes.
   The losing partition's active set members that were missing from the winning proposal are
   included in the NEXT epoch's active set (no settlements are lost — they are applied on top of
   the winning snapshot during the verification window)

```
Epoch proposals are rate-limited to one per node per epoch period. Proposals that don't
meet eligibility are silently ignored.


<a id="economics-crdt-ledger--epoch-lifecycle"></a>


### Epoch Lifecycle


1. **Propose**: An eligible node proposes a new epoch with a snapshot of current state. The
proposal includes an `active_set_hash` — a Blake3 hash of the sorted list of NodeIDs in the
active set, as observed by the proposer. This fixes the denominator for the 67% threshold.
```

Active set definition: A node is in the active set if it appears as `party_a` or `party_b` in at least one `SettlementRecord` within the last 2 epochs. Relay-only nodes (that relay packets but never settle channels) are not in the active set — they participate in the economy via mining proofs, not via epoch consensus. This keeps the active set small and the 67% threshold meaningful.

2. **Acknowledge**: Nodes compare against their local state. If they've seen the same or more settlements, they ACK. If they have unseen settlements, they gossip those first. A node ACKs the proposal's `active_set_hash` — even if its own view differs slightly, it agrees to use the proposer's set as the threshold denominator for this epoch.

3. **Activate**: At 67% acknowledgment (of the active set defined in the proposal), the epoch becomes active. Nodes can discard individual settlement records and use only the bloom filter for dedup. If a significant fraction of nodes reject the active set (NAK), the proposer must re-propose with an updated set after further gossip convergence.

4. **Verification window**: During the grace period (4 epochs after activation), any node can submit a **settlement proof** — the full `SettlementRecord` — for any settlement it believes was missed. If the settlement is valid (signatures check) and NOT in the epoch's bloom filter, it is applied on top of the snapshot.

5. **Finalize**: After the grace period, previous epoch data is fully discarded. The bloom filter is the final word.

Late Arrivals After Compaction

When a node reconnects after an epoch has been compacted, it checks its unprocessed settlements against the epoch's bloom filter:
- **Present in filter**: Already counted, discard
- **Absent from filter**: New settlement, apply on top of snapshot. If within the verification window, submit as a settlement proof.

Bloom Filter Sizing

Data	Size
1M settlement hashes (raw)	~32 MB
Bloom filter (0.01% false positive rate)	~2.4 MB
Target epoch frequency	~10,000 settlement batches
Per-node storage target	Under 5 MB

The false positive rate is set to **0.01% (1 in 10,000)** rather than 1%, because false positives cause legitimate settlements to be silently treated as duplicates. At 0.01%, the expected loss is negligible (~1 settlement per 10,000), and the verification window provides a recovery mechanism for any that are caught.

Construction: The bloom filter uses `k = 13` hash functions derived from Blake3:

Bloom filter hash construction: For each settlement_hash and index i in [0, k): h_i = Blake3(settlement_hash || i as u8) truncated to 32 bits bit_position = h_i mod m (where m = total bits in filter)

Bits per element: $m/n = -\ln(p) / (\ln 2)^2 \approx 19.2$ bits at $p = 0.0001$ k = $-\log_2(p) \approx 13.3$, rounded to 13

For 10,000 settlements: m = 192,000 bits = 24 KB For 1M settlements: m = 19.2M bits $\approx$ 2.4 MB

The Merkle tree over the account snapshot also uses Blake3 (consistent with all content
hashing in Mehr). Leaf nodes are `Blake3(NodeID || total_earned || total_spent)`, and
internal nodes are `Blake3(left_child || right_child)`.

Critical retention rule: Both parties to a settlement **must retain the full
`SettlementRecord`** until the epoch's verification window closes (4 epochs after
activation). If both parties discard the record after epoch activation (believing it was
included) and a bloom filter false positive caused it to be missed, the settlement would be
permanently lost. During the verification window, each party independently checks that its
settlements are reflected in the snapshot; if any are missing, it submits a settlement
proof. Only after the window closes may the full record be discarded.

Snapshot Scaling

At 1M+ nodes, the flat `account_snapshot` is ~32 MB — too large for constrained devices.
The solution is a **Merkle-tree snapshot** with sparse views.

Full snapshot (backbone/gateway nodes only): The account snapshot is stored as a sorted
Merkle tree keyed by NodeID. Only nodes that participate in epoch consensus need the full
tree. At 1M nodes and 32 bytes per entry, this is ~32 MB — feasible for nodes with SSDs.

Sparse snapshot (everyone else): Constrained devices store only:
- Their own balance
- Balances of direct channel partners
- Balances of trust graph neighbors (Ring 0-2)
- The Merkle root of the full snapshot

For a typical node with ~50 relevant accounts: 50 × 32 bytes = 1.6 KB.

On-demand balance verification: When a constrained node needs a balance it doesn't have
locally (e.g., to extend credit to a new node), it requests a Merkle proof from any capable
peer:

BalanceProof { node_id: NodeID, total_earned: u64, total_spent: u64, merkle_siblings: Vec, // path
from leaf to root epoch_number: u64, } // Size: ~640 bytes for 1M nodes (20 tree levels × 32-byte
hashes)

The constrained node verifies the proof against the Merkle root it already has. This proves
the balance is in the snapshot without storing the full 32 MB.

Constrained Node Epoch Summary

> LoRa nodes and other constrained devices don't participate in epoch consensus. They receive
> a compact summary from their nearest capable peer:

EpochSummary { epoch_number: u64, merkle_root: Blake3Hash, // root of full account snapshot proposer_id: NodeID, // who proposed this epoch proposer_sig: Ed25519Signature, // signature over (epoch_number || merkle_root) my_balance: (u64, u64), // (total_earned, total_spent) partner_balances: Vec<(NodeID, u64, u64)>, // channel partners + trust neighbors bloom_segment: BloomFilter, // relevant portion of settlement bloom }

```
Typical size: under 5 KB for a node with 20-30 channel partners.

<a id="economics-crdt-ledger--merkle-root-trust"></a>

### Merkle Root Trust

The `merkle_root` is the anchor for all balance verification on constrained nodes. To
prevent a malicious relay from feeding a fake root:
```

Merkle root acceptance:

- If the source is a trusted peer (in trust graph): accept immediately (trusted peers have economic skin in the game)
- If the source is untrusted: verify proposer_sig against proposer_id, then confirm with at least 1 additional independent peer in Ring 0/1 (2-source quorum, same as DHT mutable object verification)
- Cold start (no prior epoch): query 2+ peers and accept majority agreement
- Retention: keep roots for the last 4 epochs (grace period for balance proofs)

```
The proposer's signature prevents trivial forgery - an attacker must either compromise the
proposer's key or control the majority of a node's Ring 0 peers.



<div class="page-break"></div>


<a id="economics-trust-neighborhoods"></a>

# Trust & Neighborhoods

Communities in Mehr are **emergent, not declared**. There are no admin-created "zones" to
join, no governance to negotiate, no artificial boundaries between groups. Instead,
communities form naturally from a trust graph - just like in the real world.
```

```
<a id="economics-trust-neighborhoods--the-trust-graph"></a>

## The Trust Graph

Each node maintains a set of trusted peers:
```

TrustConfig { // Peers I trust — relay their traffic for free trusted_peers: Set,

```
// What I charge non-trusted traffic
default_cost_per_byte: u64,

// Per-peer cost overrides (discount for friends-of-friends, etc.)
cost_overrides: Map<NodeID, u64>,

// Optional self-assigned label (purely informational)
community_label: Option<String>,     // e.g., "portland-mesh"
```

}

```
**Adding a trusted peer is the only social action in Mehr.** Everything else — free local
communication, community identity, credit lines — emerges from the trust graph.

Trust relationships are **asymmetric and revocable at any time**. Removing a node from
`trusted_peers` immediately ends free relay for that node and downgrades any stored data
from "trusted peer" to normal priority in the [garbage collection policy](#services-mhr-
store--garbage-collection). Cost overrides are unidirectional — they apply to outbound
traffic pricing from the configuring node only.

<a id="economics-trust-neighborhoods--how-communities-emerge"></a>

## How Communities Emerge

When a cluster of nodes all trust each other, a **neighborhood** forms:
```

[Alice] ←——trust——→ [Bob] ←——trust——→ [Carol] │ │ │ └———trust———→ [Dave] ←——trust———┘

```
Alice, Bob, Carol, and Dave are a neighborhood. No one "created" it. No one "joined" it. It
exists because they trust each other.

<a id="economics-trust-neighborhoods--properties"></a>

### Properties

- **No admin**: Nobody runs the neighborhood. It has no keys, no governance, no admission
policy.
- **No fragmentation**: The trust graph is continuous. There are no hard boundaries between
```

```
    communities — neighborhoods overlap naturally when people have friends in multiple
    clusters.
    - **No UX burden**: You just mark people as trusted. The same action you'd take when adding
    a contact.
    - **Fully decentralized**: There is nothing to attack, take over, or censor.


    <a id="economics-trust-neighborhoods--free-local-communication"></a>


    ## Free Local Communication

    Traffic between trusted peers is **always free**. A relay node checks its trust list:
```

Relay decision: if sender is trusted AND destination is trusted: relay for free (no lottery, no channel update) else if sender is trusted: relay for free (helping a friend send outbound) else: relay with stochastic reward lottery

```
    Note the asymmetry: a relay helps its trusted peers **send** traffic for free, but does not
    relay free traffic from strangers just because the destination is trusted. Without this
    rule, an untrusted node could route unlimited free traffic through you to any of your
    trusted peers, shifting relay costs onto you without compensation.

    This means a village mesh where everyone trusts each other operates with **zero economic
    overhead** — no tokens, no channels, no settlements. The economic layer only activates for
    traffic crossing trust boundaries.

    <a id="economics-trust-neighborhoods--trust-based-credit"></a>


    ## Trust-Based Credit

    When a node needs MHR (e.g., to reach beyond its trusted neighborhood), its trusted peers
    can vouch for it:
```

Transitive credit: Direct trust: full credit line (set by trusting peer) Friend-of-friend (2 hops): 10% of direct credit line 3+ hops of trust: no credit (too diluted)

If a credited node defaults, the vouching peer absorbs the debt. This makes trust economically meaningful — you only trust people you'd lend to.

```
    The credit line is **rate-limited** for safety:

    | Trust Distance | Max Credit | Rate Limit |
    ┝━━━━━━━━━━━━━━━━┿━━━━━━━━━━━━━┿━━━━━━━━━━━━┥
    | Direct trusted peer | Configurable by trusting node | Per-epoch (configurable) |
    | Friend-of-friend | 10% of direct limit | Per-epoch, per friend-of-friend |
    | Beyond 2 hops | None | N/A |
```

Credit accounting: Each trusting node tracks outstanding credit per grantee:

CreditState { grantee: NodeID, credit_limit: u64, // max outstanding μMHR outstanding: u64, // currently extended granted_this_epoch: u64, // epoch-scoped rate limit last_grant_epoch: u64, // for epoch-boundary reset }

Rules: - Direct peers: each gets a separate credit_limit (set in TrustConfig) - Friend-of-friend: each gets 10% of the vouching peer's direct limit, tracked independently per grantee - granted_this_epoch resets to 0 at each epoch boundary - Outstanding credit that exceeds limit: no new grants until repaid - Default handling: vouching peer's outstanding balance increases by the defaulted amount (absorbs debt); grantee is flagged

```
<a id="economics-trust-neighborhoods--community-labels"></a>

## Community Labels

Nodes can optionally self-assign a `community_label` string:
```

Alice sets: community_label = "portland-mesh" Bob sets: community_label = "portland-mesh" Carol sets: community_label = "portland-mesh"

```
This label is:
- **Self-assigned** — no one approves it, no authority enforces uniqueness
- **Not authoritative** — it carries no protocol-level privileges (it cannot grant access,
waive fees, or modify trust)
- **Not unique** — multiple disjoint clusters can use the same label
- **Used by services** — [MHR-Name](#applications-naming) scopes human-readable names by
label, [MHR-Pub](#services-mhr-pub) supports `Neighborhood(label)` subscriptions, and [MHR-
DHT](#services-mhr-dht--neighborhood-scoped-dht) uses labels for content scoping
- **Useful for discovery** — "find nodes labeled 'portland-mesh' near me"

Community labels enable human-readable naming and discovery without any of the governance
overhead of explicit zones.

<a id="economics-trust-neighborhoods--comparison-zones-vs-trust-neighborhoods"></a>

## Comparison: Zones vs. Trust Neighborhoods

| Aspect | Explicit Zones (old) | Trust Neighborhoods (current) |
├───────────┼─────────────────────────┼───────────────────────────────┤
| Creation | Someone creates a zone | Emerges from mutual trust |
| Joining | Request + approval | Mark someone as trusted |
| Governance | Admin keys, voting | None needed |
| Boundaries | Hard, declared | Soft, overlapping |
| Free communication | Within zone boundary | Between any trusted peers |
| Naming | `alice@zone-name` | `alice@community-label` |
| Sybil resistance | Admission policy | Trust is social and economic (you absorb their
debts) |
| UX complexity | Create, join, configure | Add contacts |
```

```
<a id="economics-trust-neighborhoods--sybil-mitigation"></a>

## Sybil Mitigation

The trust graph provides natural Sybil resistance:

1. **Trust has economic cost**: Vouching for a node means absorbing its potential debts.
Sybil identities with no real relationships get no credit.
2. **Rate limiting**: Even if a malicious node gains one trust relationship, transitive
credit is capped at 10% per hop.
3. **Reputation**: A node's usefulness as a relay/service provider is what earns trust over
time. Creating many identities dilutes reputation rather than concentrating it.
4. **Local detection**: A node's trust graph is visible to its peers. A node trusting an
unusual number of new, unproven identities is itself suspicious.

<a id="economics-trust-neighborhoods--real-world-parallels"></a>

## Real-World Parallels

| Real World | Mehr Trust |
|────────────┼────────────|
| Talk to your neighbor for free | Free relay between trusted peers |
| Lend money to a friend | Transitive credit via trust |
| Wouldn't lend to a stranger | No credit without trust chain |
| Communities aren't corporations | Neighborhoods have no admin |
| You belong to multiple groups | Trust graph is continuous, not partitioned |
| Reputation builds over time | Trust earned through reliable service |




<div class="page-break"></div>



<a id="economics-real-world-impact"></a>

# Real-World Economics

Mehr doesn't exist in a vacuum. It interacts with the existing internet economy — ISPs,
cloud providers, and the people who pay for connectivity today. This page examines what
happens when a mesh network meets existing infrastructure economics.

<a id="economics-real-world-impact--the-apartment-building-scenario"></a>

## The Apartment Building Scenario

Consider a typical apartment building in Denmark:
```

Current model: 50 apartments × 200 kr/month = 10,000 kr/month to ISPs Average utilization per connection: <5% Each apartment has its own router, its own subscription, its own bill

Mehr model: 2-3 gateway nodes with internet subscriptions = 400-600 kr/month to ISPs Gateway nodes share internet via WiFi mesh to all 50 apartments Other 47 apartments pay gateway operators in MHR ISP revenue from building: drops ~94%

```
<a id="economics-real-world-impact--why-this-works"></a>

### Why This Works

Residential internet connections are massively over-provisioned. A 1 Gbps connection
serving one household averages under 50 Mbps actual usage, and most of that is concentrated
in evening hours. The infrastructure exists to handle peak load, but sits idle the vast
majority of the time.

With Mehr, 2-3 well-placed gateway nodes with good internet connections can serve an entire
building. The gateway operators earn MHR from the other residents — effectively becoming
micro-ISPs within their building.

<a id="economics-real-world-impact--what-happens-to-isps"></a>

### What Happens to ISPs?

**Mehr doesn't kill ISPs. It restructures them.**

| Today | With Mehr |
|───────|───────────|
| ISPs sell per-household subscriptions | ISPs sell per-building or per-community
connections |
| Revenue depends on subscriber count | Revenue depends on bandwidth sold |
| Last-mile infrastructure to every apartment | Last-mile to building entry point; mesh
handles internal distribution |
| ISPs handle per-customer support | Gateway operators handle local support |
| ISPs own the customer relationship | The community owns its own network |

The key insight: **ISPs already don't want to be last-mile providers.** Last-mile
infrastructure (running cable to every apartment) is their most expensive, lowest-margin
business. Mehr handles last-mile distribution through the mesh, letting ISPs focus on what
they're actually good at — backbone transit and peering.

ISPs would likely respond by:
1. Offering **building connections** — one fat pipe per building at a higher bandwidth tier
2. Pricing by **bandwidth consumed**, not by connection count
3. Becoming **backbone providers** to Mehr gateway operators
4. Running their own **Mehr backbone nodes** to earn routing fees

<a id="economics-real-world-impact--the-math-for-gateway-operators"></a>

### The Math for Gateway Operators
```

Gateway operator costs: Internet subscription: 200 kr/month Hardware (Pi 4 + LoRa + modem): ~~300 kr one-time~~ (25 kr/month amortized over 1 year) Total: ~225 kr/month

Gateway operator revenue: ~47 apartments paying for shared internet If each pays 50 kr/month equivalent in MHR: 2,350 kr/month After subtracting costs: ~2,125 kr/month profit

Resident savings: Was paying: 200 kr/month Now paying: ~50 kr/month in MHR Saving: 150 kr/month (75% reduction)

```
Both sides win. Gateway operators earn significant income from hardware they'd have anyway.
Residents save money. The only loser is the ISP's per-household billing model — which was
always an artifact of last-mile economics, not actual cost.


<a id="economics-real-world-impact--how-you-earn-on-mehr"></a>


## How You Earn on Mehr


Every node earns proportionally to the value it provides:


<a id="economics-real-world-impact--relay-earnings"></a>


### Relay Earnings


The simplest way to earn. Any node that forwards packets for non-trusted traffic
participates in the [stochastic relay lottery](#economics-payment-channels). More traffic
through your node = more lottery wins = more MHR.
```

Relay earnings estimate (at ~5 µMHR expected reward per packet): Minimal relay (ESP32 + LoRa): ~5,000–50,000 µMHR/month → ~30-300 packets/day, zero operating cost (solar powered)

Community bridge (Pi Zero + WiFi): ~50,000–500,000 µMHR/month → Bridges LoRa to WiFi, moderate traffic

Gateway (Pi 4 + cellular): ~500,000–5,000,000 µMHR/month → Internet uplink, high-value traffic

Backbone (mini PC + directional WiFi): 5,000,000+ µMHR/month → High-throughput transit between mesh segments

```
<a id="economics-real-world-impact--storage-earnings"></a>


### Storage Earnings


Nodes with disk space earn by storing data for the network via [MHR-Store](#services-mhr-
store):

- Store popular content that others request frequently
- Host replicated data for availability
- Cache content for faster local access

<a id="economics-real-world-impact--compute-earnings"></a>
```

Compute Earnings

Nodes with CPUs or GPUs earn by executing contracts and offering inference via [MHR-Compute](#services-mhr-compute):

- Run MHR-Byte contracts for constrained nodes
- Offer WASM execution for heavier workloads
- Provide ML inference (speech-to-text, translation, image generation)

Gateway Earnings

The highest-value service. Internet gateway operators earn from:

- HTTP proxy services
- DNS relay
- Bridge traffic between mesh and internet
- All of the above, plus relay/storage/compute earnings

What Makes a Node Valuable?

The marketplace naturally prices capabilities based on scarcity and utility:

Factor	Effect on Earnings
Connectivity	More links = more routing traffic = more relay earnings
Location	Strategic position (bridge between clusters) = higher routing value
Uptime	24/7 availability = more agreements, better reputation
Storage capacity	More disk = more storage contracts
Compute power	GPU = high-value inference contracts
Internet access	Gateway capability = premium pricing
Trust network size	More trusted peers = higher credit lines, more routing

Broader Economic Implications

For Developing Regions

In areas with no ISP at all, Mehr enables community networks from scratch:

1. One satellite or cellular connection serves an entire village via mesh
2. The gateway operator earns from the community
3. Community members earn by relaying for each other and for outsiders
4. Economic activity within the mesh is free (trusted peers)
5. External connectivity costs are shared, not per-household

For Urban Areas

In cities where internet is available but expensive:

1. Shared internet connections reduce per-household costs by 50-75%
2. Local services (storage, compute, messaging) run on the mesh with no cloud dependency
3. Community infrastructure becomes an income source, not a cost center

For Censorship-Resistant Communication

When governments control the internet:

1. The mesh operates independently of ISP infrastructure
2. Even if internet gateways are shut down, local communication continues
3. Gateway nodes with satellite uplinks or VPN tunnels become high-value — and the market prices them accordingly

<div class="page-break"></div>

Layer 4: Capability Marketplace

The capability marketplace is the unifying abstraction of Mehr. Every node advertises what it can do. Every node can request capabilities it lacks. The marketplace matches supply and demand through local, bilateral negotiation — no central coordinator.

This is the layer that makes Mehr a **distributed computer** rather than just a network.

The Unifying Abstraction

In Mehr, there are no fixed node roles. Instead:

- A node with a LoRa radio and solar panel advertises: *"I can relay packets 24/7"*
- A node with a GPU advertises: *"I can run Whisper speech-to-text"*
- A node with an SSD advertises: *"I can store 100 GB of data"*
- A node with a cellular modem advertises: *"I can route to the internet"*

Each of these is a **capability** — discoverable, negotiable, verifiable, and payable.

Capability Advertisement

Every node broadcasts its capabilities to the network:

NodeCapabilities { node_id: NodeID, timestamp: Timestamp, signature: Ed25519Signature,

```
    // ── CONNECTIVITY ──
    interfaces: [{
        medium: TransportType,
        bandwidth_bps: u64,
        latency_ms: u32,
        reliability: u8,              // 0-255 (avoids FP on constrained devices)
        cost_per_byte: u64,
        internet_gateway: bool,
    }],

    // ── COMPUTE ──
    compute: {
        cpu_class: enum { Micro, Low, Medium, High },
        available_memory_mb: u32,
        mhr_byte: bool,               // can run basic contracts
        wasm: bool,                   // can run full WASM
        cost_per_cycle: u64,
        offered_functions: [{
            function_id: Hash,
            description: String,
            cost_structure: CostStructure,
            max_concurrent: u32,
        }],
    },

    // ── STORAGE ──
    storage: {
        available_bytes: u64,
        storage_class: enum { Volatile, Flash, SSD, HDD },
        cost_per_byte_day: u64,
        max_object_size: u32,
        serves_content: bool,
    },

    // ── AVAILABILITY ──
    uptime_pattern: enum {
        AlwaysOn, Solar, Intermittent, Scheduled(schedule),
    },
```

}

```
    <a id="marketplace-overview--no-special-protocol-primitives"></a>

    ### No Special Protocol Primitives

    Heavy compute like ML inference, transcription, translation, and text-to-speech are **not
    protocol primitives**. They are compute capabilities offered by nodes that have the
    hardware to run them.
```

```
A node with a GPU advertises `offered_functions: [whisper-small, piper-tts]`. A consumer
requests execution through the standard compute delegation path. The protocol is agnostic
to what the function does — it only cares about discovery, negotiation, verification, and
payment.


<a id="marketplace-overview--emergent-specialization"></a>


## Emergent Specialization


Nodes naturally specialize based on hardware and market dynamics:


| Hardware | Natural Specialization | Earns From | Delegates |
├──────┼──────┼──────┼──────┤
| ESP32 + LoRa + solar | Packet relay, availability | Routing fees | Everything else |
| Raspberry Pi + LoRa + WiFi | Compute, LoRa/WiFi bridge | Compute delegation, bridging |
Bulk storage |
| Mini PC + SSD + Ethernet | Storage, DHT, HTTP proxy | Storage fees, proxy fees | Nothing
|
| Phone (intermittent) | Consumer, occasional relay | Relaying while moving | Almost
everything |
| GPU workstation | Heavy compute (inference, etc.) | Compute fees | Nothing |


<a id="marketplace-overview--capability-chains"></a>


## Capability Chains


Delegation cascades naturally:
```

Node A (LoRa relay) └── delegates compute to → Node B (Raspberry Pi) └── delegates storage
to → Node C (Mini PC + SSD) └── delegates connectivity to → Node D (Gateway)

```
Each link is a bilateral agreement with its own [payment channel](#economics-payment-
channels). No central coordination required.



<div class="page-break"></div>


<a id="marketplace-discovery"></a>


# Capability Discovery


Discovery uses concentric rings to minimize bandwidth while ensuring nodes can find the
capabilities they need. Most needs are satisfied locally — physically close nodes are
cheapest and fastest.


<a id="marketplace-discovery--discovery-rings"></a>


## Discovery Rings
```

```
<a id="marketplace-discovery--ring-0-direct-neighbors"></a>

### Ring 0 — Direct Neighbors
```

Scope: Nodes directly connected via any transport Update frequency: Every gossip round (60 seconds) Detail level: Full capability exchange Cost: Free (direct neighbor communication)

```
This is the most detailed and most frequently updated view. A node knows exactly what its
immediate neighbors can offer.

<a id="marketplace-discovery--ring-1-2-3-hops"></a>

### Ring 1 — 2-3 Hops
```

Scope: Nodes reachable in 2-3 hops Update frequency: Every few minutes Detail level: Summarized capabilities, aggregated by type Example: "There's a WASM node 2 hops away, cost ~X"

```
Capabilities are summarized to reduce gossip bandwidth. Instead of full advertisements,
nodes share aggregated summaries using `CapabilitySummary` records:
```

CapabilitySummary { type: u8, // matches beacon bitfield (0=relay, 1=gateway, 2=storage, etc.) count: u8, // number of providers of this type (capped at 255) min_cost: u16, // cheapest provider (log$_2$-encoded μMHR/byte) avg_cost: u16, // average cost across providers (log$_2$-encoded) min_hops: u8, // nearest provider (hop count) max_hops: u8, // farthest provider (hop count) } // 8 bytes per capability type; typical Ring 1 summary: 5-6 types × 8 = 40-48 bytes

Cost encoding: identical to CompactPathCost log$_2$ formula: encoded = round(16 × log$_2$(value + 1)) decoded = (2 ^ (encoded / 16.0)) - 1

Special values: 0x0000 = free (0 μMHR) — valid for trusted-peer services 0xFFFF = cost unknown or unavailable

```
A Ring 1 gossip message contains one `CapabilitySummary` per capability type present in the
2-3 hop neighborhood. Nodes that appear in multiple capability types are counted in each.

<a id="marketplace-discovery--ring-2-trust-neighborhood"></a>

### Ring 2 — Trust Neighborhood
```

Scope: Nodes reachable through the trust graph (friends of friends) Update frequency: Periodic, via trust-weighted gossip Detail level: Neighborhood capability summary Example: "Your

neighborhood has 5 gateways, 20 storage nodes"

```
The trust graph provides a natural scope for aggregated capability information. Trusted
peers share more detailed information than strangers — this is both efficient (trust =
proximity in most cases) and privacy-preserving.


<a id="marketplace-discovery--ring-3-beyond-neighborhood"></a>


### Ring 3 — Beyond Neighborhood
```

Scope: Nodes beyond the trust graph Update frequency: On demand (query-based) Detail level: Coarse hints via Reticulum announces Example: "A node with GPU compute exists at cost ~X, 8 hops away"

```
Beyond-neighborhood discovery is intentionally coarse and query-driven. The details are
resolved when a node actually needs to use a remote capability.


<a id="marketplace-discovery--bandwidth-efficiency"></a>


## Bandwidth Efficiency


The ring structure ensures that the most detailed (and most bandwidth-expensive) capability
information is only exchanged between direct neighbors, where communication is free. As
discovery scope increases, detail decreases proportionally:
```

Ring 0: ~200 bytes per neighbor per round (full capabilities) Ring 1: ~50 bytes per summary per round (aggregated) Ring 2: proportional to trust neighborhood size (periodic) Ring 3: 0 bytes proactive (query-only)

```
On constrained links (< 10 kbps), Rings 2-3 are pull-only — no proactive gossip, only
responses to explicit requests. This fits within [Tier 3 of the bandwidth budget]
(#protocol-network-protocol--bandwidth-budget).


<a id="marketplace-discovery--discovery-process"></a>


## Discovery Process


When a node needs a capability it doesn't have locally:


1. **Check Ring 0**: Can any direct neighbor provide this?
2. **Check Ring 1**: Are there known providers 2-3 hops away?
3. **Check Ring 2**: Does the trust neighborhood have this capability?
4. **Query Ring 3**: Send a capability query beyond the neighborhood


Most requests resolve at Ring 0 or Ring 1. The further out a query goes, the higher the
latency and cost — which naturally incentivizes local provision of common capabilities.
```

```
<a id="marketplace-discovery--mobile-handoff"></a>

## Mobile Handoff

When a mobile node (phone, laptop, vehicle) moves between areas, its Ring 0 neighbors
change. Old relay agreements and payment channels become unreachable. The handoff protocol
re-establishes connectivity in the new location.

<a id="marketplace-discovery--presence-beacons"></a>

### Presence Beacons

Mehr nodes periodically broadcast a lightweight presence beacon on all their interfaces:
```

PresenceBeacon { node_id: [u8; 16], // destination hash capabilities: u16, // bitfield (see below) cost_tier: u8, // 0=free/trusted, 1=cheap, 2=moderate, 3=expensive load: u8, // current utilization (0-255) } // 20 bytes — broadcast every 10 seconds

Capability bitfield assignments: Bit 0: relay (L1+ — will forward packets) Bit 1: gateway (internet uplink available) Bit 2: storage (MHR-Store provider) Bit 3: compute_byte (MHR-Byte interpreter) Bit 4: compute_wasm (WASM runtime — Light or Full) Bit 5: pubsub (MHR-Pub hub) Bit 6: dht (MHR-DHT participant) Bit 7: naming (MHR-Name resolver) Bits 8-15: reserved (must be 0; future: inference, bridge, etc.)

```
Beacons are transport-agnostic — they go out over whatever interfaces the node has (LoRa,
WiFi, BLE, etc.). A mobile node passively receives beacons to discover local Mehr nodes
before initiating any connection. This is the decentralized equivalent of a cellular tower
scan.

**Beacon propagation rules**:
- Beacons are broadcast by the originating node only — **not relayed** by others
- Scope: local interface (each transport broadcasts independently)
- Collision handling: CSMA/CA at the transport layer (listen-before-talk on LoRa)
- Missed beacons: a node that misses one beacon catches the next in 10 seconds
- **Density adaptation**: if local channel utilization exceeds 50% (measured via CSMA/CA
back-off frequency), beacon interval doubles to 20 seconds. Above 75%, interval increases
to 30 seconds. This prevents beacons from consuming excessive bandwidth in dense
deployments
- Redundancy: Ring 1 gossip (CapabilitySummary) provides backup discovery for nodes that
miss beacons — discovery is not solely beacon-dependent

<a id="marketplace-discovery--handoff-sequence"></a>

### Handoff Sequence

When a mobile node detects new beacons (new area) or loses contact with its current relay:
```

Handoff:

1. Select best relay from received beacons (lowest cost × load)

2. Connect and establish link-layer encryption

3. Open payment channel (both sign initial state)

4. Resume communication through new relay

```
On high-bandwidth links (WiFi, BLE), this completes in under 500ms. On LoRa, a few seconds.

<a id="marketplace-discovery--credit-based-fast-start"></a>

### Credit-Based Fast Start

For latency-sensitive handoffs (e.g., active voice call), a credit mechanism allows
immediate relay before the payment channel is fully established:
```

CreditGrant { grantor: NodeID, // relay grantee: NodeID, // mobile node credit_limit_bytes: u32, // relay allowance before channel required valid_for_ms: u16, // credit window (default: 30 seconds) condition: enum { VisibleBalance(min_mhr), // grantee has balance on CRDT ledger TrustGraph, // grantee is in grantor's trust graph Unconditional, // free initial credit (attract users) }, }

```
The relay checks the mobile node's balance on the CRDT ledger (already available via
gossip) and extends temporary credit. Packets flow immediately while the channel opens in
the background.

**Staleness tolerance**: The relay's CRDT view may be stale (especially after a partition).
The credit grant is bounded by `credit_limit_bytes` and `valid_for_ms`, limiting risk to at
most one credit window of unpaid traffic. Relays rate-limit fast start grants to **one
active grant per unknown node** — a node that exhausts its credit without opening a channel
cannot receive another grant for 10 minutes. For `VisibleBalance` grants, the relay
requires a balance of at least `2 × credit_limit_bytes × cost_per_byte` to absorb
staleness.

If the mobile node has no visible balance and no trust relationship, it must complete the
channel open first.

<a id="marketplace-discovery--roaming-cache"></a>

### Roaming Cache

Mobile nodes cache relay information for areas they've visited:
```

RoamingEntry { area_fingerprint: [u8; 16], // Blake3 hash of sorted beacon node_ids relays: [{ node_id: NodeID, capabilities: u16, last_cost_tier: u8, last_seen: Timestamp, channel: Option, // preserved from last visit }], ttl: Duration, // expire after 30 days of non-visit }

When a mobile node enters a previously visited area, it recognizes the beacon fingerprint and reconnects to a cached relay. If a preserved `ChannelState` exists and the relay is still alive, the old channel resumes with zero handoff latency — no new negotiation needed.

Fingerprint tolerance: The area fingerprint is approximate — node churn between visits is expected. The mobile node matches if at least 60% of current beacon node_ids appear in the cached fingerprint's sorted set. This is computed as `|intersection| / |cached_set| ≥ 0.6`. If below threshold, the area is treated as new (full discovery). Beacon node_ids are sorted by numeric value of the destination hash.

Graceful Departure

No explicit teardown:
- Old agreements expire naturally via `valid_until`
- Old payment channels remain valid and settle lazily (next contact, or via gossip)
- The mobile node's trust graph travels with it — if trusted peers exist in the new area, relay is free immediately

<div class="page-break"></div>

Capability Agreements

When a requester finds a suitable provider through [discovery](#marketplace-discovery), they form a bilateral agreement. Agreements are between two parties only — no network-wide registration required.

Cost Structure

Agreements use a discriminated cost model that adapts to different capability types:

CostStructure: enum { PerByte { cost_per_byte: u64, // µMHR per byte transferred }, PerInvocation { cost_per_call: u64, // µMHR per function invocation max_input_bytes: u32, // cost covers up to this input size }, PerDuration { cost_per_epoch: u64, // µMHR per epoch of service }, PerCycle { cost_per_million_cycles: u64, // µMHR per million compute cycles max_cycles: u64, // hard limit }, }

```
| Capability | Typical CostStructure |
|————————————|———————————————————————|
| Relay / Bandwidth | `PerByte` |
| Storage | `PerDuration` |
| Compute (contract) | `PerCycle` |
```

```
| Compute (function) | `PerInvocation` |
| Internet gateway | `PerByte` or `PerDuration` |


<a id="marketplace-agreements--agreement-structure"></a>


## Agreement Structure
```

CapabilityAgreement { provider: NodeID, consumer: NodeID, capability: CapabilityType, // compute, storage, relay, proxy payment_channel: ChannelID, // existing bilateral channel cost: CostStructure, valid_until: Timestamp,

```
proof_method: enum {
    DeliveryReceipt,            // for relay
    ChallengeResponse,          // for storage (random read challenges)
    ResultHash,                 // for compute (hash of output)
    Heartbeat,                  // for ongoing services
},

signatures: (Sig_Provider, Sig_Consumer),
```

}

```
<a id="marketplace-agreements--key-properties"></a>


## Key Properties

<a id="marketplace-agreements--bilateral"></a>


### Bilateral

Agreements are strictly between two parties. This means:

- No central registry of agreements
- No third party needs to be involved or informed
- Agreements can be formed and dissolved without network-wide coordination
- Privacy is preserved — only the two parties know the terms

<a id="marketplace-agreements--payment-linked"></a>


### Payment-Linked

Every agreement references an existing [payment channel](#economics-payment-channels)
between the two parties. Payment flows automatically as the service is delivered.

<a id="marketplace-agreements--time-bounded"></a>


### Time-Bounded

Agreements have an expiration (`valid_until`). This prevents stale agreements from
```

persisting when nodes move or go offline. Parties can renew by forming a new agreement.

Proof-Verified

Each agreement specifies how the consumer verifies that the provider is actually delivering. See [Verification](#marketplace-verification) for details on each proof method.

Agreement Lifecycle

Agreement states: Active: now < valid_until — service is being delivered Expired: now >= valid_until — no new service; grace period begins Grace: expired + up to 1 gossip round (60s) — allows in-flight operations to complete Closed: after grace period — agreement is fully terminated

Expiry Behavior

Capability	On Expiry	Grace Period
────────────	────────────	────────────────
Relay/Bandwidth	No new packets routed after `valid_until`	In-flight packets in queue are delivered (up to 60s drain)
Storage	No new writes accepted	Data remains stored for 1 additional epoch; then subject to [garbage collection](#services-mhr-store--garbage-collection)
Compute	No new invocations accepted	Running invocations complete; results remain retrievable for 60s
Internet Gateway	Connection torn down at `valid_until`	In-flight TCP streams drained for up to 60s

Renewal

To renew, the consumer sends a new `CapabilityRequest` before the current agreement expires. If terms are unchanged, the provider can respond with a `CapabilityOffer` that extends `valid_until` — no full re-negotiation needed. If terms change, both parties sign a new `CapabilityAgreement`.

Billing Boundaries

- **PerByte / PerInvocation / PerCycle**: Charged for actual usage up to `valid_until`. No charge after expiry.
- **PerDuration**: Charged for completed epochs only. Partial epochs at agreement end are not billed.


```
## Agreement Types

| Capability | Typical Duration | Proof Method | Example |
├──────────────┼──────────────────────┼──────────────────┼──────────────┤
| **Relay/Bandwidth** | Per-packet or ongoing | Delivery Receipt | "Route my packets for
the next hour" |
| **Storage** | Hours to months | Challenge-Response | "Store this 10 MB file for 30 days"
|
| **Compute** | Per-invocation | Result Hash | "Run Whisper on this audio file" |
| **Internet Gateway** | Ongoing | Heartbeat | "Proxy my traffic to the internet" |

<a id="marketplace-agreements--negotiation"></a>

## Negotiation

Negotiation is **single-round** (take-it-or-leave-it) and strictly local — no auction, no
bidding, no global price discovery:
```

Negotiation protocol:

1. Consumer sends CapabilityRequest to provider

2. Provider responds with CapabilityOffer (or Reject)

3. Consumer accepts (signs) or walks away

4. If accepted: both signatures form the CapabilityAgreement

5. Service begins; payment flows through the channel

CapabilityRequest { consumer: NodeID, capability: CapabilityType, // compute, storage, relay, proxy desired_cost: CostStructure, // max cost consumer will accept desired_duration: u32, // seconds payment_channel: ChannelID, // existing channel with this provider proof_preference: ProofMethod, // DeliveryReceipt, ChallengeResponse, etc. nonce: u64, // replay prevention } // Signed by consumer

CapabilityOffer { request_nonce: u64, // matches the request provider: NodeID, actual_cost: CostStructure, // provider's terms (≤ desired_cost, or reject) valid_until: Timestamp, // agreement expiration proof_method: ProofMethod, // may differ from preference constraints: Option<Vec>, // provider-specific (e.g., max object size) } // Signed by provider

```
**Timeout**: If the provider doesn't respond within 30 seconds (or 3 gossip rounds on
constrained links), the request is considered rejected. The consumer may retry with a
different provider.

**No counter-offers**: The provider either meets or undercuts the consumer's desired cost,
or rejects. This keeps negotiation to a single round-trip — critical for LoRa where each
message takes seconds. If the consumer wants to negotiate, they send a new request with
adjusted terms.
```

```
    Prices are set by providers based on their own cost structure. Within [trust neighborhoods]
    (#economics-trust-neighborhoods), trusted peers often offer discounted or free services.



    <div class="page-break"></div>



    <a id="marketplace-verification"></a>

    # Verification

    The capability marketplace requires that consumers can verify providers are actually
    delivering the agreed service. Mehr uses different verification methods depending on the
    type of capability.

    <a id="marketplace-verification--relay-bandwidth-verification"></a>

    ## Relay / Bandwidth Verification

    **Method**: Cryptographic delivery receipts

    The destination node signs a receipt proving the packet arrived. The relay chain can prove
    it delivered. This creates an unforgeable chain of evidence:
```

Packet sent by Alice → relayed by Bob → relayed by Carol → received by Dave

Dave signs: Receipt(packet_hash, timestamp) Carol proves: "I forwarded to Dave, here's Dave's receipt" Bob proves: "I forwarded to Carol, here's the chain"

```
    A relay node can only earn routing fees by actually delivering packets to their
    destination.

    <a id="marketplace-verification--storage-verification"></a>

    ## Storage Verification

    **Method**: Merkle-proof challenge-response (see [MHR-Store](#services-mhr-store--proof-of-
    storage) for full details)

    The consumer challenges a random chunk and the provider returns a Blake3 hash plus a Merkle
    proof:
```

Challenge-Response Protocol:

1. At storage time, consumer builds a Merkle tree over 4 KB chunks and stores only the merkle_root locally

2. Periodically, consumer sends: Challenge(data_hash, random_chunk_index, nonce)

3. Provider responds: Proof(Blake3(chunk_data || nonce), merkle_siblings)

4. Consumer recomputes merkle root from proof — if it matches, data is verified

```
This is:
- **Lightweight**: Runs on ESP32 in under 10ms — no GPU, no heavy crypto
- **Nonce-protected**: The random nonce prevents pre-computation of responses
- **Merkle-verified**: Consumer only stores the root hash, not the full data
- **Bandwidth-efficient**: ~320 bytes per proof (for a 1 MB file)
- **Partition-safe**: Works between any two directly connected nodes, no chain needed

Three consecutive failed challenges trigger [repair](#services-mhr-store--repair) — the
consumer reconstructs the lost shard from erasure-coded replicas and stores it on a
replacement node.

<a id="marketplace-verification--compute-verification"></a>

## Compute Verification

Compute verification uses three tiers, scaled to the stakes involved:

<a id="marketplace-verification--tier-1-reputation-trust-cheapest"></a>

### Tier 1: Reputation Trust (Cheapest)

Accept the result. The provider has no incentive to lie — getting caught destroys their
reputation and all future income.

**Use for**: Low-stakes operations where the cost of a wrong answer is low.

<a id="marketplace-verification--tier-2-optimistic-verification-moderate"></a>

### Tier 2: Optimistic Verification (Moderate)

Accept the result but randomly re-execute 1-in-N requests on a different node. Divergent
results flag the provider for investigation.
```

Optimistic Verification:

1. Send compute request to Provider A

2. Accept result immediately

3. With probability 1/N, also send same request to Provider B

4. Compare results

5. If divergent: flag Provider A, reduce reputation

```
**Use for**: Medium-stakes operations. The random audit probability can be tuned — higher
for newer/less-trusted providers, lower for established ones.

<a id="marketplace-verification--tier-3-redundant-execution-expensive"></a>
```

```
### Tier 3: Redundant Execution (Expensive)

Send the same request to K independent nodes. The majority result wins.
```

Redundant Execution:

1. Send compute request to K nodes (e.g., K=3)

2. Collect results

3. Majority wins (2 of 3 agree)

4. Dissenting node is flagged

```
**Use for**: High-stakes operations where the result affects payments or irreversible state
changes.

<a id="marketplace-verification--verification-cost-tradeoffs"></a>

## Verification Cost Tradeoffs

| Tier | Cost | Latency | Trust Required | Use Case |
|──────|──────|─────────|────────────────|──────────|
| Reputation | 1x | 1x | High | Cheap, frequent ops |
| Optimistic | ~1.1x | 1x | Moderate | Default for most compute |
| Redundant | Kx | ~1x (parallel) | Minimal | Payment-affecting compute |

<a id="marketplace-verification--heartbeat-verification"></a>

## Heartbeat Verification

For ongoing services (internet gateway, persistent connections), a simple heartbeat
mechanism verifies continued availability:
```

Heartbeat Protocol:

1. Consumer sends periodic ping (every N seconds)

2. Provider responds with signed pong

3. If M consecutive heartbeats are missed: agreement terminated

4. Payment stops when heartbeats stop

```
This is suitable for services where the consumer can directly observe whether the service
is working (e.g., "I can reach the internet through this gateway").



<div class="page-break"></div>
```

```
<a id="services-mhr-store"></a>

# MHR-Store: Content-Addressed Storage

MHR-Store is the storage layer of Mehr. Every piece of data is addressed by its content
hash — if you know the hash, you can retrieve the data from anywhere in the network.
Storage is maintained through bilateral agreements, verified through lightweight challenge-
response proofs, and protected against data loss through erasure coding.

<a id="services-mhr-store--data-objects"></a>

## Data Objects
```

DataObject { hash: Blake3Hash, // content hash = address content_type: enum { Immutable, Mutable, Ephemeral }, owner: Option, // for mutable objects created: Timestamp, ttl: Option, // for ephemeral objects size: u32, priority: enum { Critical, Normal, Lazy }, min_bandwidth: u32, // don't attempt transfer below this bps

```
// Merkle tree root over 4 KB chunks (for verification)
merkle_root: Blake3Hash,

payload: enum {
    Inline(Vec<u8>),            // small objects (under 4 KB)
    Chunked([ChunkHash]),       // large objects (4 KB chunks)
},
```

}

```
<a id="services-mhr-store--content-types"></a>

## Content Types

<a id="services-mhr-store--immutable"></a>

### Immutable

Once created, the content never changes. The hash is the permanent address. Used for:
messages, posts, media files, contract code.

<a id="services-mhr-store--mutable"></a>

### Mutable

The owner can publish updated versions, signed with their key. The highest sequence number
wins. Any node can verify the signature. Used for: profiles, status updates, configuration.

**Versioning rules**:
- Sequence numbers must be **strictly monotonic** — each update must have a higher sequence
number than the previous one
```

```
- Updates are only valid when signed by the owner's Ed25519 key
- **Fork detection**: If two updates with the same sequence number but different content
are observed (both validly signed), this is treated as evidence of key compromise or device
cloning

**Fork handling**:
```

When a fork is detected (sequence N, two different content hashes, both validly signed):

1. Record: store (object_key, sequence, both content hashes, detection_time)

2. Block: reject new updates to this object from this owner for 24 hours or until a KeyCompromiseAdvisory with SignedByBoth evidence is received

3. Gossip: include the fork evidence in the next gossip round as advisory metadata (both conflicting hashes + sequence). This is informational — receiving nodes independently verify and may apply their own block

4. Dedup: fork records are retained for 7 days to prevent re-reporting

5. Resolution: a KeyCompromiseAdvisory with SignedByBoth clears the block and allows the new key's updates to proceed. SignedByOldOnly does NOT clear the block (could be the attacker)

```
<a id="services-mhr-store--ephemeral"></a>

### Ephemeral

Data with a time-to-live (TTL). Automatically garbage-collected after expiration. Used for:
presence information, temporary caches, session data.

<a id="services-mhr-store--storage-agreements"></a>

## Storage Agreements

Storage on Mehr is maintained through **bilateral agreements** between data owners and
storage nodes. This is how data stays alive on the network.
```

StorageAgreement { data_hash: Blake3Hash, // what's being stored data_size: u32, // bytes provider: NodeID, // who stores it consumer: NodeID, // who pays for it payment_channel: ChannelID, // bilateral channel cost_per_epoch: u64, // MHR per epoch duration_epochs: u32, // how long challenge_interval: u32, // how often to verify (in gossip rounds) erasure_role: Option,// if part of an erasure-coded set signatures: (Sig_Provider, Sig_Consumer), }

```
<a id="services-mhr-store--payment-model"></a>

### Payment Model

Storage is **pay-per-duration** - like rent, not like purchase. The data owner pays the
```

```
    storage node a recurring fee via their bilateral [payment channel](#economics-payment-
    channels).

    | Duration | Billing | Use Case |
    ├─────────────┼──────────────┼─────────────┤
    | Short-term (hours) | Per-epoch micro-payments | Temporary caches, session data |
    | Medium-term (weeks) | Prepaid for N epochs | Messages, posts, media |
    | Long-term (months+) | Recurring per-epoch | Persistent data, profiles, hosted content |

    When payment stops, the storage node garbage-collects the data after a grace period (1
    epoch). The data owner is responsible for maintaining payment — there is no "permanent
    storage" guarantee.

    <a id="services-mhr-store--free-storage-between-trusted-peers"></a>

    ### Free Storage Between Trusted Peers

    Just like [relay traffic](#economics-trust-neighborhoods), storage between trusted peers is
    **free**:
```

Storage decision: if data owner is trusted: store for free (no agreement needed, no payment) else: require a StorageAgreement with payment

```
    A trust neighborhood where members store each other's data operates with zero economic
    overhead — no tokens, no agreements, no challenges. This is how a community mesh handles
    local content naturally.

    <a id="services-mhr-store--proof-of-storage"></a>

    ## Proof of Storage

    How does a data owner know a storage node actually has their data? Through **lightweight
    challenge-response proofs** that run on any hardware, including ESP32.

    <a id="services-mhr-store--challenge-response-protocol"></a>

    ### Challenge-Response Protocol
```

Proof of Storage:

1. Data owner builds a Merkle tree over 4 KB chunks at storage time (stores only the merkle_root — not the full tree)

2. Periodically, owner sends: Challenge { data_hash: Blake3Hash, chunk_index: u32, // random chunk to verify nonce: [u8; 16], // prevents pre-computation }

3. Storage node responds: Proof { chunk_hash: Blake3(chunk_data || nonce), merkle_proof: [sibling hashes from chunk to root], }

4. Owner verifies: a. Recompute merkle root from chunk_hash + merkle_proof b. Compare against stored merkle_root c. If match: storage verified d. If mismatch: node is lying or lost the data

```
<a id="services-mhr-store--why-this-works-on-constrained-devices"></a>

### Why This Works on Constrained Devices

| Operation | Compute Cost | RAM Required |
|───────────────┼──────────────────┼───────────────|
| Generate challenge | 16 random bytes | Negligible |
| Compute chunk hash (storage node) | 1 Blake3 hash of 4 KB | 4 KB |
| Verify Merkle proof (owner) | ~10 Blake3 hashes (for 1 MB file) | ~320 bytes |

An ESP32 can verify a storage proof in under 10ms. No GPU needed, no heavy cryptography, no
sealing. This is intentionally simpler than Filecoin's Proof of Replication — we trade the
ability to detect deduplicated storage for something that actually runs on mesh hardware.


<a id="services-mhr-store--challenge-frequency"></a>

### Challenge Frequency

| Data Priority | Challenge Interval |
|───────────────┼──────────────────────|
| Critical | Every gossip round (60 seconds) |
| Normal | Every 10 gossip rounds (~10 minutes) |
| Lazy | Every 100 gossip rounds (~100 minutes) |

Challenges are staggered across stored objects so a storage node never faces a burst of
challenges at once.


<a id="services-mhr-store--what-if-a-challenge-fails"></a>

### What If a Challenge Fails?
```

Challenge failure handling:

1. First failure: retry after 1 gossip round (could be transient)

2. Second consecutive failure: flag the storage node

3. Third consecutive failure: consider data lost on this node → trigger repair (see Erasure Coding below) → reduce node's storage reputation → terminate the StorageAgreement

```
<a id="services-mhr-store--erasure-coding"></a>

## Erasure Coding

Full replication is wasteful. Storing 3 complete copies of a file costs 3x the storage.
**Erasure coding** achieves the same durability with far less overhead.
```

```
<a id="services-mhr-store--reed-solomon-coding"></a>

### Reed-Solomon Coding

Mehr uses Reed-Solomon erasure coding to split data into **k data shards + m parity
shards**, where any k of (k + m) shards can reconstruct the original:
```

Erasure coding example (4, 2): Original file: 1 MB → Split into 4 data shards (256 KB each) → Generate 2 parity shards (256 KB each) → 6 shards total, stored on 6 different nodes → Any 4 of 6 shards can reconstruct the original → Total storage: 1.5 MB (1.5x overhead)

Compare with 3x replication: → 3 full copies = 3 MB (3x overhead) → Tolerates 2 node failures (same as erasure coding) → But uses 2x more storage

```
<a id="services-mhr-store--default-erasure-parameters"></a>

### Default Erasure Parameters

| Data Size | Scheme | Shards | Overhead | Tolerates |
|———————|————|————|—————|——————|
| Under 4 KB | No erasure (inline) | 1 | 1x | Replication only |
| 4 KB - 1 MB | (2, 1) | 3 shards | 1.5x | 1 node loss |
| 1 MB - 100 MB | (4, 2) | 6 shards | 1.5x | 2 node losses |
| Over 100 MB | (8, 4) | 12 shards | 1.5x | 4 node losses |

The data owner chooses the scheme based on durability requirements and willingness to pay.
Higher redundancy = more shards = more storage agreements = higher cost.

<a id="services-mhr-store--shard-distribution"></a>

### Shard Distribution

Shards are distributed across nodes in **different trust neighborhoods** to maximize
independence:
```

Shard placement strategy:

1. Prefer nodes in different trust neighborhoods

2. Prefer nodes with different transport types (LoRa, WiFi, cellular)

3. Prefer nodes with proven uptime (high storage reputation)

4. Never place two shards of the same object on the same node

```
This ensures that a single neighborhood going offline (power outage, network split) doesn't
lose more shards than the erasure code can tolerate.

<a id="services-mhr-store--repair"></a>
```

```
## Repair

When a storage node fails challenges or goes offline, the data owner must **repair** —
reconstruct the lost shard and store it on a new node.
```

Repair flow:

1. Detect: storage node fails 3 consecutive challenges

2. Assess: how many shards are still healthy?

   - If >= k shards remain: reconstruction is possible

   - If fewer than k: data is lost (this is why shard distribution matters)

3. Reconstruct: download k healthy shards, regenerate the lost shard

4. Re-store: form a new StorageAgreement with a different node

5. Upload the reconstructed shard

```
<a id="services-mhr-store--automated-repair"></a>

### Automated Repair

For users who can't be online to monitor their data, repair can be delegated:
```

RepairAgent { data_hash: Blake3Hash, shard_map: Map<ShardIndex, NodeID>, merkle_root: Blake3Hash, authorized_spender: NodeID, // can spend from owner's channel max_repair_cost: u64, // budget cap }

```
A RepairAgent is an [MHR-Compute contract](#services-mhr-compute) that periodically
challenges storage nodes on behalf of the data owner. If a shard is lost, it handles
reconstruction and re-storage automatically, spending from the owner's pre-authorized
budget.

<a id="services-mhr-store--bandwidth-adaptation"></a>

## Bandwidth Adaptation

The `min_bandwidth` field controls how data propagates across links of different speeds:
```

Example: A 500 KB image declares min_bandwidth: 10000 (10 kbps)

LoRa node (1 kbps): → Propagates hash and metadata only → Never attempts to transfer the full image

WiFi node (100 Mbps): → Transfers normally

```
This is a property of the data object that the storage and routing layers respect.
Applications set `min_bandwidth` based on the nature of the data, and the network handles
the rest.


<a id="services-mhr-store--garbage-collection"></a>


## Garbage Collection


Storage nodes manage their disk space through a priority-based garbage collection system:
```

Garbage collection priority (lowest priority deleted first):

1. Expired TTL + no active agreement → immediate deletion

2. Unpaid (agreement expired, no renewal) → delete after 1 epoch grace

3. Cached content (no agreement, just opportunistic) → LRU eviction

4. Low-priority paid data → delete only under extreme space pressure

5. Normal paid data → never delete while agreement is active

6. Critical paid data → never delete while agreement is active

7. Trusted peer data → never delete while trust relationship exists

```
A storage node never deletes data that has an active, paid agreement. Data whose agreement
expires is kept for a 1-epoch grace period (to allow renewal), then garbage-collected.


<a id="services-mhr-store--chunking"></a>


## Chunking


Large objects are split into 4 KB chunks, each independently addressed by hash:
```

Large file (1 MB): → Split into 256 chunks of 4 KB each → Each chunk has its own Blake3 hash → Merkle tree built over all chunk hashes → The DataObject stores the chunk hash list and merkle_root → Chunks can be retrieved from different nodes in parallel → Missing chunks can be re-requested individually

```
Chunking enables:
- Parallel downloads from multiple peers
- Efficient deduplication (identical chunks across objects are stored once)
- Resumable transfers on unreliable links
- Fine-grained storage proofs (challenge any individual chunk)
- Erasure coding at the chunk level for large objects


<a id="services-mhr-store--reassembly"></a>
```

```
### Reassembly
```

Fragment reassembly protocol: Per-chunk timeout: 30 seconds (configurable per StorageAgreement) Retry policy: exponential backoff (2s, 4s, 8s, max 30s), up to 3 retries After 3 retries: mark chunk provider as unreliable, try alternate via DHT Overall timeout: 5 minutes (all chunks must arrive within this window)

Resumable downloads: Consumer tracks received chunk indices. To resume, send: ChunkRequest { data_hash: Blake3Hash, chunk_indices: Vec } Provider responds with only the requested chunks, avoiding retransmission of already-received data.

```
<a id="services-mhr-store--comparison-with-other-storage-protocols"></a>

## Comparison with Other Storage Protocols

| Aspect | Filecoin | Arweave | Mehr (MHR-Store) |
|————————|————————|————————|——————————————————|
| **Payment** | Per-deal, on-chain | One-time endowment | Per-duration, bilateral channels
|
| **Proof** | PoRep + PoSt (GPU-heavy, minutes to seal) | SPoRA (mining-integrated) |
Challenge-response (milliseconds, runs on ESP32) |
| **Durability** | Slashing for failures (requires blockchain) | Incentivized mining of
historical data | Erasure coding + repair agents |
| **Permanent storage** | No (deals expire) | Yes (pay once) | No (pay per duration, data
owner's responsibility) |
| **Blockchain** | Required (proof submission on-chain) | Required (block weave) | Not
needed (bilateral agreements) |
| **Minimum hardware** | GPU for sealing | Standard PC | ESP32 for verification, Pi for
storage |
| **Partition tolerance** | No (needs chain access) | No (needs chain access) | Yes
(bilateral proofs work offline) |
| **Free tier** | No | No | Yes (trusted peer storage) |

Mehr deliberately chooses lightweight proofs over heavy cryptographic guarantees. The
tradeoff: a storage node could store the same data once and claim to store it twice (unlike
Filecoin's Proof of Replication). This is acceptable because:

1. The economic incentive is weak — the node earns the same fee either way
2. The data owner doesn't care *how* the node stores the data, only that it can return it
on demand
3. Erasure coding across multiple nodes provides real redundancy regardless



<div class="page-break"></div>


<a id="services-mhr-dht"></a>
```

MHR-DHT: Distributed Hash Table

MHR-DHT maps keys to the nodes that store the corresponding data. It uses proximity-weighted gossip rather than Kademlia-style strict XOR routing, because link quality varies wildly on a mesh network.

Distance Metrics: Routing vs. DHT

Mehr uses two different distance metrics for different purposes:

- **Ring distance** (routing layer): `min(|a - b|, 2^128 - |a - b|)` over the destination hash space. Used for [greedy forwarding](#protocol-network-protocol--small-world-routing-model) to route packets toward their destination. This is the Kleinberg small-world model.
- **XOR distance** (DHT layer): `a ⊕ b` over DHT key space. Used for determining storage responsibility — which nodes are "closest" to a given key and should store its data.

Both operate over 128-bit spaces derived from the same hash functions, but they serve different roles. Routing cares about navigating to a destination efficiently; the DHT cares about partitioning key-space responsibility among nodes.

Why Not Kademlia?

Traditional Kademlia routes lookups based on XOR distance between node IDs and key hashes, assuming roughly uniform latency between any two nodes. On a Mehr mesh:

- A node 1 XOR-hop away might be 10 LoRa hops away
- A node 10 XOR-hops away might be a direct WiFi neighbor
- Link quality varies by orders of magnitude

MHR-DHT uses **proximity-weighted gossip** that considers both XOR distance and actual network cost when deciding where to route lookups. XOR distance determines the **target** (which nodes should store a key); network cost determines the **path** (how to reach those nodes efficiently).

Routing Algorithm

Lookup Scoring Function

Each DHT lookup hop selects the next node by minimizing:

dht_score(candidate, key) = w_xor × norm_xor_distance(candidate.id, key) + (1 - w_xor) × norm_network_cost(candidate)

```
Where:
- `norm_xor_distance` = `xor(candidate.id, key) / max_xor_in_candidate_set`, normalized to
[0, 1]
- `norm_network_cost` = `cumulative_cost_to(candidate) / max_cost_in_candidate_set`,
normalized to [0, 1]
- `w_xor = 0.7` (default — favor key-space closeness, but avoid expensive paths)


This produces the same iterative-closest-node behavior as Kademlia but routes around
expensive links rather than blindly following XOR distance.


<a id="services-mhr-dht--replication-factor"></a>


### Replication Factor


Each key is stored on the **k=3 closest nodes** in XOR distance that are reachable within a
cost budget:
```

Storage responsibility:

1. Sort all known nodes by xor(node.id, key)

2. Walk the sorted list; skip nodes whose network cost exceeds 10× the cheapest

3. First k=3 reachable nodes are the storage set

```
The cost filter prevents a node on the far side of a LoRa link from being assigned storage
responsibility for a key it can barely reach. The XOR ordering ensures deterministic
agreement on who stores what.


<a id="services-mhr-dht--rebalancing"></a>


### Rebalancing


- **Node join**: A new node announces itself. Existing nodes check whether any stored keys
are now closer (in XOR) to the new node. After **2 gossip rounds** (for announcement
convergence), affected keys are pushed via gossip metadata. The new node pulls full data
and becomes part of the storage set.
- **Node departure**: Detected via missed heartbeats (3 consecutive gossip rounds = ~3
minutes). The departed node is immediately marked down — no new writes are sent to it.
After **6 additional missed rounds** (~6 minutes total since last response), remaining
storage-set members initiate re-replication to the next-closest reachable node, restoring
k=3. If no reachable replacement exists within the cost budget, durability is temporarily
degraded (k=2 or k=1) and a warning is logged. Normal replication resumes when a suitable
node becomes available.


<a id="services-mhr-dht--lookup-process"></a>


## Lookup Process
```

DHT Lookup:

1. Query direct neighbors for the key

2. Each responds with either the data or a referral to a closer node (selected by dht_score — balancing XOR closeness and network cost)

3. Follow referrals iteratively until data is found or all k closest nodes queried

4. Cache result locally with TTL

5. Parallel lookups: query α=3 nodes concurrently, use first valid response

```
<a id="services-mhr-dht--bandwidth-per-lookup"></a>

### Bandwidth per Lookup

| Component | Size |
├─────────────┼────────┤
| Query | ~64 bytes |
| Response (referral) | ~48 bytes (node_id + cost hint) |
| Response (data found) | ~128 bytes + data size |
| Typical lookup (3-5 hops on LoRa) | 2-3 seconds |

<a id="services-mhr-dht--publication-process"></a>

## Publication Process
```

DHT Publication:

1. Store the object locally

2. Gossip key + metadata (not full data) to neighbors

3. Nodes close to the key's hash (k=3 storage set) pull the full data

4. Neighborhood-scoped objects gossip within the trust neighborhood only

```
Publication gossips only metadata — the full data is pulled on demand. This prevents large
objects from flooding the gossip channel.

<a id="services-mhr-dht--metadata-format"></a>

### Metadata Format
```

DHTMetadata { key: [u8; 32], // Blake3 content hash (32 bytes) size: u32, // object size in bytes (4 bytes) content_type: u8, // 0=Immutable, 1=Mutable, 2=Ephemeral (1 byte) owner: [u8; 16], // publisher's destination hash (16 bytes) ttl_remaining: u32, // seconds until expiry (4 bytes) lamport_ts: u64, // publisher's Lamport timestamp (8 bytes, for mutable ordering) signature: [u8; 64], // Ed25519 signature over (key || size || content_type || lamport_ts) (64 bytes) } // Total: 129 bytes per metadata entry // Signature prevents metadata forgery; content hash prevents data forgery

For **mutable objects**, `lamport_ts` determines freshness — the highest timestamp with a valid signature wins. For **immutable objects**, `lamport_ts` is the publication time and the content hash is sufficient for verification.

Cache invalidation: Mutable objects are invalidated by receiving a metadata entry with a higher `lamport_ts` for the same `owner` and logical key. There is no push-invalidation mechanism — caches rely on TTL expiry and periodic re-query of the storage set for freshness-critical data.

Cache TTL

Cache lifetime follows a two-level policy:

- **Publisher TTL**: Set by the data owner. Maximum lifetime for the cached copy. Range: 60 seconds to 30 days.
- **Local cap**: `min(publisher_ttl, 24 hours)`. Prevents stale caches from persisting when publishers update their data.
- **Access refresh**: Accessing a cached item resets its local TTL to `min(remaining_publisher_ttl, 24 hours)`. Frequently accessed items stay cached; idle items expire.
- **Eviction**: When local cache exceeds its storage budget, least-recently-used entries are evicted first regardless of remaining TTL.

Neighborhood-Scoped DHT

Objects can be scoped to a [trust neighborhood](#economics-trust-neighborhoods), meaning:

- Their metadata only gossips between trusted peers and their neighbors
- Only nodes within the trust neighborhood can discover them
- Storage nodes within the neighborhood are preferred
- Cross-neighborhood lookups require explicit queries (Ring 3 discovery)

This is useful for community content that doesn't need global visibility. Scoping emerges naturally from the trust graph — there is no explicit "zone" to configure.

Caching

Lookup results are cached locally with a TTL (time-to-live). This means:

- Frequently accessed data is served from local cache
- The DHT is queried only when the cache expires
- Popular content naturally distributes across many caches
- Cache TTL is set by the data publisher


```
## Light Client Verification

Mobile nodes and other constrained devices delegate DHT lookups to a nearby relay rather
than participating in the DHT directly. This creates a trust problem: how does the light
client know the relay's response is honest?

Three verification tiers handle this, scaled by data type:

<a id="services-mhr-dht--tier-1-content-addressed-lookups-zero-overhead"></a>

### Tier 1 — Content-Addressed Lookups (Zero Overhead)

Most DHT objects are stored by content hash. Verification is automatic:
```

Light client lookup by hash:

1. Request key K from relay

2. Relay returns data D

3. Verify: Blake3(D) == K

4. Match → data is authentic (relay honesty irrelevant)

5. Mismatch → discard, flag relay, retry via different node

```
No extra bandwidth, no extra queries. The hash the client already knows is the proof.

<a id="services-mhr-dht--tier-2-signed-object-lookups-signature-check"></a>

### Tier 2 — Signed Object Lookups (Signature Check)

For mutable data (MHR-Name records, capability advertisements, profile updates), objects
carry the owner's Ed25519 signature:
```

Light client lookup for mutable object:

1. Request mutable key from relay

2. Relay returns: { data, owner_pubkey, signature, lamport_timestamp }

3. Verify: Ed25519_verify(owner_pubkey, data || timestamp, signature)

4. Valid → data is authentic (relay cannot forge owner's signature)

5. Invalid → discard, flag relay

```
A malicious relay can return **stale but validly signed** data. It cannot forge new data.
Staleness is handled by Tier 3.

<a id="services-mhr-dht--tier-3-multi-source-queries-anti-censorship-anti-staleness"></a>

### Tier 3 — Multi-Source Queries (Anti-Censorship, Anti-Staleness)
```

> For lookups where censorship or staleness matters, the client queries multiple independent nodes:

Multi-source lookup (quorum_size N):

1. Send lookup to N independent nodes (relay + N-1 others from Ring 0/1)

2. Collect responses with timeout

3. Content-addressed: any valid response is sufficient

4. Mutable: highest lamport_timestamp with valid signature wins

5. "Not found" accepted only if unanimous across all N

6. Divergent results: flag dissenting node(s), trust majority

```
Default quorum sizes:

| Lookup Type | Default N | Notes |
├──────────────┼────────────┼────────┤
| Content-addressed | 1 | Hash verification is sufficient |
| Name resolution | 2 | First resolution of unknown name uses N=3 |
| Mutable object | 2 | N=3 if freshness is critical |
| Service discovery | 1 | N=2 if results seem incomplete |

<a id="services-mhr-dht--trusted-relay-shortcut"></a>

### Trusted Relay Shortcut

If the client's relay is in its [trust graph](#economics-trust-neighborhoods), single-
source queries (N=1) are sufficient for all tiers. Multi-source queries are only needed for
untrusted relays. A trusted relay has economic skin in the game — trust means absorbing the
trusted node's debts, making dishonesty self-punishing.

<a id="services-mhr-dht--overhead"></a>

### Overhead

| Scenario | Extra Queries | Extra Bandwidth |
├───────────┼────────────────┼──────────────────┤
| Content-addressed, any relay | 0 | 0 |
| Mutable, trusted relay | 0 | 0 |
| Mutable, untrusted relay | +1 | ~192 bytes |
| Name resolution | +1 | ~192 bytes |



<div class="page-break"></div>


<a id="services-mhr-pub"></a>
```

```
# MHR-Pub: Publish/Subscribe

MHR-Pub provides a publish/subscribe system for real-time notifications across the mesh. It
supports multiple subscription types and delivery modes, allowing applications to choose
the right tradeoff between immediacy and bandwidth.

<a id="services-mhr-pub--subscriptions"></a>

## Subscriptions
```

Subscription { subscriber: NodeID, topic: enum { Key(hash), // specific key changed Prefix(hash_prefix), // any key with prefix changed Node(NodeID), // any publication by this node Neighborhood(label), // any publication in this community label }, delivery: enum { Push, // immediate, full payload Digest, // batched summaries, periodic PullHint, // hash-only notification }, }

```
<a id="services-mhr-pub--subscription-topics"></a>

## Subscription Topics

| Topic Type | Use Case |
├──────────────┼──────────────┤
| **Key** | Watch a specific data object for changes (e.g., a friend's profile) |
| **Prefix** | Watch a category of keys (e.g., all posts in a forum) |
| **Node** | Follow all publications from a specific user |
| **Neighborhood** | Watch all activity from nodes with a given community label |

<a id="services-mhr-pub--delivery-modes"></a>

## Delivery Modes

<a id="services-mhr-pub--push"></a>

### Push

Full payload delivered immediately when published. Best for high-bandwidth links where
real-time updates matter.

**Use on**: WiFi, Ethernet, Cellular

<a id="services-mhr-pub--digest"></a>

### Digest

Batched summaries delivered periodically. Reduces bandwidth by aggregating multiple updates
into a single digest.

**Use on**: Moderate bandwidth links, or when real-time isn't critical

<a id="services-mhr-pub--pullhint"></a>
```

PullHint

Only the hash of new content is delivered. The subscriber decides whether and when to pull the full data.

Use on: LoRa and other constrained links where bandwidth is precious

Application-Driven Delivery Selection

Delivery mode selection is the **application's responsibility**, informed by link quality. The protocol provides tools; the application decides:

// Application code (pseudocode) let link = query_link_quality(publisher_node);

if link.bandwidth_bps > 1_000_000 { subscribe(topic, Push); // WiFi: get everything immediately } else if link.bandwidth_bps > 10_000 { subscribe(topic, Digest); // moderate: batched summaries } else { subscribe(topic, PullHint); // LoRa: just tell me what's new }

The pub/sub system doesn't make this decision — the application does, based on `query_link_quality()` from the capability layer.

Bandwidth Characteristics

Delivery Mode	Per-notification overhead	Suitable for
Push	Full object size	WiFi, Ethernet
Digest	~50 bytes per item (hash + summary)	Moderate links
PullHint	~32 bytes (hash only)	LoRa, constrained links

<div class="page-break"></div>

MHR-Compute: Contract Execution

MHR-Compute provides a restricted execution environment for data validation, state transitions, and access control. It supports two execution tiers: MHR-Byte (a minimal bytecode for constrained devices) and WASM (for capable nodes).

MHR-Byte: Minimal Bytecode

MHR-Contract { hash: Blake3Hash, code: Vec, // MHR-Byte bytecode max_memory: u32, max_cycles: u64, max_state_size: u32, state_key: Hash, // current state in MHR-Store functions: [FunctionSignature], }

```
MHR-Byte is a minimal bytecode with a ~50 KB interpreter, designed to run on constrained
devices like the ESP32. It supports:

| Capability | Description |
├───────────┼────────────┤
| **Cryptographic primitives** | Hash, sign, verify |
| **CRDT operations** | Merge, compare |
| **CBOR/JSON manipulation** | Structured data processing |
| **Bounded control flow** | Loops with hard cycle limits |

MHR-Byte explicitly **does not** support:
- I/O operations
- Network access
- Filesystem access
- Unbounded computation

All execution is **pure deterministic computation**. Given the same inputs, any node
running the same contract produces the same output. This is what makes [verification]
(#marketplace-verification) possible.

<a id="services-mhr-compute--opcode-set-47-opcodes"></a>

### Opcode Set (47 Opcodes)

| Category | Opcodes | Cycle Cost | Description |
├──────────┼─────────┼────────────┼─────────────┤
| **Stack** (6) | PUSH, POP, DUP, SWAP, OVER, ROT | 1 | Stack manipulation |
| **Arithmetic** (9) | ADD, SUB, MUL, DIV, MOD, NEG, ABS, MIN, MAX | 1-3 | 64-bit integer,
overflow traps |
| **Bitwise** (6) | AND, OR, XOR, NOT, SHL, SHR | 1 | Bitwise operations |
| **Comparison** (6) | EQ, NEQ, LT, GT, LTE, GTE | 1 | Pushes 0 or 1 |
| **Control** (7) | JMP, JZ, JNZ, CALL, RET, HALT, ABORT | 2-5 | Bounded control flow |
| **Crypto** (3) | HASH, VERIFY_SIG, VERIFY_VRF | 500-2000 | Blake3, Ed25519, ECVRF |
| **System** (10) | BALANCE, SENDER, SELF, EPOCH, TRANSFER, LOG, LOAD, STORE, MSIZE, EMIT |
2-50 | State access and side effects |

**Cycle cost model**: The base unit is 1 cycle ≈ 1 μs on ESP32 (the reference platform).
Faster hardware executes more cycles per wall-clock second but charges the same cycle cost
per opcode. Gas price in μMHR/cycle is set by each compute provider in their capability
advertisement.

**Specification approach**: The reference interpreter (in Rust) serves as the authoritative
specification. A comprehensive test vector suite ensures cross-platform conformance. Formal
specification (Yellow Paper-style) is deferred until the opcode set stabilizes through
real-world usage.

<a id="services-mhr-compute--wasm-full-execution"></a>
```

```
## WASM: Full Execution

Gateway nodes and more capable hardware can offer full WASM (WebAssembly) execution as an
additional compute capability. A contract declares its WASM requirement tier:
```

Contract execution path:

1. Contract specifies: wasm_tier: None → Can run on any node with MHR-Byte interpreter (~50 KB)

2. Contract specifies: wasm_tier: Light → Requires Community-tier or above (Pi Zero 2W+) → 16 MB memory limit, 10^8 fuel limit, 5 second wall-clock

3. Contract specifies: wasm_tier: Full → Requires Gateway-tier or above (Pi 4/5+) → 256 MB memory limit, 10^10 fuel limit, 30 second wall-clock → Delegated via capability marketplace if local node can't execute

```
<a id="services-mhr-compute--wasm-sandbox"></a>

### WASM Sandbox

The WASM execution environment uses **Wasmtime** (Bytecode Alliance, Rust-native) as the
reference runtime. Wasmtime provides AOT compilation on Gateway+ nodes, fuel-based
execution metering that maps to MHR-Byte cycle accounting, and configurable memory limits
per contract.
```

WasmSandbox { runtime: Wasmtime, max_memory: u32, // from contract's max_memory field max_fuel: u64, // from contract's max_cycles (1 fuel ≈ 1 MHR-Byte cycle) max_wall_time_ms: u32, // 5,000 (Light) or 30,000 (Full) }

```
**Host imports**: WASM contracts call back into the Mehr system through a restricted host
API mirroring the MHR-Byte System opcodes:

| Host Function | MHR-Byte Equivalent | Fuel Cost |
├───────────────┼──────────────────────┼───────────────┤
| `mehr_balance(node_id) → u64` | BALANCE | 10 |
| `mehr_sender() → [u8; 16]` | SENDER | 2 |
| `mehr_self() → [u8; 16]` | SELF | 2 |
| `mehr_epoch() → u64` | EPOCH | 5 |
| `mehr_transfer(to, amount) → bool` | TRANSFER | 50 |
| `mehr_log(data)` | LOG | 10 |
| `mehr_store_load(key) → Vec<u8>` | LOAD | 3 |
| `mehr_store_save(key, value)` | STORE | 3 |
| `mehr_hash(data) → [u8; 32]` | HASH | 500 |
| `mehr_verify_sig(pubkey, msg, sig) → bool` | VERIFY_SIG | 1000 |

No other host imports are available. WASM contracts cannot access the filesystem, network,
```

```
    clock, or random number generator — all execution remains pure and deterministic.


    <a id="services-mhr-compute--light-wasm-community-tier"></a>


    ### Light WASM (Community Tier)


    Community-tier devices (Pi Zero 2W, 512 MB RAM) support a restricted WASM profile: 16 MB
    max memory, 10^8 max fuel, 5-second wall-clock limit, interpreted via Cranelift baseline
    (no AOT). Contracts exceeding Light WASM limits are automatically delegated to a more
    capable node via [compute delegation](#services-mhr-compute--compute-delegation).


    <a id="services-mhr-compute--compute-delegation"></a>


    ## Compute Delegation


    If a node can't execute a contract locally, it delegates to a capable neighbor via the
    [capability marketplace](#marketplace-overview):
```

Delegation flow:

1. Node receives request to execute contract

2. Node checks: can I run this locally?

3. If no: query nearby capabilities for compute

4. Find a provider, form agreement, send execution request

5. Receive result, verify (per agreement's proof method)

6. Return result to requester

```
    This is transparent to the original requester — they don't need to know whether their
    contract ran locally or was delegated.


    <a id="services-mhr-compute--heavy-compute-as-capabilities"></a>


    ## Heavy Compute as Capabilities


    ML inference, transcription, translation, text-to-speech, and any other heavy computation
    are **not protocol primitives**. They are compute capabilities offered by nodes that have
    the hardware:
```

A GPU node advertises: offered_functions: [{ function_id: hash("whisper-small"), cost: 50 μMHR/minute }, { function_id: hash("piper-tts"), cost: 30 μMHR/minute },]

```
    A consumer requests execution of that function through the standard compute delegation
    path. The protocol is **agnostic to what the function does** — it only cares about
    discovery, negotiation, execution, verification, and payment.


    <a id="services-mhr-compute--contract-use-cases"></a>
```

```
## Contract Use Cases

| Application | Contract Purpose |
|────────────|──────────────────|
| **Naming** | Community-label-scoped name resolution (`maryam@tehran-mesh` → NodeID) |
| **Forums** | Append-only log management, moderation rules |
| **Marketplace** | Listing validation, escrow logic |
| **Wiki** | CRDT merge rules for collaborative documents |
| **Group messaging** | Symmetric key rotation, member management |
| **Access control** | Permission checks for mutable data objects |

<a id="services-mhr-compute--resource-limits"></a>

## Resource Limits

Every contract declares its resource bounds upfront:

- **max_memory**: Maximum memory allocation
- **max_cycles**: Maximum CPU cycles before forced termination
- **max_state_size**: Maximum persistent state

These limits are enforced by the runtime. A contract that exceeds its declared limits is
terminated immediately. This prevents denial-of-service through runaway computation.

<a id="services-mhr-compute--private-compute-optional"></a>

## Private Compute (Optional)

By default, compute delegation has **no input privacy** — the compute node sees your input
and produces a result. This is fine for most workloads (contract execution, public data
processing, non-sensitive queries). But for sensitive data — medical records, private
messages, financial analysis — you need the compute node to process data it cannot read.

Private compute is **opt-in per agreement**. The consumer chooses a privacy tier based on
sensitivity and willingness to pay:
```

CapabilityAgreement { ... privacy: enum { None, // default — compute node sees input/output SplitInference, // model partitioned, node sees only middle layers SecretShared, // input split across multiple nodes TEE, // hardware-attested secure enclave }, }

```
<a id="services-mhr-compute--tier-0-no-privacy-default"></a>

### Tier 0: No Privacy (Default)

The compute node receives plaintext input, executes, and returns the result. Verification
is via [result hash](#marketplace-verification) or redundant execution. This is the
cheapest and fastest option.

**Use for**: Public data, non-sensitive queries, contract execution, anything where the
input isn't secret.
```

```
<a id="services-mhr-compute--tier-1-split-inference"></a>

### Tier 1: Split Inference

For ML/AI workloads. The neural network is partitioned across nodes so no single node sees
both the raw input and the final output:
```

Split inference flow:

1. Consumer runs first 1-3 layers locally (transforms raw input)

2. Intermediate activations are sent to Inference node (optionally with calibrated DP noise for formal privacy guarantees)

3. Inference node runs the heavy middle layers

4. Intermediate result sent back to consumer

5. Consumer runs final 1-2 layers locally (produces final output)

What the Inference node sees: ✗ Raw input (transformed by early layers) ✗ Final output (produced by consumer's final layers) ✓ Intermediate activations (a compressed, transformed representation)

```
**Overhead**: ~1.2-2x latency vs plaintext. Bandwidth for activation transfer at cut
points. Consumer needs enough compute for a few neural network layers (Gateway tier or
above).

**Privacy strength**: Moderate. Adding differential privacy noise to activations at cut
points provides formal (ε, δ)-privacy guarantees at the cost of some accuracy (2-15%
depending on privacy budget).

**Use for**: AI inference on personal data — voice transcription, document analysis, image
processing — where the compute node shouldn't see the raw content.

<a id="services-mhr-compute--tier-2-secret-shared-computation"></a>

### Tier 2: Secret-Shared Computation

Input data is split using Shamir's Secret Sharing into N shares, each sent to a different
compute node. No individual node can reconstruct the input.
```

Secret-shared compute flow:

1. Consumer splits input into 3 shares (2-of-3 threshold)

2. Each share sent to a different compute node

3. Each node computes on its share independently

   - Additions and scalar multiplications: free (local computation)

   - Multiplications between secrets: one communication round between nodes

4. Consumer collects result shares and reconstructs the output

What each compute node sees: ✗ Original input (only a random-looking share) ✗ Other nodes' shares ✗ Final output ✓ Its own share (information-theoretically meaningless alone)

```
**Overhead**: 3x bandwidth (3 shares), 3x compute cost (3 nodes). Linear operations are
nearly free; non-linear operations require inter-node communication.

**Trust assumption**: At most 1 of 3 nodes may be malicious (honest majority). The consumer
selects 3 nodes from different trust neighborhoods to minimize collusion risk.

**Best for**: Linear/affine workloads — aggregation, statistics, linear classifiers, search
queries. For neural networks with many non-linear layers, combine with Tier 1: secret-share
the input, run the first layers as MPC on shares, then switch to split inference for the
deep non-linear layers.

**Use for**: Medical data analysis, private search, financial computation — anything where
the input must remain hidden from all compute providers.

<a id="services-mhr-compute--tier-3-tee-hardware-attested"></a>

### Tier 3: TEE (Hardware-Attested)

Compute runs inside a Trusted Execution Environment (AMD SEV-SNP, NVIDIA H100 Confidential
Computing, or ARM CCA). The hardware enforces that even the node operator cannot read the
data being processed.
```

TEE compute flow:

1. Consumer discovers a node advertising TEE capability

2. Consumer requests and verifies a remote attestation report (proves specific code is running inside a genuine TEE)

3. Consumer sends encrypted input (encrypted to the TEE's ephemeral key)

4. TEE decrypts, processes, encrypts output for consumer

5. Consumer decrypts result

What the node operator sees: ✗ Input (encrypted for the TEE) ✗ Output (encrypted for the consumer) ✗ Intermediate state (protected by hardware) ✓ That a computation happened, its duration, and data sizes

```
**Overhead**: Under 5% compute overhead. Near-zero bandwidth overhead. Requires server-
grade hardware (AMD EPYC, NVIDIA H100).

**Trust assumption**: You trust the hardware vendor (AMD, Intel, NVIDIA) to have correctly
implemented the TEE. You do NOT trust the node operator.
```

Limitation: Only available on Inference-tier nodes with server hardware. Not available on ESP32, Raspberry Pi, or consumer hardware.

Use for: Highest-sensitivity workloads — end-to-end encrypted AI inference, confidential data processing — where you're willing to trust the hardware vendor but not the node operator.

Choosing a Privacy Tier

Tier	Overhead	Privacy	Hardware Required	Cost
None	1x	None	Any	Cheapest
Split Inference	1.2–2x	Moderate (DP-configurable)	Consumer: Gateway+. Provider: any.	Low premium
Secret Shared	3x+	Strong (information-theoretic)	3 compute nodes	3x compute cost
TEE	~1x	Strong (hardware-attested)	Provider: server-grade with TEE	Slight premium

The default is **no privacy**. Most compute delegation doesn't need it — you're running a public contract on public data, or the result hash verification is sufficient. Private compute is for when the **input itself** is sensitive.

Combining Tiers

Tiers can be combined for defense in depth:

- **Split + Secret Shared**: Secret-share the input, run first layers as MPC across 3 nodes, then split inference for deep layers. Maximum software-based privacy.
- **Split + TEE**: Run the heavy middle layers inside a TEE. The TEE never sees raw input (early layers run locally), and you get hardware attestation for the critical computation.

The consumer specifies the desired combination in the capability agreement. The marketplace handles discovery of nodes that support the requested privacy tier.

<div class="page-break"></div>

Messaging

End-to-end encrypted, store-and-forward messaging built on the Mehr service primitives.

Architecture

```
Messaging composes multiple service layers:

| Component | Built On |
├──────────────┼──────────────┤
| Message storage & persistence | [MHR-Store](#services-mhr-store) |
| Delivery notifications | [MHR-Pub](#services-mhr-pub) |
| Transport encryption | Link-layer encryption (Reticulum-derived) |
| End-to-end encryption | [E2E encryption](#protocol-security--end-to-end-encryption-data-
payloads) |

<a id="applications-messaging--how-it-works"></a>

## How It Works

1. **Compose**: Alice writes a message to Bob
2. **Encrypt**: Message encrypted end-to-end for Bob's public key
3. **Store**: Encrypted message stored as an immutable DataObject in MHR-Store
4. **Notify**: MHR-Pub sends a notification to Bob (or his relay nodes)
5. **Deliver**: If Bob is online, he retrieves immediately. If offline, relay nodes cache
the message for later delivery.
6. **Pay**: Relay and storage fees paid automatically via [payment channels](#economics-
payment-channels)

<a id="applications-messaging--offline-delivery"></a>

## Offline Delivery

Relay nodes cache messages for offline recipients. When Bob comes back online:

1. His nearest relay nodes inform him of pending messages
2. He retrieves and decrypts them
3. The relay nodes are paid for the storage duration

This is store-and-forward messaging — similar to email, but encrypted and decentralized.

<a id="applications-messaging--group-messaging"></a>

## Group Messaging

Group messages use shared symmetric keys managed by an MHR-Compute contract:
```

GroupState { group_id: Blake3Hash, members: Set, current_key: ChaCha20Key, // current group symmetric key key_epoch: u64, // increments on every rotation admin: NodeID, // creator; can add/remove members co_admins: Vec, // up to 3 delegated co-admins admin_sequence: u64, // monotonic counter for admin operations }

CoAdminCertificate { co_admin: NodeID, permissions: enum { Full, MembersOnly, RotationOnly }, granted_by: NodeID, // must be the group creator signature: Ed25519Signature, // creator's signature over (group_id, co_admin, permissions) }

```
<a id="applications-messaging--key-management"></a>
```

Key Management

- **Creation**: The group creator generates the first symmetric key and encrypts it individually for each member's public key (standard E2E envelope per member)
- **Rotation**: When a member joins or leaves, the admin (or any authorized co-admin) generates a new key and distributes it to all current members. The key epoch increments. Old keys are retained locally so members can decrypt historical messages
- **No forward secrecy for groups**: A new member receives only the current key — they cannot decrypt messages sent before they joined. A removed member retains old keys for messages they already received but cannot decrypt new messages (new key was never sent to them)
- **Maximum group size**: Practical limit of ~100 members, constrained by key distribution bandwidth (each rotation sends one E2E-encrypted key envelope per member, ~100 bytes each)

```
<a id="applications-messaging--co-admin-delegation"></a>
```

Co-Admin Delegation

The group creator can delegate admin authority to up to 3 co-admins via signed `CoAdminCertificate` records. This solves the single-admin availability problem without requiring threshold cryptography.

- **Any co-admin can independently**: add/remove members, rotate the group key, and (if granted `Full` permission) promote/demote other co-admins
- **Conflict resolution**: All admin operations carry a monotonically increasing `admin_sequence` number. If two co-admins issue conflicting operations (e.g., simultaneous key rotations), members accept the operation with the highest sequence number. Ties are broken by lowest admin public key hash
- **No threshold crypto**: Co-admin delegation uses only Ed25519 signatures — no multi-round key generation protocols, no new cryptographic primitives. Each delegation certificate is ~128 bytes
- **Graceful degradation**: If all admins go offline, the group continues functioning with its current key. No key rotation or membership changes occur until at least one admin returns

```
<a id="applications-messaging--bandwidth-on-lora"></a>
```

Bandwidth on LoRa

A 1 KB text message over LoRa takes approximately 10 seconds to transmit — comparable to SMS delivery times. This is viable for text-based communication in constrained environments.

Attachments are DataObjects with `min_bandwidth` set appropriately. A photo attachment might declare `min_bandwidth: 10000` (10 kbps), meaning it will transfer when the recipient has a WiFi link available but won't be attempted over LoRa.

```
<div class="page-break"></div>
```

```
<a id="applications-social"></a>
```

Social

A decentralized social network built on Mehr primitives. No central servers, no algorithmic recommendations — just chronological feeds assembled locally from followed users.

```
<a id="applications-social--architecture"></a>
```

Architecture

Each user has:

- **Profile**: A mutable DataObject containing display name, bio, avatar hash, etc.
- **Feed**: An append-only log of posts, where each post is an immutable DataObject
- **Followers**: Subscribers are tracked via MHR-Pub subscriptions

```
<a id="applications-social--feed-assembly"></a>
```

Feed Assembly

Feed aggregation is entirely local. Each device:

1. Maintains a list of followed users (NodeIDs)
2. Subscribes to each followed user via [MHR-Pub](#services-mhr-pub)
3. Receives notifications when followed users publish new posts
4. Assembles the timeline locally in chronological order

There is no algorithmic recommendation. No engagement optimization. Just a reverse-chronological feed of content from people you follow.

```
<a id="applications-social--media-tiering"></a>
```

Media Tiering

Media adapts to available bandwidth using `min_bandwidth` on DataObjects:

Content Type	Size	min_bandwidth	Available On
Text post	~200 bytes	0 (any link)	Everywhere
Blurhash thumbnail	~64 bytes	0	Everywhere, including LoRa
Compressed image	~50 KB	10,000 (10 kbps)	WiFi and above
Full resolution image	~500 KB	100,000 (100 kbps)	WiFi and above
Video	>1 MB	1,000,000 (1 Mbps)	High-bandwidth links only

The application decides which tier to request based on current link quality:

let link = query_link_quality(author_node);

if link.bandwidth_bps < 1000 { // LoRa: text + blurhash only fetch(post.text); fetch(post.blurhash_thumbnail); } else if link.bandwidth_bps < 100_000 { // Moderate: add compressed images fetch(post.compressed_image); } else { // High bandwidth: full resolution fetch(post.full_image); }

```
<a id="applications-social--privacy"></a>

## Privacy

- Posts can be public (replicated broadly) or neighborhood-scoped (visible only within a
trust neighborhood)
- No central server has a copy of anyone's social graph
- Following is a local operation — only you and the person you follow need to know
- Unfollowing is purely local — just stop subscribing



<div class="page-break"></div>



<a id="applications-voice"></a>

# Voice

Voice communication in Mehr ranges from push-to-talk over LoRa to full-duplex calls over
WiFi, adapting to available bandwidth.

<a id="applications-voice--codec-selection-by-link-quality"></a>

## Codec Selection by Link Quality

| Link Type | Codec | Bitrate | Mode |
├─────────────┼────────┼──────────────┼─────────────┤
| LoRa (10+ kbps) | Codec2 | 700-3,200 bps | Push-to-talk |
| WiFi / Cellular | Opus | 6-510 kbps | Full-duplex |

<a id="applications-voice--codec2-on-lora"></a>

### Codec2 on LoRa

Codec2 is an open-source voice codec designed for very low bitrates. At 700 bps, it
produces intelligible speech — not high-fidelity, but functional for communication. At
3,200 bps, quality is similar to AM radio.

A 10 kbps LoRa link has enough bandwidth for Codec2 push-to-talk with room for protocol
overhead.

<a id="applications-voice--opus-on-wifi"></a>

### Opus on WiFi
```

```
On higher-bandwidth links, Opus provides near-CD-quality voice with full-duplex operation
(both parties can talk simultaneously).


<a id="applications-voice--encryption"></a>


## Encryption

Voice streams are **end-to-end encrypted** using the standard [E2E encryption](#protocol-
security--end-to-end-encryption-data-payloads) mechanism — each session generates an
ephemeral X25519 keypair, and the symmetric session key is derived from a Diffie-Hellman
exchange with the remote party's public key. Relay nodes carry encrypted voice packets they
cannot decrypt.


<a id="applications-voice--bandwidth-bridging"></a>


## Bandwidth Bridging

When participants are on different link types, the application can use compute delegation
to bridge:
```

Scenario: Alice is on LoRa, Bob is on WiFi

Option 1: Codec conversion Alice sends Codec2 audio over LoRa to a bridge node Bridge node transcodes Codec2 → Opus Bridge node sends Opus audio to Bob over WiFi

Option 2: Speech-to-text bridging Alice sends Codec2 audio over LoRa A nearby compute node runs STT (Whisper) on the audio Text is sent to Bob over WiFi Bob's device optionally runs TTS to play it as audio

```
This is an **application-level decision** using standard [compute delegation](#marketplace-
agreements). The protocol has no concept of "voice" — it routes bytes. The application
decides how to adapt between bandwidth tiers.


<a id="applications-voice--push-to-talk-protocol"></a>


## Push-to-Talk Protocol

On half-duplex links (LoRa), push-to-talk works as:

1. Sender presses talk button
2. Audio captured, encoded with Codec2
3. Encoded frames sent as a stream of small packets
4. Receiver buffers and plays back
5. Sender releases button, receiver can now respond

The protocol handles this as ordinary data packets — there is no special voice channel.



<div class="page-break"></div>
```

```
<a id="applications-naming"></a>

# Naming (MHR-Name)

MHR-Name provides human-readable names scoped to community labels. There is no global
namespace — names resolve locally based on proximity and trust.

<a id="applications-naming--community-scoped-names"></a>

## Community-Scoped Names

Names follow the format `name@community-label`:
```

maryam@tehran-mesh alice@portland-mesh relay-7@backbone-west

```
Resolution works by proximity:

1. Parse the community label from the name (`portland-mesh`)
2. Search nearby nodes with that `community_label` set
3. Within that cluster, resolve the name via local naming records
4. Use the returned NodeID for routing

<a id="applications-naming--why-no-global-namespace"></a>

## Why No Global Namespace?

A global namespace requires global consensus on name ownership. Global consensus
contradicts Mehr's partition tolerance requirement. If two partitioned networks both
register the name "alice," there is no partition-safe way to resolve the conflict.

Community-scoped names solve this:
- Names are locally consistent within their trust neighborhood
- Different communities can have different "alice" users — they are `alice@portland-mesh`
and `alice@tehran-mesh`
- No global coordination needed
- Community labels are self-assigned, not centrally managed

<a id="applications-naming--name-registration"></a>

## Name Registration

Names are registered locally by announcing a name binding:
```

NameBinding { name: String, // "alice" community_label: String, // "portland-mesh" node_id:
NodeID, signature: Ed25519Signature, }

```
Name bindings propagate via gossip within the trust neighborhood. Conflicts (two nodes
```

claiming the same name in the same community label) are resolved by precedence:

1. **Trust-weighted** (highest priority) — if one claimant is in your trust graph and the other is not, the trusted claimant wins regardless of timing. Between two trusted claimants, the one with the shorter trust distance wins.
2. **First-seen** (tiebreaker) — if both claimants have equal trust status (both trusted at the same distance, or both untrusted), the first binding your node received wins.
3. **Local petnames** (ultimate fallback) — if you need guaranteed resolution regardless of conflicts, assign a local petname (see below). Petnames override all network name resolution.

Cross-Community Resolution

To resolve a name in a different community:

1. Query propagates toward nodes with the target community label
2. Nearest matching cluster responds with the name binding
3. Result is cached locally with a TTL

Because community labels are not unique, resolution finds the **nearest** cluster with that label. This is usually what you want — `alice@portland-mesh` resolves to the Portland cluster nearest to you.

Local Petnames

As a fallback and privacy feature, each user can assign **petnames** — local nicknames for NodeIDs on their own device:

User's petname mapping (local only, never shared): "Mom" → 0x3a7f...b2c1 "Work" → 0x8e2d...f4a9 "Doctor" → 0x1b5c...d3e7

```
Petnames are:
- Completely private (stored only on the user's device)
- Not resolvable by anyone else
- Useful when community names aren't available or desired
- The most censorship-resistant naming possible — no one can take your petnames from you



<div class="page-break"></div>



<a id="applications-community-apps"></a>

# Community Applications

Community applications — forums, marketplaces, and wikis — are built on [MHR-Compute]
```

(#services-mhr-compute) contracts managing CRDT state. All degrade gracefully to text-only on constrained links.

Forums

Forums are append-only logs managed by moderation contracts:

- **Posts** are immutable DataObjects, appended to a topic log
- **Moderation** is handled by an MHR-Compute contract that enforces community rules
- **Threading** is local — each client assembles thread views from the flat log
- **Propagation** uses neighborhood-scoped gossip for local forums, or wider replication for public forums

Moderation Model

The moderation contract defines:
- Who can post (trusted peers, vouched users, anyone)
- Content rules (enforced at the contract level)
- Moderator keys (who can remove content)
- Appeal mechanisms

Since moderation is a contract, different forums can have different moderation policies. There is no platform-wide content policy.

Marketplace

Marketplace listings are DataObjects with neighborhood-scoped propagation:

- **Listings** are mutable DataObjects (sellers can update price, availability)
- **Search** is local — each node indexes listings it has received
- **Transactions** happen off-protocol (physical exchange, external payment) or through MHR escrow contracts
- **Reputation** feeds back into the node's general [reputation score](#protocol-security--sybil-resistance)

Escrow

For MHR-denominated transactions, an escrow contract can hold payment:

1. Buyer deposits MHR into escrow contract
2. Seller delivers goods/services
3. Buyer confirms delivery
4. Contract releases payment to seller
5. Disputes resolved by community moderators (trusted peers with moderator keys)


```
## Wiki

Wikis are CRDT-merged collaborative documents:

- **Pages** are mutable DataObjects using CRDT merge rules
- **Concurrent edits** merge automatically without conflicts (using operational transforms
or CRDT text types)
- **History** is preserved as a chain of immutable snapshots
- **Permissions** managed by an MHR-Compute contract (who can edit, who can view)

<a id="applications-community-apps--bandwidth-degradation"></a>

## Bandwidth Degradation

All community applications degrade to text-only on constrained links:

| Application | Full Experience | LoRa Experience |
├───────────────┼──────────────────────┼──────────────────┤
| Forums | Rich text, images, threads | Text posts, flat view |
| Marketplace | Photos, maps, categories | Text listings |
| Wiki | Formatted text, images, tables | Plain text |

The application layer handles this adaptation using `query_link_quality()` - the protocol
doesn't need to know about the application's content format.



<div class="page-break"></div>


<a id="applications-hosting"></a>

# Hosting on Mehr

Traditional web hosting requires a server, a domain name, a certificate, and ongoing
payment to a hosting provider. On Mehr, hosting is just storing DataObjects and letting the
network serve them. No server. No DNS. No certificate authority.

<a id="applications-hosting--hosting-a-website"></a>

## Hosting a Website

A "website" on Mehr is a collection of [DataObjects](#services-mhr-store) - HTML, CSS,
JavaScript, images - stored in MHR-Store and addressed by content hash or human-readable
[MHR-Name](#applications-naming).

<a id="applications-hosting--static-site"></a>

### Static Site
```

Site structure: root = DataObject { hash: Blake3("index.html contents"), content_type: Immutable, payload: Inline("...links to sub-objects..."), replication: Network(5), // 5 copies across the network }

Sub-objects: style.css → DataObject { hash: ..., replication: Network(5) } logo.png → DataObject { hash: ..., replication: Network(3) } about.html → DataObject { hash: ..., replication: Network(5) }

```
A visitor retrieves the root DataObject by name (`mysite@portland-mesh`) or by hash. The
root object links to sub-objects by their content hashes. The visitor's node fetches each
sub-object from the nearest replica in the mesh.

<a id="applications-hosting--how-visitors-access-your-site"></a>

### How Visitors Access Your Site
```

Access flow:

1. Visitor queries MHR-Name: "mysite@portland-mesh"

2. Name resolves to root DataObject hash

3. Visitor's node fetches root from nearest MHR-Store replica

4. Root contains hashes of sub-objects (CSS, images, etc.)

5. Visitor's node fetches sub-objects (in parallel, from different replicas)

6. Local browser or app renders the content

```
Content-addressed storage means:
- **No single server** — content is served from whichever node has a copy
- **No downtime** — as long as any replica is reachable, the site is available
- **No tampering** — content hash guarantees integrity
- **Global CDN by default** — popular content gets cached everywhere automatically

<a id="applications-hosting--updating-your-site"></a>

### Updating Your Site

For static content, publish new DataObjects and update the MHR-Name binding to point to the
new root hash. Old content remains available at its hash (immutable), but the name now
resolves to the new version.

For dynamic content (blog, profile, etc.), use **mutable DataObjects**:
```

Blog post feed: feed = DataObject { content_type: Mutable, owner: your_node_id, payload: Inline([post_hash_1, post_hash_2, ...]), // Owner can update the post list by publishing a new version // Each individual post is immutable — only the feed index changes }

```
<a id="applications-hosting--hosting-a-social-feed"></a>

## Hosting a Social Feed
```

```
A social feed is an append-only log of posts, served via [MHR-Pub](#services-mhr-pub):
```

Your feed:

1. Profile: Mutable DataObject (name, bio, avatar hash)

2. Posts: Immutable DataObjects (each post is permanent)

3. Feed index: Mutable DataObject listing post hashes in order

4. Subscriber notifications via MHR-Pub

```
<a id="applications-hosting--publishing-a-post"></a>

### Publishing a Post
```

Publishing flow:

1. Create an immutable DataObject for the post content

2. Store it in MHR-Store with replication

3. Update your feed index (mutable) to include the new post hash

4. MHR-Pub notifies all subscribers of the update

```
<a id="applications-hosting--following-someone"></a>

### Following Someone
```

Following flow:

1. Subscribe to their feed via MHR-Pub (topic: Node(their_node_id))

2. Choose delivery mode based on your link quality:

   ○ WiFi: Push (full content immediately)

   ○ Moderate: Digest (batched summaries)

   ○ LoRa: PullHint (hash-only, pull content when convenient)

3. Your device assembles your timeline locally from all followed feeds

```
No server assembles your feed. No algorithm decides what you see. Your device pulls from
the people you follow, in chronological order.

<a id="applications-hosting--cost-of-hosting"></a>

## Cost of Hosting
```

```
| What | Cost | Notes |
├───────┼────────┼────────┤
| Store a 10 KB page | ~50 µMHR/month | With 5 replicas |
| Store a 1 MB image | ~5,000 µMHR/month | With 3 replicas |
| MHR-Name registration | Free | First-seen-wins within your community label |
| Bandwidth when someone views your page | Paid by the viewer | You don't pay for serving —
viewers pay relay costs |


The key economic difference from traditional hosting: **you pay for storage, not for
traffic.** The viewer pays the relay cost to reach your content. Popular content is cheaper
to host because it gets widely cached.


<a id="applications-hosting--comparison-with-traditional-hosting"></a>


## Comparison with Traditional Hosting


| Aspect | Traditional Web | Mehr Hosting |
├───────┼─────────────────┼──────────────────┤
| **Server** | Required (or hosting provider) | None — content lives in the mesh |
| **Domain name** | Rent from registrar ($10-50/year) | MHR-Name (free, self-registered) |
| **SSL certificate** | Required (free via Let's Encrypt, or paid) | Not needed — all links
encrypted, content verified by hash |
| **Uptime** | Depends on your server/provider | Depends on replica count — more replicas =
higher availability |
| **Bandwidth costs** | You pay for traffic spikes | Viewers pay their own relay costs |
| **Censorship resistance** | Server can be seized, domain can be seized | No single point
to seize — content is replicated across the mesh |
| **Offline access** | Not possible | Cached content available even when original author is
offline |


<a id="applications-hosting--hosting-a-community-forum"></a>


## Hosting a Community Forum


A forum is a more complex application, but the primitives are the same:
```

Forum structure: Forum config: Mutable DataObject (rules, moderator keys) Topic list: Mutable DataObject (list of topic hashes) Each topic: Mutable DataObject (list of post hashes) Each post: Immutable DataObject (content + author signature) Moderation: MHR-Compute contract enforcing community rules

```
New posts are published as DataObjects, appended to the topic log, and propagated via
neighborhood-scoped MHR-Pub to all subscribers. Moderation rules are enforced by an MHR-
Compute contract — see [Community Apps](#applications-community-apps) for details.


<a id="applications-hosting--running-a-service"></a>


## Running a Service
```

Beyond static hosting, you can run persistent services on the network:

- **API endpoint**: An MHR-Compute contract that responds to requests
- **Bot/automation**: A node running custom logic, discoverable via the capability marketplace
- **Proxy service**: Bridge Mehr to the traditional web (serve Mehr content via HTTP, or fetch web content for mesh users)

Services are advertised as capabilities and discovered through the [marketplace] (#marketplace-overview). Consumers find your service, form agreements, and pay via payment channels — all handled by the protocol.

<div class="page-break"></div>

Hardware Reference Designs

Mehr is designed to run on hardware ranging from $30 solar-powered relays to GPU workstations. Every device participates at whatever level its hardware allows.

Device Tiers Overview

Tier	Hardware	Cost	Power	Primary Role
Minimal	ESP32 + LoRa SX1276	~$30	0.5W (solar)	Relay only
Community	Pi Zero 2 W + LoRa HAT + WiFi	~$100	3W	LoRa/WiFi bridge, basic compute
Gateway	Pi 4/5 + LoRa + cellular modem + SSD	~$300	10W	Internet uplink, storage, compute
Backbone	Mini PC + directional WiFi + fiber	~$500+	25W+	High-bandwidth backbone
Inference	x86 + GPU + Ethernet	~$500+	100W+	Heavy compute, ML inference

Minimal Relay Node

Target: Lowest-cost, always-on mesh relay

Components:

- ESP32-S3 microcontroller

- LoRa SX1276/SX1262 radio module

- Small solar panel (2W) + LiPo battery

- Weatherproof enclosure

Capabilities:

- Packet relay only
- MHR-Byte interpreter (~50 KB)
- No storage beyond routing tables
- 24/7 operation on solar power

Software:

- Mehr firmware (Rust, no_std)
- Transport: LoRa only
- Runs: routing, payment channels, gossip
- Cannot run: WASM, storage, heavy compute

```
**Earns from**: Routing fees (1-5 µMHR per packet relayed)

**Range**: 2-15 km line-of-sight with LoRa

<a id="hardware-reference-designs--community-bridge-node"></a>

## Community Bridge Node

**Target**: Bridge between LoRa mesh and local WiFi network
```

Components:

- Raspberry Pi Zero 2 W
- LoRa HAT (SX1262)
- Built-in WiFi
- SD card (32 GB)
- USB power supply (5V/2A)

Capabilities:

- LoRa ↔ WiFi bridging
- Basic MHR-Compute (MHR-Byte + light WASM)
- Local storage (~16 GB usable)
- MHR-DHT participation
- Message caching for offline nodes

Software:

- Mehr daemon (Rust, Linux)

- Dual transport: LoRa + WiFi
- Full protocol stack

```
**Earns from**: Bridging fees, compute delegation, storage

<a id="hardware-reference-designs--gateway-node"></a>

## Gateway Node

**Target**: Internet uplink for the mesh
```

Components:

- Raspberry Pi 4/5 (4 GB+ RAM)
- LoRa HAT
- 4G/LTE cellular modem (or Ethernet)
- SSD (256 GB+)
- Powered supply (12V)

Capabilities:

- Internet gateway (HTTP proxy, DNS relay)
- Full MHR-Store storage node
- MHR-DHT backbone participation
- Full WASM compute
- Epoch consensus participation

Software:

- Full Mehr stack
- Triple transport: LoRa + WiFi + Cellular/Ethernet
- Gateway proxy services

```
**Earns from**: Internet gateway fees, storage fees, compute fees, routing

<a id="hardware-reference-designs--backbone-node"></a>

## Backbone Node

**Target**: High-bandwidth infrastructure linking mesh segments
```

Components:

- Mini PC (Intel NUC or equivalent)
- Directional WiFi antenna (point-to-point)
- Fiber connection (where available)
- SSD (1 TB+)
- UPS/battery backup

Capabilities:

- High-throughput routing (100+ Mbps)
- Large-scale storage
- Full compute services
- Neighborhood discovery services
- Epoch consensus coordination

Software:

- Full Mehr stack, optimized for throughput
- Transport: Directional WiFi + Fiber + Ethernet

```
**Earns from**: Bulk routing fees, backbone transit, storage

<a id="hardware-reference-designs--inference-node"></a>

## Inference Node

**Target**: Heavy compute (ML inference, transcription, TTS)
```

Components:

- x86 PC or server
- GPU (NVIDIA RTX series or equivalent)
- Ethernet connection
- SSD (512 GB+)
- Standard power supply

Capabilities:

- ML model inference (Whisper, LLaMA, Stable Diffusion, etc.)
- Speech-to-text, text-to-speech
- Translation services
- Any GPU-accelerated computation

Software:

- Full Mehr stack

- WASM runtime + native GPU compute

- Model serving framework

- Advertises offered_functions with pricing

```
**Earns from**: Compute fees for ML inference and heavy processing



<div class="page-break"></div>



<a id="hardware-device-tiers"></a>

# Device Capabilities by Tier

Each hardware tier has different capabilities, which determine what the node can do on the
network and how it earns MHR.

<a id="hardware-device-tiers--capability-matrix"></a>

## Capability Matrix

| Capability | Minimal | Community | Gateway | Backbone | Inference |
├─────────────┼─────────┼───────────┼─────────┼──────────┼───────────┤
| Packet relay | Yes | Yes | Yes | Yes | Yes |
| LoRa transport | Yes | Yes | Yes | Optional | No |
| WiFi transport | No | Yes | Yes | Yes (directional) | Optional |
| Cellular transport | No | No | Yes | No | No |
| Ethernet/Fiber | No | No | Optional | Yes | Yes |
| MHR-Byte contracts | Yes | Yes | Yes | Yes | Yes |
| WASM contracts | No | Light | Full | Full | Full |
| MHR-Store storage | No | ~16 GB | ~256 GB | ~1 TB | ~512 GB |
| MHR-DHT participation | Minimal | Yes | Backbone | Backbone | Yes |
| Epoch consensus | No | No | Yes | Yes | Yes |
| Internet gateway | No | No | Yes | Optional | Optional |
| ML inference | No | No | No | No | Yes |
| Gossip participation | Full | Full | Full | Full | Full |
| Payment channels | Yes | Yes | Yes | Yes | Yes |

<a id="hardware-device-tiers--power-and-deployment"></a>

## Power and Deployment

| Tier | Power Draw | Power Source | Typical Deployment |
├──────────┼────────────┼──────────────┼────────────────────┤
| Minimal | 0.5W | Solar + LiPo | Outdoor, pole/tree-mounted |
| Community | 3W | USB wall adapter | Indoor, near window for LoRa |
```

```
| Gateway | 10W | 12V supply | Indoor, weatherproof enclosure |
| Backbone | 25W+ | Mains + UPS | Indoor, rack-mounted or desktop |
| Inference | 100W+ | Mains | Indoor, rack or desktop |
```


Earning Potential

What each tier naturally earns from:

Minimal (ESP32 + LoRa)
- Packet relay fees only
- Low per-packet revenue but high volume and zero operating cost (solar)
- Value: extends mesh range, maintains connectivity

Community (Pi Zero + LoRa + WiFi)
- Bridging fees (LoRa ↔ WiFi translation)
- Basic compute delegation
- Small-scale storage
- Value: connects LoRa mesh to local WiFi network

Gateway (Pi 4/5 + Cellular)
- Internet gateway fees (highest per-byte revenue)
- Storage fees
- Compute fees
- Epoch consensus participation
- Value: connects mesh to the wider internet

Backbone (Mini PC + Directional WiFi)
- High-volume transit routing
- Large-scale storage
- Full compute services
- Value: high-bandwidth links between mesh segments

Inference (GPU Workstation)
- ML inference fees (highest per-invocation revenue)
- Heavy compute services
- Value: provides advanced capabilities to the entire mesh

Delegation Patterns

Since nodes delegate what they can't do locally, natural delegation chains form:

Minimal relay → delegates everything except routing to → Community bridge → delegates bulk storage and internet to → Gateway node → delegates heavy compute to → Inference node

Each delegation is a bilateral [capability agreement](#marketplace-agreements) with payment flowing through [channels](#economics-payment-channels).

<div class="page-break"></div>

Implementation Roadmap

The Mehr implementation is organized into four phases, progressing from core networking fundamentals to a full ecosystem. Each phase has concrete milestones with acceptance criteria.

Phase 1: Foundation

Focus: Core networking and basic economics — the minimum to run a mesh with cost-aware routing and micropayments.

Milestone 1.1: Transport + Identity

- Implement `NodeIdentity` (Ed25519 keypair generation, destination hash derivation, X25519 conversion)
- Link-layer encryption (X25519 ECDH + ChaCha20-Poly1305, counter-based nonces, key rotation)
- Packet framing (Reticulum-compatible: header, addresses, context, data)
- Interface abstraction (LoRa, WiFi, serial — at least 2 transports working)
- Announce generation and forwarding with Ed25519 signature verification

Acceptance: Two nodes on different transports (e.g., LoRa + WiFi) can establish an encrypted link, exchange announces, and forward packets to each other. Packet contents are encrypted and unauthenticated nodes are rejected.

Milestone 1.2: Routing + Gossip

- `CompactPathCost` (6-byte encoding/decoding, log-scale math, relay update logic)
- Routing table (`RoutingEntry` with cost, latency, bandwidth, hop count, reliability)

- Greedy forwarding with `PathPolicy` scoring (Cheapest, Fastest, Balanced)
- Gossip protocol (60-second rounds, bloom filter state summaries, delta exchange)
- Bandwidth budget enforcement (4-tier allocation, constrained-link adaptation)
- Announce propagation rules (event-driven + 30-min refresh, hop limit, expiry, link failure detection)

Acceptance: A 5-node mesh (mixed LoRa + WiFi) converges routing tables within 3 gossip rounds. Packets are forwarded via cost-optimal paths. Removing a node causes re-routing within 3 minutes. Gossip overhead stays within 10% budget on all links.

Milestone 1.3: Congestion Control

- CSMA/CA on half-duplex links (listen-before-talk with exponential backoff)
- Per-neighbor token bucket (fair share, payment-weighted priority)
- 4-level priority queuing (P0 real-time through P3 bulk, starvation prevention)
- Backpressure signaling (4-byte CongestionSignal)
- Dynamic cost response (quadratic formula, propagated via gossip)

Acceptance: Under synthetic load, P0 packets (voice) maintain latency below 500ms while P3 (bulk) is throttled. Congestion on one link causes traffic to reroute via cost increase. No link exceeds its bandwidth budget.

Milestone 1.4: Payment Channels

- VRF lottery implementation (ECVRF-ED25519-SHA512-TAI per RFC 9381)
- Adaptive difficulty (local per-link, formula: `win_prob = target_updates / observed_packets`)
- `ChannelState` (200 bytes, dual-signed, sequence-numbered)
- Channel lifecycle (open, update on win, settle, dispute with 2,880-round window, abandon after 4 epochs)
- `SettlementRecord` generation and dual-signature

Acceptance: Two nodes relay 1,000 packets. The relay wins the VRF lottery approximately `1000 × win_probability` times (within 2σ). Channel updates occur only on wins. Settlement produces a valid dual-signed record. Dispute resolution correctly rejects old states.

Milestone 1.5: CRDT Ledger

- `AccountState` (GCounter for earned/spent, GSet for settlements)
- GCounter merge (pointwise max per-node entries)
- Settlement flow (validation: 2 sig checks + Blake3 hash + GSet dedup, gossip forward)
- Balance derivation (`earned - spent`, reject negative)
- Gossip integration (settlements propagate via Tier 2 bandwidth budget)

Acceptance: Three nodes settle channels pairwise. All balances converge correctly across the mesh within O(log N) gossip rounds. Duplicate settlements are rejected. A forged settlement (bad signature) is silently dropped by all nodes.

```
<a id="development-roadmap--milestone-16-hardware-targets"></a>
```

Milestone 1.6: Hardware Targets

- ESP32 + LoRa firmware (relay only: transport, routing, gossip, payment channels, CRDT ledger)
- Raspberry Pi software (bridge: all ESP32 capabilities + multi-interface bridging + basic CLI)
- CLI tools: `mehr-keygen`, `mehr-node` (start/stop), `mehr-status` (routing table, channels, balances), `mehr-peer` (add/remove trusted peers)

Acceptance: An ESP32 node relays packets and earns VRF lottery wins. A Raspberry Pi bridges LoRa to WiFi. CLI tools show correct routing table, channel states, and balances. ESP32 firmware fits in flash and runs within 520 KB RAM.

```
<a id="development-roadmap--phase-1-deliverable"></a>
```

Phase 1 Deliverable

A working mesh network with cost-aware routing, encrypted links, stochastic micropayments, and a convergent CRDT ledger — running on real hardware (ESP32 + Raspberry Pi) with CLI management tools.

```
<a id="development-roadmap--phase-2-economics"></a>
```

Phase 2: Economics

Focus: Real-world deployment, economic mechanisms, and the minimum viable marketplace.

```
<a id="development-roadmap--milestone-21-trust-neighborhoods"></a>
```

Milestone 2.1: Trust Neighborhoods

- `TrustConfig` implementation (trusted peers, cost overrides, community labels)
- Free relay logic (sender trusted AND destination trusted → no lottery)
- Transitive credit (direct: full, friend-of-friend: 10%, 3+ hops: none)
- Credit rate limiting per trust distance

Acceptance: Trusted peers relay traffic for free with zero economic overhead. A friend-of-friend gets exactly 10% of the direct credit line. An untrusted node gets no transitive credit.

```
<a id="development-roadmap--milestone-22-epoch-compaction"></a>
```

Milestone 2.2: Epoch Compaction

- Epoch trigger logic (3-trigger: settlement count, GSet size, small-partition adaptive)
- Epoch proposer selection (eligibility rules, conflict resolution)
- Bloom filter construction (k=13 Blake3-derived, 0.01% FPR)
- Epoch lifecycle (Propose → Acknowledge at 67% → Activate → Verify → Finalize)
- Settlement proof submission during grace period
- Merkle-tree snapshot (full on backbone, sparse on constrained)

- `BalanceProof` generation and verification (~640 bytes for 1M nodes)
- `RelayWinSummary` aggregation and spot-check verification

Acceptance: A 20-node test network triggers epochs correctly under all three conditions. Bloom filter FPR is within 2× of theoretical 0.01%. Late-arriving settlements are recovered during the grace period. ESP32 nodes operate with sparse snapshots under 5 KB.

Milestone 2.3: Capability Discovery

- `NodeCapabilities` advertisement structure (connectivity, compute, storage, availability)
- `PresenceBeacon` (20 bytes, broadcast every 10 seconds)
- Discovery rings (Ring 0 full, Ring 1 summarized, Ring 2 periodic, Ring 3 query-only)
- Mobile handoff (beacon scan → relay selection → link establishment → channel open)
- Credit-based fast start (`CreditGrant` with staleness tolerance and rate limiting)
- Roaming cache (area fingerprint with 60% overlap tolerance, 30-day TTL)

Acceptance: A mobile node (laptop) moves between two WiFi areas and hands off to a new relay within 500ms. Credit-based fast start allows immediate traffic while the channel opens. Returning to the first area reuses the cached channel with zero handoff latency.

Milestone 2.4: Reputation

- `PeerReputation` scoring (relay, storage, compute scores 0–10000)
- Score update formulas (success: diminishing gains, failure: 10% penalty)
- Trust-weighted referrals (1-hop, capped at 50%, weighted at 30% × trust, 500-round expiry)
- Reputation integration with credit line sizing and capability selection

Acceptance: A node that successfully relays 100 packets has a relay score above 5000. A node that fails 3 out of 10 agreements has a score below 3000. Referral scores are capped at 5000 and overwritten by first-hand experience.

Milestone 2.5: Test Networks

- Deploy 3–5 physical test networks (urban, rural, indoor, mixed)
- Each network: 10–50 nodes across at least 2 transport types
- Instrument for: routing convergence time, payment overhead, epoch timing, gossip bandwidth
- Run for at least 4 weeks per network
- Document: failure modes, parameter tuning, real-world performance vs. spec predictions

Acceptance: Test networks operate continuously for 4 weeks. Routing converges within spec. Payment overhead stays within 0.5% on LoRa. At least one epoch completes successfully per network. Published test report with metrics.


```
### Phase 2 Deliverable

Test networks with functioning trust-based economics, epoch compaction, capability
discovery, mobile handoff, and reputation scoring — validated on real hardware over
multiple weeks.

---

<a id="development-roadmap--phase-3-services"></a>

## Phase 3: Services

**Focus**: Service primitives and first applications.

<a id="development-roadmap--milestone-31-mhr-store"></a>

### Milestone 3.1: MHR-Store

- `DataObject` types (Immutable, Mutable, Ephemeral)
- Storage agreements (bilateral, pay-per-duration)
- Proof of storage (Blake3 Merkle challenge-response)
- Erasure coding (Reed-Solomon, default schemes by size)
- Repair protocol (detect failure → assess → reconstruct → re-store)
- Garbage collection (7-tier priority)

<a id="development-roadmap--milestone-32-mhr-dht"></a>

### Milestone 3.2: MHR-DHT

- DHT routing (XOR distance + cost weighting, α=0.7)
- k=3 replication with cost-bounded storage set
- Lookup and publication protocols
- Cache management (publisher TTL, 24-hour local cap, LRU eviction)
- Light client verification (3-tier: content-hash, signature, multi-source)

<a id="development-roadmap--milestone-33-mhr-pub"></a>

### Milestone 3.3: MHR-Pub

- Subscription types (Key, Prefix, Node, Neighborhood)
- Delivery modes (Push, Digest, PullHint)
- Bandwidth-adaptive mode selection

<a id="development-roadmap--milestone-34-mhr-compute-mhr-byte"></a>

### Milestone 3.4: MHR-Compute (MHR-Byte)

- 47-opcode interpreter implementation in Rust
- Cycle cost enforcement (ESP32-calibrated)
- Resource limit enforcement (max_memory, max_cycles, max_state_size)
- Compute delegation via capability marketplace
- Reference test vector suite for cross-platform conformance

<a id="development-roadmap--milestone-35-applications"></a>
```

```
### Milestone 3.5: Applications

- Messaging (E2E encrypted, store-and-forward, group messaging with co-admin delegation)
- Social (profiles, feeds, followers, media tiering)
- MHR-Name (community-label-scoped naming, conflict resolution, petnames)

<a id="development-roadmap--phase-3-deliverable"></a>

### Phase 3 Deliverable

Usable messaging and social applications running on the mesh, with decentralized storage,
DHT-based content discovery, and contract execution on constrained devices.

---

<a id="development-roadmap--phase-4-ecosystem"></a>

## Phase 4: Ecosystem

**Focus**: Advanced capabilities and ecosystem growth.

<a id="development-roadmap--milestone-41-advanced-compute"></a>

### Milestone 4.1: Advanced Compute

- WASM execution environment for gateway/backbone nodes
- Private compute tiers (Split Inference, Secret Sharing, TEE)
- Heavy compute capabilities (ML inference, transcription, TTS)

<a id="development-roadmap--milestone-42-rich-applications"></a>

### Milestone 4.2: Rich Applications

- Voice (Codec2 on LoRa, Opus on WiFi, bandwidth bridging)
- Forums (append-only logs, moderation contracts)
- Marketplace (listings, escrow contracts)
- Wiki (CRDT-merged collaborative documents)

<a id="development-roadmap--milestone-43-interoperability"></a>

### Milestone 4.3: Interoperability

- Third-party protocol bridges (SSB, Matrix, Briar) - [standalone gateway services]
(#development-design-decisions--protocol-bridges-standalone-gateway-services) with identity
attestation
- Onion routing implementation (`PathPolicy.ONION_ROUTE`, per-packet layered encryption)

<a id="development-roadmap--milestone-44-ecosystem-growth"></a>

### Milestone 4.4: Ecosystem Growth

- Hardware partnerships and reference design refinement
- Developer SDK and documentation
```

- Community-driven capability development

Phase 4 Deliverable

A full-featured decentralized platform with rich applications, privacy-enhanced routing, and interoperability with existing protocols.

Implementation Language

The primary implementation language is **Rust**, chosen for:

- Memory safety without garbage collection (critical for embedded targets)
- `no_std` support for ESP32 firmware
- Strong ecosystem for cryptography and networking
- Single codebase from microcontroller to server

Test Network Strategy

Real physical test networks, not simulations:

- Simulation cannot capture the realities of LoRa propagation, WiFi interference, and real-world device failure modes
- Each test network should represent a different deployment scenario (urban, rural, indoor, mixed)
- Test networks validate both the protocol and the economic model

Phase 1 Implementability Assessment

Phase 1 is **fully implementable** with the current specification. All protocol-level gaps have been resolved:

Component	Spec Status	Key References
Identity + Encryption	Complete	[Security](#protocol-security)
Packet format + CompactPathCost	Complete (wire format specified)	[Network Protocol](#protocol-network-protocol--mehr-extension-compact-path-cost)
Routing + Announce propagation	Complete (scoring, announce rules, expiry, failure detection)	[Network Protocol](#protocol-network-protocol--routing)
Gossip protocol	Complete (bloom filter, bandwidth budget, 4-tier)	[Network Protocol](#protocol-network-protocol--gossip-protocol)
Congestion control	Complete (3-layer, priority levels, backpressure)	[Network Protocol](#protocol-network-protocol--congestion-control)
VRF lottery + Payment channels	Complete (RFC 9381, difficulty formula, channel lifecycle)	[Payment Channels](#economics-payment-channels)

| CRDT ledger + Settlement | Complete (validation rules, GCounter merge, GSet dedup) |
[CRDT Ledger](#economics-crdt-ledger) |
| Hardware targets | Complete (ESP32 + Pi reference designs) | [Reference Designs]
(#hardware-reference-designs) |

<div class="page-break"></div>

Design Decisions Log

This page documents the key architectural decisions made during Mehr protocol design,
including alternatives considered and the rationale for each choice.

Network Stack: Reticulum as Initial Transport

| | |
├───┼───┤
| **Chosen** | Use [Reticulum Network Stack](https://reticulum.network/) as the initial
transport implementation; treat it as a [swappable layer](#development-design-decisions--
transport-layer-swappable-implementation) |
| **Alternatives** | Clean-room implementation from day one, libp2p, custom protocol |
| **Rationale** | Reticulum already solves transport abstraction, cryptographic identity,
mandatory encryption, sender anonymity (no source address), and announce-based routing —
all proven on LoRa at 5 bps. Mehr extends it with CompactPathCost annotations and economic
primitives rather than reinventing a tested foundation. The transport layer is an
implementation detail: Mehr defines the interface it needs and can switch to a clean-room
implementation in the future without affecting any layer above. |

Routing: Kleinberg Small-World with Cost Weighting

| | |
├───┼───┤
| **Chosen** | Greedy forwarding on a Kleinberg small-world graph with cost-weighted
scoring |
| **Alternatives** | Pure Reticulum announce model, Kademlia, BGP-style routing, Freenet-
style location swapping |
| **Rationale** | The physical mesh naturally forms a small-world graph: short-range radio
links serve as lattice edges, backbone/gateway links serve as Kleinberg long-range
contacts. Greedy forwarding achieves $O(\log^2 N)$ expected hops — a formal scalability
guarantee. Cost weighting trades path length for economic efficiency. Unlike Freenet, no
location swapping is needed because destination hashes are self-assigned and Reticulum
announcements build the navigable topology. |

Payment: Stochastic Relay Rewards

———	———
Chosen	Probabilistic micropayments via VRF-based lottery (channel update only on wins)
Alternatives	Per-packet accounting, per-minute batched accounting, subscription-based, random-nonce lottery
Rationale	Per-packet and batched payment require frequent channel state updates, consuming ~2–4% bandwidth on LoRa links. Stochastic rewards achieve the same expected income but trigger updates only on lottery wins — reducing economic overhead by ~10x. Adaptive difficulty ensures fairness across traffic levels. The law of large numbers guarantees convergence for active relays. **The lottery uses a VRF (ECVRF-ED25519-SHA512-TAI, RFC 9381)** rather than a random nonce to prevent relay nodes from grinding nonces to win every packet. The VRF produces exactly one verifiable output per (relay key, packet) pair, reusing the existing Ed25519 keypair.

Settlement: CRDT Ledger

———	———
Chosen	CRDT ledger (GCounters + GSet)
Alternatives	Blockchain, federated sidechain
Rationale	Partition tolerance is non-negotiable. CRDTs converge without consensus. A blockchain requires global ordering, which is impossible when network partitions are expected operating conditions. **Tradeoff**: double-spend prevention is probabilistic, not perfect. Mitigated by channel deposits, credit limits, reputation staking, and blacklisting — making cheating economically irrational for micropayments.

Communities: Emergent Trust Neighborhoods

———	———
Chosen	Trust graph with emergent neighborhoods (no explicit zones)
Alternatives	Explicit zones with admin keys and admission policies (v0.8 design)
Rationale	Explicit zones require someone to create and manage them — centralized thinking in decentralized clothing. They impose UX burden and artificially fragment communities. Trust neighborhoods emerge naturally from who you trust: free communication between trusted peers, paid between strangers. No admin, no governance, no admission policies. Communities form the same way they form in real life — through relationships, not administrative acts. The trust graph provides Sybil resistance economically (vouching peers absorb debts).

Compaction: Epoch Checkpoints with Bloom Filters

———	———
Chosen	Epoch checkpoints with bloom filters

| **Alternatives** | Per-settlement garbage collection, TTL-based expiry |
| **Rationale** | The settlement GSet grows without bound. Bloom filters at 0.01% FPR compress 1M settlement hashes from ~32 MB to ~2.4 MB. A settlement verification window during the grace period recovers any settlements lost to false positives. Epochs are triggered by settlement count (~10,000 batches), not wall-clock time, for partition tolerance. |

Compute Contracts: MHR-Byte

—	—
Chosen	MHR-Byte (minimal bytecode, ~50 KB interpreter)
Alternatives	Full WASM everywhere
Rationale	ESP32 microcontrollers can't run a WASM runtime. MHR-Byte provides basic contract execution on even the most constrained devices. WASM is offered as an optional capability on nodes with sufficient resources.

Encryption: Ed25519 + X25519

—	—
Chosen	Ed25519 for identity/signing, X25519 for key exchange (Reticulum-compatible)
Alternatives	RSA, symmetric-only
Rationale	Ed25519 has 32-byte public keys (compact for radio), fast signing/verification, and is widely proven. X25519 provides efficient Diffie-Hellman key exchange. Compatible with Reticulum's crypto model. RSA keys are too large for constrained links.

Source Privacy: No Source Address (Default)

—	—
Chosen	No source address in packet headers (inherited from Reticulum) as the default; [opt-in onion routing](#development-design-decisions--onion-routing-per-packet-layered-encryption-opt-in) for high-threat environments
Alternatives	Mandatory onion routing for all traffic
Rationale	Onion routing adds 21% payload overhead on LoRa — unacceptable as a default for all traffic. Omitting the source address is free and effective against casual observation. Per-packet layered encryption is available opt-in via `PathPolicy.ONION_ROUTE` for users who need stronger traffic analysis resistance.

Naming: Neighborhood-Scoped, No Global Namespace

| | |
|—|—|

Chosen	Community-label-scoped names (e.g., `alice@portland-mesh`)
Alternatives	Global names via consensus
Rationale	Global consensus contradicts partition tolerance. Community labels are self-assigned and informational — no authority, no uniqueness enforcement. Multiple disjoint clusters can share a label; resolution is proximity-based. Local petnames provide a fallback.

Cost Annotations: Compact Path Cost (No Per-Relay Signatures)

——	——
Chosen	6-byte constant-size `CompactPathCost` (running totals updated by each relay, no per-relay signatures)
Alternatives	Per-relay signed CostAnnotation (~84 bytes per hop), aggregate signatures, signature-free with Merkle proof
Rationale	Per-relay signatures make announces grow linearly with path length — 84 bytes × N hops. On a 1 kbps LoRa link with 3% routing budget, this limits convergence to ~1 announce per 22+ seconds. CompactPathCost uses 6 bytes total regardless of path length: log-encoded cumulative cost, worst-case latency, bottleneck bandwidth, and hop count. Per-relay signatures are unnecessary because routing decisions are local (you trust your link-authenticated neighbor), trust is transitive at each hop, and the market enforces honesty (overpriced relays get routed around, underpriced relays lose money). The announce itself remains signed by the destination node.

Congestion Control: Three-Layer Design

——	——
Chosen	Link-level CSMA/CA + per-neighbor token bucket + 4-level priority queuing + economic cost response
Alternatives	Pure CSMA/CA only, rigid TDMA, end-to-end TCP-style congestion control
Rationale	A single mechanism is insufficient across the bandwidth range (500 bps to 10 Gbps). CSMA/CA handles collision avoidance on half-duplex radio. Token buckets enforce fair sharing across neighbors. Priority queuing ensures real-time traffic (voice) isn't starved by bulk transfers. Economic cost response (quadratic cost increase under congestion) signals scarcity through the existing cost routing mechanism, causing natural traffic rerouting without new protocol extensions. End-to-end congestion control (TCP-style) is wrong for a mesh — the bottleneck is typically a single constrained link, and hop-by-hop control responds faster. Rigid TDMA wastes bandwidth when some slots are unused.

Transport Layer: Swappable Implementation

| | |
|——|——|
| **Chosen** | Define transport as an interface with requirements; use Reticulum as the

current implementation |
| **Alternatives** | Hard dependency on Reticulum, clean-room from day one |
| **Rationale** | Reticulum provides a proven transport layer tested at 5 bps on LoRa, saving significant implementation effort. But coupling Mehr to Reticulum's codebase or community roadmap creates fragility. Instead, Mehr defines the transport interface it needs (transport-agnostic links, announce-based routing, mandatory encryption, sender anonymity) and uses Reticulum as the current implementation. Mehr extensions are carried as opaque payload in the announce DATA field — a clean separation. Three participation levels (L0 transport-only, L1 Mehr relay, L2 full Mehr) ensure interoperability with the underlying transport. This allows a future clean-room implementation without affecting any layer above transport. |

Storage: Pay-Per-Duration with Erasure Coding

—	—
Chosen	Bilateral storage agreements, pay-per-duration, Reed-Solomon erasure coding, lightweight challenge-response proofs
Alternatives	Filecoin-style PoRep/PoSt with on-chain proofs; Arweave-style one-time-payment permanent storage; simple full replication
Rationale	Filecoin's Proof of Replication requires GPU-level computation (minutes to seal a sector) — impossible on ESP32 or Raspberry Pi. Arweave's permanent storage requires a blockchain endowment model and assumes perpetually declining storage costs. Both require global consensus. Mehr uses simple Blake3 challenge-response proofs (verifiable in under 10ms on ESP32) and bilateral agreements settled via payment channels. Erasure coding (Reed-Solomon) provides the same durability as 3x replication at 1.5x storage overhead. The tradeoff: we can't prove a node stores data *uniquely* (no PoRep), but we can prove it stores data *at all* — and the data owner doesn't care how the node organizes its disk.

Mobile Handoff: Presence Beacons + Credit-Based Fast Start

—	—
Chosen	Transport-agnostic presence beacons, credit-based fast start, roaming cache with channel preservation
Alternatives	Pre-negotiated handoff (cellular-style), pure re-discovery from scratch, always-connected overlay
Rationale	Cellular handoff requires a central controller — incompatible with decentralized mesh. Pure re-discovery works but is slow for latency-sensitive sessions. Presence beacons (20 bytes, broadcast every 10 seconds on any interface) let mobile nodes passively discover local relays before connecting. Credit-based fast start allows immediate relay if the mobile node has visible CRDT balance, while the payment channel opens in the background. The roaming cache preserves channels for previously visited areas, enabling zero-latency reconnection on return. No explicit teardown needed — old agreements expire via `valid_until`.

## Light Client Trust: Content Verification + Multi-Source Queries

———	———
Chosen	Three-tier verification: content-hash check (Tier 1), owner-signature check (Tier 2), multi-source queries (Tier 3)
Alternatives	Merkle proofs over DHT state, SPV-style state root verification, full DHT replication on clients
Rationale	Merkle proofs require a global state root, which contradicts partition tolerance (no consensus). SPV-style verification has the same problem. Full DHT replication is too bandwidth-heavy for phones. Instead, content addressing (Tier 1) gives zero-overhead verification for the most common case — `Blake3(data) == key` proves authenticity regardless of relay honesty. Signed objects (Tier 2) prevent forgery of mutable data. Multi-source queries (Tier 3, N=2-3 independent nodes) detect censorship and staleness. Trusted relays skip to single-source for all tiers. Overhead is minimal: at most one extra 192-byte query for critical lookups via untrusted relays.


Epoch Triggers: Adaptive Multi-Criteria
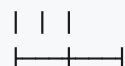
———	———
Chosen	Three-trigger system: settlement count ≥ 10,000 (large mesh), GSet size ≥ 500 KB (memory pressure), or partition-proportional threshold with gossip-round floor (small partitions)
Alternatives	Fixed 10,000-settlement-only trigger, wall-clock timer, per-node independent compaction
Rationale	A 20-node village on LoRa with low traffic might take months to reach 10,000 settlements. ESP32 nodes (520 KB usable RAM) would exhaust memory first. The adaptive trigger fires at max(200, active_set × 10) settlements with a 1,000-gossip-round floor (~17 hours), keeping GSet under ~6.4 KB for small partitions. The 500 KB GSet size trigger is a safety net regardless of partition size. Proposer eligibility adapts too — only 3 direct links needed (not 10) in small partitions. Wall-clock timers were rejected because they require clock synchronization. Per-node compaction was rejected because it fragments global state.

Ledger Scaling: Merkle-Tree Snapshots with Sparse Views
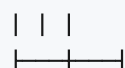
———	———
Chosen	Merkle-tree account snapshot; full tree on backbone nodes, sparse view + Merkle root on constrained devices, on-demand balance proofs (~640 bytes)
Alternatives	Flat snapshot everywhere, sharded epochs, neighborhood-only ledgers
Rationale	At 1M nodes the flat snapshot is ~32 MB — unworkable on ESP32 or phones. Sharded epochs fragment the ledger and complicate cross-shard verification. Neighborhood-only ledgers lose global balance consistency. A Merkle tree over the sorted account snapshot gives the best of both: backbone nodes store the full tree (32 MB, feasible on SSD), constrained devices store only their own balance + channel partners + trust neighbors (~1.6 KB for 50 accounts) plus the Merkle root. Any balance can be verified on demand with a ~640-byte proof (20 tree levels × 32 bytes). No global state transfer needed.

Transforms/Inference: Compute Capability, Not Protocol Primitive

| | |
├──┼──┤
Chosen	STT/TTS/inference as compute capabilities in the marketplace
Alternatives	Dedicated transform layer (considered in v0.5 draft)
Rationale	Speech-to-text, translation, and other transforms are just compute. Making them protocol primitives over-engineers the foundation. The capability marketplace already handles discovery, negotiation, execution, verification, and payment for arbitrary compute functions.

Group Admin: Delegated Co-Admin (No Threshold Signatures)

| | |
├──┼──┤
Chosen	Delegated co-admin model: group creator signs delegation certificates for up to 3 co-admins; any co-admin can independently rotate keys and manage members
Alternatives	Threshold signatures (e.g., 2-of-3 Schnorr multisig), single admin only, leaderless consensus
Rationale	Threshold signatures (Schnorr multisig, FROST) require multi-round key generation and signing protocols that are too expensive for ESP32 and too complex for LoRa latency. Leaderless consensus contradicts the "no global coordination" principle. Instead, the group creator signs `CoAdminCertificate` records (admin public key + permissions + sequence number, ~128 bytes each) that authorize up to 3 co-admins. Any co-admin can independently add/remove members, rotate the group symmetric key, and promote/demote other co-admins (if authorized). Operations are sequence-numbered; conflicts are resolved by highest sequence number, ties broken by lowest admin public key hash. Overhead is one Ed25519 signature per admin action — no new cryptographic primitives needed. If all admins go offline, the group continues functioning with its current key; no key rotation or membership changes occur until at least one admin returns.

Reputation: Bounded Trust-Weighted Referrals

| | |
├──┼──┤
| **Chosen** | First-hand scores primary; 1-hop trust-weighted referrals as advisory bootstrap for unknown peers, capped and decaying |
| **Alternatives** | Pure first-hand only (no referrals), full gossip-based reputation, global reputation aggregation |
| **Rationale** | Pure first-hand scoring means new nodes have zero information about any peer until direct interaction — leading to poor initial routing and credit decisions. Full reputation gossip enables manipulation: a colluding cluster can flood the network with inflated scores. Global aggregation requires consensus. The chosen design: trusted peers (direct trust edges) can share their first-hand scores as referrals. Referrals are weighted by the querier's trust in the referrer (trust_score / max_score × 0.3) and capped at 50% of

max reputation — a referral alone cannot make a peer fully trusted. Only 1-hop referrals are accepted (no transitive gossip), limiting the manipulation surface to corruption of direct trusted peers, which already breaks the trust model. Referral scores are advisory: they initialize a peer's reputation but are overwritten by first-hand experience after the first few direct interactions. Referrals expire after 500 gossip rounds (~8 hours) without refresh. This helps new nodes bootstrap while keeping the attack surface minimal. |

Onion Routing: Per-Packet Layered Encryption (Opt-In)

——	——
Chosen	Per-packet layered encryption via `PathPolicy.ONION_ROUTE`, stateless relays, 3 hops default, optional cover traffic
Alternatives	Circuit-based onion routing (Tor-style), mix networks, no onion routing (status quo)
Rationale	Circuit-based onion routing requires multi-round-trip circuit establishment — on a 1 kbps LoRa link with 2-second round trips, building a 3-hop circuit takes ~12 seconds before any data flows. Mix networks add latency by design (batching and reordering), which is unacceptable for interactive messaging. Per-packet layered encryption has zero setup: the sender wraps the message in N encryption layers (default N=3), each layer containing the next-hop destination hash and the inner ciphertext. Each relay decrypts one layer, reads the next hop, and forwards. No circuit state on relays — each packet is independently routable. Overhead: 32 bytes per hop (16-byte nonce + 16-byte Poly1305 tag), so 3 hops = 96 bytes, leaving 369 of 465 usable bytes on LoRa (~21% overhead). Relay selection: sender picks from known relays, requiring at least one relay outside the sender's trust neighborhood. Optional constant-rate cover traffic (1 dummy packet/minute, off by default) provides additional resistance to timing analysis on high-threat links. Onion routing is opt-in and not recommended for real-time voice (latency and payload overhead). The existing no-source-address design remains the default privacy layer for most traffic.

MHR-Byte: 47 Opcodes with Reference Interpreter

——	——
Chosen	47 opcodes in 7 categories, reference interpreter in Rust, cycle costs calibrated to ESP32
Alternatives	Formal specification (Yellow Paper-style), minimal 20-opcode set, EVM-compatible opcode set
Rationale	A formal specification (Ethereum Yellow Paper style) would freeze the design prematurely — the opcode set needs real-world usage feedback before committing to a formal spec. An EVM-compatible set imports unnecessary complexity (256-bit words, gas semantics) that doesn't fit constrained devices. A minimal 20-opcode set omits crypto and system operations needed for the core use cases. The chosen 47 opcodes cover 7 categories: **Stack** (6): PUSH, POP, DUP, SWAP, OVER, ROT. **Arithmetic** (9): ADD, SUB, MUL, DIV, MOD, NEG, ABS, MIN, MAX — all 64-bit integer, overflow traps. **Bitwise** (6): AND, OR, XOR, NOT, SHL, SHR. **Comparison** (6): EQ, NEQ, LT, GT, LTE, GTE. **Control** (7): JMP, JZ, JNZ, CALL, RET, HALT, ABORT. **Crypto** (3): HASH (Blake3), VERIFY_SIG (Ed25519), VERIFY_VRF. **System** (10): BALANCE, SENDER, SELF, EPOCH, TRANSFER, LOG, LOAD, STORE,

MSIZE, EMIT. Cycle costs are tiered: stack/arithmetic/bitwise/comparison (1-3 cycles), control (2-5), memory (2-3), crypto (500-2000), system (10-50). ESP32 is the reference platform (~1 μs per base cycle). Faster hardware executes more cycles per second but charges the same cycle cost — gas price in μMHR/cycle is set by each compute provider. A comprehensive test vector suite ensures cross-platform conformance. Formal specification is a post-stabilization goal. |

Emission Schedule: Epoch-Counted Discrete Halving

——	——
Chosen	Initial reward of 10^12 μMHR per epoch, discrete halving every 100,000 epochs, tail emission floor at 0.1% of circulating supply annualized
Alternatives	Continuous exponential decay, wall-clock halving (every 2 calendar years), fixed perpetual emission
Rationale	Wall-clock halving requires clock synchronization, which contradicts partition tolerance — partitioned nodes would disagree on the current halving period. Continuous decay is mathematically elegant but harder to reason about and implement correctly on integer-only constrained devices. Fixed perpetual emission provides no scarcity signal. Discrete halving every 100,000 epochs is epoch-counted (partition-safe), easy to compute (bit-shift), and predictable. At an estimated ~1 epoch per 10 minutes (varies with network activity since epochs are settlement-triggered), 100,000 epochs ≈ 1.9 years — close to the original 2-year target. Initial reward of 10^12 μMHR/epoch yields ~1.5% of the supply ceiling minted in the first halving period, providing strong bootstrap incentive. Tail emission activates when the halved reward drops below `0.001 × circulating_supply / estimated_epochs_per_year` (trailing 1,000-epoch moving average of epoch frequency). This ensures perpetual relay incentive while keeping inflation negligible. Partition minting interaction: each partition mints `(local_active_relays / estimated_global_relays) × epoch_reward`; on merge, GCounter max-merge preserves individual balances, and overminting is bounded by the existing partition_scale_factor tolerance (max 1.5×).

Protocol Bridges: Standalone Gateway Services

——	——
Chosen	Bridges as standalone gateway services advertising in the capability marketplace; one-way identity attestation; bridge operator bears Mehr-side costs
Alternatives	Bridges as MHR-Compute contracts, bridge as protocol-level primitive, no bridge design (defer entirely)
Rationale	MHR-Compute contracts are sandboxed with no I/O or network access — bridges require persistent connections to external protocols (SSB replication, Matrix homeserver federation, Briar Tor transport). A protocol-level primitive over-engineers the foundation for what is fundamentally a gateway service. Bridges run as standalone processes that advertise their bridging capability in the marketplace like any other service. **Identity mapping**: Users create signed bridge attestations linking their Mehr Ed25519 key to their external identity. The bridge stores these attestations and handles translation. No global identity registry — the bridge is the only entity that knows the mapping. External users see messages as coming from the bridge identity in the external protocol. **Payment model**: Mehr-to-external traffic — the Mehr sender pays the bridge

operator via a standard marketplace agreement. External-to-Mehr traffic — the bridge
operator pays Mehr relay costs and recoups via the external protocol's economics (or
operates as a public good). This keeps the Mehr economic model clean: the bridge is just
another service consumer/provider. Bridge operators can support multiple protocols
simultaneously and set their own pricing. |

WASM Sandbox: Wasmtime with Tiered Profiles

| | |
|——|——|
| **Chosen** | Wasmtime (Bytecode Alliance) as WASM runtime; two tiers: Light (Community,
16 MB / 10^8 fuel / 5s) and Full (Gateway+, 256 MB / 10^10 fuel / 30s); 10 host imports
mirroring MHR-Byte System opcodes |
| **Alternatives** | Wasmer, custom interpreter, single WASM profile for all devices |
| **Rationale** | Wasmtime is Rust-native (matches the reference implementation language),
provides fuel-based metering that maps directly to MHR-Byte cycle accounting, and supports
AOT compilation on Gateway+ nodes. Wasmer has a broader language ecosystem but less tight
Rust integration. A custom interpreter would duplicate Wasmtime's battle-tested sandboxing.
A single WASM profile either excludes Community-tier devices (too high limits) or handicaps
Gateway nodes (too low limits). Two tiers match the natural hardware split: Pi Zero 2W (512
MB RAM, interpreted Cranelift) vs. Pi 4/5+ (4-8 GB, AOT). Host imports are restricted to 10
functions mirroring MHR-Byte System opcodes — no filesystem, network, clock, or RNG access.
WASM execution remains pure and deterministic, enabling the same verification methods as
MHR-Byte. Contracts exceeding Light limits are automatically delegated to a more capable
node via compute delegation. |

Presence Beacon: 8-Bit Capability Bitfield

| | |
|——|——|
| **Chosen** | 8 assigned capability bits in 16-bit field; bits 8-15 reserved for future
use |
| **Alternatives** | Variable-length capability list, TLV-encoded capabilities, separate
beacon per capability |
| **Rationale** | The beacon must fit in 20 bytes total and broadcast every 10 seconds on
LoRa — every byte matters. A 16-bit bitfield encodes up to 16 boolean capabilities in 2
bytes, zero parsing overhead. The 8 assigned bits cover all current service types: relay
(L1+), gateway, storage, compute-byte, compute-wasm, pubsub, DHT, and naming. Bits 8-15 are
reserved (must be zero) for future services like inference, bridge, etc. Variable-length
lists or TLV encoding would bloat the beacon and complicate parsing on ESP32. Separate
beacons per capability would multiply broadcast bandwidth. |

Ring 1 Discovery: CapabilitySummary Format

| | |
|——|——|
| **Chosen** | 8-byte `CapabilitySummary` per capability type: type (u8), count (u8),
min/avg cost (u16 each, $\log_2$-encoded), min/max hops (u8 each) |

| **Alternatives** | Full capability advertisements forwarded, Bloom filter of providers, free-text summaries |
| **Rationale** | Ring 1 gossips summaries every few minutes — bandwidth must stay under ~50 bytes per round. At 8 bytes per type and typically 5-6 types present in a 2-3 hop neighborhood, the total is 40-48 bytes — within budget even on LoRa. Forwarding full capability advertisements would scale linearly with provider count. Bloom filters compress provider identity but lose cost/distance information needed for routing decisions. The log_2-encoded cost fields match `CompactPathCost` encoding, keeping the representation consistent across the protocol. Count is capped at 255 (u8) — sufficient since Ring 1 covers only 2-3 hops. |

DHT Metadata: 129-Byte Signed Entries

—	—
Chosen	129-byte `DHTMetadata`: 32-byte Blake3 key, u32 size, u8 content_type (Immutable/Mutable/Ephemeral), 16-byte owner, u32 TTL, u64 Lamport timestamp, 64-byte Ed25519 signature
Alternatives	Minimal key-only gossip (no metadata), full object gossip, variable-length metadata with optional fields
Rationale	DHT publication gossips metadata to let storage-set nodes decide whether to pull full data. Key-only gossip forces blind pulls — wasting bandwidth on unwanted or expired data. Full object gossip floods the gossip channel with arbitrarily large payloads. The 129-byte fixed format includes everything a storage node needs to decide: content hash (for deduplication), size (for storage budgeting), content type (for cache policy), owner (for mutable-object freshness ordering), TTL (for garbage collection), and Lamport timestamp (for mutable conflict resolution). The Ed25519 signature prevents metadata forgery — a node cannot falsely advertise objects it doesn't own. Content hash prevents data forgery on pull. For mutable objects, highest Lamport timestamp with valid signature wins; for immutable objects, the content hash is the sole arbiter. Cache invalidation is TTL-based with no push-invalidation — keeping the protocol simple and partition-tolerant.

Negotiation Protocol: Single-Round Take-It-or-Leave-It

—	—
Chosen	Single round-trip negotiation: consumer sends `CapabilityRequest` (with desired cost, duration, proof preference, nonce); provider responds with `CapabilityOffer` (actual terms) or reject; consumer accepts or walks away. 30-second timeout. No counter-offers.
Alternatives	Multi-round bidding/auction, Dutch auction, sealed-bid auction, negotiation-free fixed pricing
Rationale	Multi-round negotiation is untenable on LoRa where each message takes seconds — a 3-round negotiation would take 30+ seconds before service begins. Auctions require multiple participants to discover each other simultaneously, which contradicts the bilateral, privacy-preserving nature of agreements. Fixed pricing (no negotiation) removes the consumer's ability to express budget constraints. The single-round protocol: the consumer states their maximum acceptable cost; the provider either meets it, undercuts it, or rejects. One round-trip, then service begins. The nonce in `CapabilityRequest` prevents

replay attacks; the `request_nonce` echo in `CapabilityOffer` binds offer to request. Both messages are signed — the two signatures together form the `CapabilityAgreement`. If the consumer wants different terms, they send a new request with adjusted parameters — no counter-offer complexity. This keeps capability negotiation under 2 seconds on WiFi and under 10 seconds on LoRa. |

Formal Verification: Priority-Ordered TLA+ Targets

———	———
Chosen	TLA+ for concurrent protocol properties; 4-tier priority: (1) CRDT merge, (2) payment channels, (3) epoch checkpoints, (4) full composition deferred
Alternatives	Coq/Lean theorem proving, no formal verification (testing only), verify everything before v1.0
Rationale	Coq/Lean have a steep learning curve that limits contributor access. Pure testing cannot prove absence of bugs in concurrent distributed protocols. Verifying everything delays launch indefinitely. TLA+ is battle-tested for distributed systems (used by Amazon for AWS services, by Microsoft for Azure) and has a practical learning curve. **Priority 1 — CRDT merge convergence** (must verify before v1.0): Prove commutativity, associativity, and idempotency of GCounter max-merge and GSet union. If merge diverges, the entire ledger is broken. **Priority 2 — Payment channel state machine** (must verify before v1.0): Prove no balance can go negative, dispute resolution always terminates within the challenge window, and channel states form a total order by sequence number. Direct financial impact if buggy. **Priority 3 — Epoch checkpoint correctness** (should verify): Prove no confirmed settlement is permanently lost after finalization, bloom filter false positive recovery covers all edge cases during the grace period. Property-based testing (QuickCheck-style) initially, formal TLA+ proof if resources allow. **Priority 4 — Full protocol composition** (defer to post-v1.0): Interaction between subsystems (e.g., channel dispute during epoch transition) is tracked as a long-term research goal. Individual component proofs provide sufficient confidence for launch.

<div class="page-break"></div>

Open Questions

All questions — architectural and implementation-level — have been resolved. This page tracks the full resolution history.

Resolved in v1.0 Spec Review (Round 4)

Implementation-level questions resolved with concrete specifications added to the relevant pages:

| # | Question | Resolution | Location |
|———|———|———|———|

1	Token Bucket Bandwidth Measurement	EMA with 60-second half-life; reset on transport change; 10% protocol overhead reserve	[Network Protocol](#protocol-network-protocol--congestion-control)
2	Epoch Active Set Conflict Resolution	5% settlement count threshold for ACK vs NAK; 3-round wait before re-propose; post-merge reconciliation	[CRDT Ledger](#economics-crdt-ledger--epoch-proposer-selection)
3	Mutable Object Fork Detection Reporting	5-step protocol: record, block 24h, gossip advisory, 7-day dedup, resolution via KeyCompromiseAdvisory	[MHR-Store](#services-mhr-store--mutable)
4	CongestionSignal on Multi-Interface Nodes	Replaced link_id with scope enum (ThisLink/AllOutbound); 3 bytes total	[Network Protocol](#protocol-network-protocol--congestion-control)
5	DHT Rebalancing Timing	2 gossip rounds convergence on join; 6 additional missed rounds before re-replication; graceful degradation	[MHR-DHT](#services-mhr-dht--rebalancing)
6	Beacon Collision Handling	Density adaptation: interval doubles at 50% utilization, triples at 75%; Ring 1 gossip provides redundancy	[Discovery](#marketplace-discovery--presence-beacons)
7	Fragment Reassembly	30s per-chunk timeout, exponential backoff (3 retries), 5-min overall, resumable via ChunkRequest	[MHR-Store](#services-mhr-store--reassembly)
8	Transitive Credit Rate-Limiting	Per-epoch, per-grantee tracking via CreditState struct; epoch-boundary reset; vouching peer absorbs defaults	[Trust Neighborhoods](#economics-trust-neighborhoods--trust-based-credit)

Resolved in v1.0 Spec Review (Round 3)

Protocol-level gaps identified in the third comprehensive review, resolved inline:

Gap	Resolution	Location
Serialization format (endianness, encoding)	Little-endian, fixed-size binary, TLV for extensions, normalized scores on decoded values	[Specification](#specification--serialization-rules)
CompactPathCost normalization ambiguity	Normalization uses decoded values, not log-encoded wire values	[Network Protocol](#protocol-network-protocol--greedy-forwarding-with-cost-weighting)
Agreement lifecycle (expiry, renewal, billing)	Defined Active/Expired/Grace/Closed states, per-capability expiry behavior, renewal protocol	[Agreements](#marketplace-agreements--agreement-lifecycle)
Settlement timing (eager vs lazy)	Settlements to CRDT ledger are not per-win; created on cooperative close, dispute, or periodic finalization	[Payment Channels](#economics-payment-channels--settlement-timing)
Channel sequence semantics	Sequence is monotonic version number; replay protection via final_sequence comparison	[Payment Channels](#economics-payment-channels--sequence-number-semantics)
CapabilitySummary cost encoding	Identical $\log_2$ formula to CompactPathCost; 0x0000=free, 0xFFFF=unknown	[Discovery](#marketplace-discovery--ring-1--2-3-hops)
Merkle root trust for constrained nodes	Signed by proposer; trusted peers accept immediately; untrusted requires 2-source quorum	[CRDT Ledger](#economics-crdt-ledger--merkle-root-trust)
Reputation initialization values	New peers start at 0; referrals capped at 5000; first-hand replaces after 5 interactions	[Security](#protocol-security--how-reputation-is-

earned) |

Resolved in Implementation Spec (Phase 1-2)

#	Question	Resolution	Location
1	WASM Sandbox Specification	Wasmtime runtime; Light (16 MB, 10^8 fuel, 5s) and Full (256 MB, 10^10 fuel, 30s) tiers; 10 host imports mirroring MHR-Byte System opcodes	[MHR-Compute](#services-mhr-compute--wasm-full-execution)
2	Presence Beacon Capability Bitfield	8 assigned bits (relay, gateway, storage, compute-byte, compute-wasm, pubsub, dht, naming); bits 8-15 reserved	[Discovery](#marketplace-discovery--presence-beacons)
3	Ring 1 Capability Aggregation Format	`CapabilitySummary` struct: 8 bytes per type (type, count, min/avg cost, min/max hops)	[Discovery](#marketplace-discovery--ring-1--2-3-hops)
4	DHT Metadata Format	`DHTMetadata` struct: 129 bytes (key, size, content_type, owner, ttl, lamport_ts, Ed25519 signature)	[MHR-DHT](#services-mhr-dht--metadata-format)
5	Negotiation Protocol Wire Format	Single-round take-it-or-leave-it; 30-second timeout; `CapabilityRequest` + `CapabilityOffer` with nonce replay prevention	[Agreements](#marketplace-agreements--negotiation)

Resolved in v1.0 Spec Review

#	Question	Resolution	Design Decision
1	Multi-admin group messaging	Delegated co-admin model (up to 3 co-admins, no threshold signatures)	[Design Decisions](#development-design-decisions--group-admin-delegated-co-admin-no-threshold-signatures)
2	Reputation gossip vs. first-hand only	Bounded 1-hop trust-weighted referrals, capped at 50%, advisory only	[Design Decisions](#development-design-decisions--reputation-bounded-trust-weighted-referrals)
3	Onion routing for high-threat environments	Per-packet layered encryption, opt-in via PathPolicy, 3 hops default, 21% overhead on LoRa	[Design Decisions](#development-design-decisions--onion-routing-per-packet-layered-encryption-opt-in)
4	MHR-Byte full opcode specification	47 opcodes in 7 categories, reference interpreter in Rust, ESP32-calibrated cycle costs	[Design Decisions](#development-design-decisions--mhr-byte-47-opcodes-with-reference-interpreter)
5	Bootstrap emission schedule parameters	10^12 µMHR/epoch initial, discrete halving every 100K epochs, 0.1% tail floor	[Design Decisions](#development-design-decisions--emission-schedule-epoch-counted-discrete-halving)
6	Protocol bridge design (SSB, Matrix, Briar)	Standalone gateway services, identity attestation, bridge operator pays Mehr costs	[Design Decisions](#development-design-decisions--protocol-bridges-standalone-gateway-services)
7	Formal verification targets	TLA+ priority-ordered: CRDT merge, payment channels, epoch checkpoints; composition deferred	[Design Decisions](#development-design-decisions--formal-verification-priority-ordered-tla-targets)

Resolved in v1.0 Spec Hardening

Protocol-level gaps identified during comprehensive spec review, all resolved inline in the relevant spec pages:

Gap	Resolution	Location
Announce propagation frequency	Event-driven + 30-min periodic refresh, 128-hop limit, 3-round link failure detection	[Network Protocol](#protocol-network-protocol--announce-propagation-rules)
ChaCha20 nonce handling	64-bit counter per session key, zero-padded to 96 bits	[Security](#protocol-security--link-layer-encryption-hop-by-hop)
Session key rotation timing	Local monotonic clock, either side can initiate	[Security](#protocol-security--link-layer-encryption-hop-by-hop)
KeyCompromiseAdvisory replay	Added monotonic `sequence` field	[Security](#protocol-security--key-compromise-advisory)
Difficulty target formula	Local per-link computation, `win_prob = target_updates / observed_packets`	[Payment Channels](#economics-payment-channels--adaptive-difficulty)
Settlement validation	Every node validates (2 sig checks + hash + GSet lookup), invalid dropped silently	[CRDT Ledger](#economics-crdt-ledger--settlement-flow)
Active set definition	Nodes appearing as party in at least 1 settlement in last 2 epochs	[CRDT Ledger](#economics-crdt-ledger--epoch-lifecycle)
Bloom filter construction	k=13 Blake3-derived hash functions, 19.2 bits/element at 0.01% FPR	[CRDT Ledger](#economics-crdt-ledger--bloom-filter-sizing)
Relay compensation tracking	RelayWinSummary with spot-check proofs, challengeable during grace period	[CRDT Ledger](#economics-crdt-ledger--relay-compensation-tracking)
Roaming cache fingerprint stability	60% overlap threshold for area recognition	[Discovery](#marketplace-discovery--roaming-cache)
Credit-based fast start staleness	Rate-limited grants, 2× balance requirement for staleness tolerance	[Discovery](#marketplace-discovery--credit-based-fast-start)
Trust relationship revocation	Asymmetric, revocable at any time, downgrades stored data priority	[Trust Neighborhoods](#economics-trust-neighborhoods--the-trust-graph)

<div class="page-break"></div>

Mehr Protocol Specification v1.0

This page is the normative reference for the Mehr protocol. Individual documentation pages provide detailed explanations; this page summarizes the protocol constants, wire formats, and layer dependencies in one place.

Status

| | | |

```
├────────┼────────┤
| **Version** | 1.0 |
| **Status** | Design complete, pre-implementation |
| **Normative sections** | Layers 0–5 (transport through services) |
| **Informative sections** | Layer 6 (applications), hardware reference, roadmap |
```


Protocol Constants

Constant	Value	Defined In
Gossip interval	60 seconds	[Network Protocol](#protocol-network-protocol--gossip-protocol)
Protocol overhead budget	≤10% of link bandwidth	[Bandwidth Budget](#protocol-network-protocol--bandwidth-budget)
CompactPathCost size	6 bytes (constant)	[Network Protocol](#protocol-network-protocol--mehr-extension-compact-path-cost)
MehrExtension magic byte	`0x4E` ('N')	[Network Protocol](#protocol-network-protocol--mehr-extension-compact-path-cost)
Destination hash size	16 bytes (128-bit)	[Network Protocol](#protocol-network-protocol--identity-and-addressing)
Smallest MHR unit	1 µMHR	[MHR Token](#economics-mhr-token--properties)
Supply ceiling	2^64 µMHR (asymptotic)	[MHR Token](#economics-mhr-token--supply-model)
Default relay lottery probability	1/100	[Stochastic Rewards](#economics-payment-channels--example)
Payment channel state size	200 bytes	[Payment Channels](#economics-payment-channels--channel-state)
Dispute challenge window	2,880 gossip rounds (~48h)	[Payment Channels](#economics-payment-channels--channel-lifecycle)
Channel abandonment threshold	4 epochs	[Payment Channels](#economics-payment-channels--channel-lifecycle)
Epoch trigger: settlement count	≥10,000 batches	[CRDT Ledger](#economics-crdt-ledger--epoch-triggers)
Epoch trigger: GSet memory	≥500 KB	[CRDT Ledger](#economics-crdt-ledger--epoch-triggers)
Epoch acknowledgment threshold	67% of active set	[CRDT Ledger](#economics-crdt-ledger--epoch-lifecycle)
Epoch verification window	4 epochs after activation	[CRDT Ledger](#economics-crdt-ledger--epoch-lifecycle)
Bloom filter FPR (epoch)	0.01%	[CRDT Ledger](#economics-crdt-ledger--bloom-filter-sizing)
DHT replication factor	k=3	[MHR-DHT](#services-mhr-dht--replication-factor)
DHT XOR weight (w_xor)	0.7	[MHR-DHT](#services-mhr-dht--lookup-scoring-function)
Storage chunk size	4 KB	[MHR-Store](#services-mhr-store--chunking)
Presence beacon size	20 bytes	[Discovery](#marketplace-discovery--presence-beacons)
Presence beacon interval	10 seconds	[Discovery](#marketplace-discovery--presence-beacons)
Transitive credit limit	10% per hop, max 2 hops	[Trust & Neighborhoods](#economics-trust-neighborhoods--trust-based-credit)


```
## Cryptographic Primitives

| Purpose | Algorithm | Output / Key Size |
├──────────────┼──────────────┼──────────────────┤
| Identity / Signing | Ed25519 | 256-bit (32-byte public key) |
| Key Exchange | X25519 (Curve25519 DH) | 256-bit |
| Identity Hashing | Blake2b | 256-bit → 128-bit truncated |
| Content Hashing | Blake3 | 256-bit |
| Symmetric Encryption | ChaCha20-Poly1305 | 256-bit key, 96-bit nonce |
| Relay Lottery (VRF) | ECVRF-ED25519-SHA512-TAI (RFC 9381) | 80-byte proof |
| Erasure Coding | Reed-Solomon | Configurable k,m |

<a id="specification--layer-dependency-graph"></a>

## Layer Dependency Graph
```

Layer 6: Applications ├── Messaging, Social, Voice, Naming, Forums, Hosting └── depends on ↓

Layer 5: Service Primitives ├── MHR-Store, MHR-DHT, MHR-Pub, MHR-Compute └── depends on ↓

Layer 4: Capability Marketplace ├── Discovery, Agreements, Verification └── depends on ↓

Layer 3: Economic Protocol ├── MHR Token, Stochastic Rewards, CRDT Ledger, Trust Neighborhoods └── depends on ↓

Layer 2: Security ├── Link encryption, E2E encryption, Authentication, Key management └── depends on ↓

Layer 1: Network Protocol ├── Identity, Addressing, Routing, Gossip, Congestion Control └── depends on ↓

Layer 0: Physical Transport └── LoRa, WiFi, Cellular, Ethernet, BLE, Fiber, Serial

```
Each layer depends **only** on the layer directly below it. Applications never touch
transport details. Payment never touches routing internals.

<a id="specification--serialization-rules"></a>

## Serialization Rules

All Mehr wire formats use the following conventions:

| Rule | Value |
├─────────────┼─────────────┤
| **Byte order** | Little-endian for all multi-byte integers (u16, u32, u64, i64) |
| **Encoding** | Fixed-size binary fields; no self-describing framing (not CBOR, not JSON)
|
| **TLV extensions** | Type (u8), Length (u8, max 255), Data (variable). Used in
```

```
MehrExtension only |
| **Strings** | UTF-8, length-prefixed with u16 (community labels, function IDs) |
| **Hashes** | Raw bytes, no hex encoding on the wire |
| **Signatures** | Raw 64-byte Ed25519 signatures, no ASN.1/DER wrapping |
| **Normalized scores** | Computed on **decoded** values, then divided by the max decoded
value in the candidate set. Result is IEEE 754 f32 on nodes that support FP; 16-bit fixed-
point (Q0.16, value × 65535) on constrained nodes. Both yield equivalent routing decisions
within rounding tolerance |


<a id="specification--wire-format-summary"></a>


## Wire Format Summary


<a id="specification--packet-format-reticulum-derived"></a>


### Packet Format (Reticulum-derived)
```

[HEADER 2B] [DEST_HASH 16B] [CONTEXT 1B] [DATA 0-465B] Max packet size: 484 bytes Source address: NOT PRESENT (structural sender anonymity)

```
<a id="specification--mehr-announce-extension"></a>


### Mehr Announce Extension
```

[MAGIC 1B: 0x4E] [VERSION 1B] [CompactPathCost 6B] [TLV extensions...] Minimum: 8 bytes. Carried in announce DATA field.

```
<a id="specification--compactpathcost"></a>


### CompactPathCost
```

[cumulative_cost 2B] [worst_latency_ms 2B] [bottleneck_bps 1B] [hop_count 1B] Total: 6 bytes (constant regardless of path length)

```
<a id="specification--payment-channel-state"></a>


### Payment Channel State
```

[channel_id 16B] [party_a 16B] [party_b 16B] [balance_a 8B] [balance_b 8B] [sequence 8B] [sig_a 64B] [sig_b 64B] Total: 200 bytes

```
<a id="specification--specification-sections"></a>
```

Specification Sections

Spec Section	Documentation Page
0. Design Philosophy	[Introduction](#introduction)
1. Layer 0: Physical Transport	[Physical Transport](#protocol-physical-transport)
2. Layer 1: Network Protocol	[Network Protocol](#protocol-network-protocol)
3. Layer 2: Security	[Security](#protocol-security)
4. Layer 3: Economic Protocol	[MHR Token](#economics-mhr-token), [Stochastic Relay Rewards](#economics-payment-channels), [CRDT Ledger](#economics-crdt-ledger), [Trust & Neighborhoods](#economics-trust-neighborhoods), [Real-World Economics](#economics-real-world-impact)
5. Layer 4: Capability Marketplace	[Overview](#marketplace-overview), [Discovery](#marketplace-discovery), [Agreements](#marketplace-agreements), [Verification](#marketplace-verification)
6. Layer 5: Service Primitives	[MHR-Store](#services-mhr-store), [MHR-DHT](#services-mhr-dht), [MHR-Pub](#services-mhr-pub), [MHR-Compute](#services-mhr-compute)
7. Layer 6: Applications	[Messaging](#applications-messaging), [Social](#applications-social), [Voice](#applications-voice), [Naming](#applications-naming), [Community Apps](#applications-community-apps), [Hosting](#applications-hosting)
8. Hardware Reference	[Reference Designs](#hardware-reference-designs), [Device Tiers](#hardware-device-tiers)
9. Implementation Roadmap	[Roadmap](#development-roadmap)
10. Design Decisions	[Design Decisions](#development-design-decisions)
11. Open Questions	[Open Questions](#development-open-questions)

Version History

Version	Status	Description
v0.1-v0.5	Superseded	Early design iterations
v0.8	Superseded	Introduced explicit community zones (later replaced by trust neighborhoods)
v1.0	**Current**	Consolidated specification — Reticulum foundation, stochastic relay rewards, emergent trust neighborhoods

---

This specification consolidates design work from v0.1 through v1.0. The foundation — Reticulum-based transport, cryptographic identity, Kleinberg small-world routing, stochastic relay rewards, CRDT settlement, epoch compaction, emergent trust neighborhoods, and the capability marketplace — is the protocol. Everything above it — storage, compute, pub/sub, naming, and applications — are services built on that foundation.

<div class="page-break"></div>