Mehr Protocol

Complete Specification

Version 1.0 — Design Complete, Pre-Implementation

Generated February 27, 2026

Table of Contents

Mehr Network

Decentralized Mesh Infrastructure Powered by Proof of Service

Proof of work wastes electricity. Proof of stake rewards capital, not contribution. Mehr uses **proof of service** — a token is minted only when a real service is delivered to a real paying client through a funded payment channel. Relay a packet, store a block, run a computation — that's how MHR enters circulation. No work is wasted. No token is unearned.

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.

What Makes Mehr Different

Proof of Service

Most decentralized networks create tokens through artificial work (hashing) or capital lockup (staking). Mehr mints tokens only when a provider delivers a real service — relaying traffic, storing data, or executing computations — to a client who pays through a funded payment channel. Minting is proportional to real economic activity and capped at 50% of net service income. A 2% burn on every payment creates a deflationary counterforce that keeps supply bounded.

Zero Trust Economics

The economic layer assumes every participant is adversarial. Two mechanisms make cheating structurally unprofitable in connected networks: **non-deterministic service assignment** (the client can't choose who serves the request) and a **net-income revenue cap** (cycling MHR produces zero minting). In isolated partitions, active-set-scaled emission and the service burn bound damage to a finite equilibrium. No staking, no slashing, no trust scores required.

Free Between Friends

Communication within your trust network is free — no tokens, no channels, no economic overhead. A local mesh where everyone trusts each other operates at zero cost. The economic layer only activates when traffic crosses trust boundaries. This mirrors how communities actually work: you help your neighbors for free, but charge strangers for using your infrastructure.

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.

If someone wants to use the network without running a relay, they can acquire MHR from someone who has earned it. The network doesn't care how you got your tokens — only that you have them.

Finding Things

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

Subscribe to your city's geographic feed. Content on Mehr is tagged with hierarchical scopes — geographic (where it's relevant) and interest (what it's about). Your app uses these to show you relevant content:

1. **Neighborhood feed**: Everything posted by people near you. Cheapest and fastest — content is close.
2. **City feed**: Everything from your city. Subscribe to Portland and you see all Portland neighborhoods. Costs a bit more (further away) but still local.
3. **Region/country feed**: Broader scope, higher cost, more content. Use digest or headline-only mode to keep bandwidth low.
4. **Interest feeds**: Subscribe to topics — gaming, science, music, local events. These connect you with people who share your interests, regardless of where they live.
5. **Curated feeds**: Follow a curator — a real person who picks the best content. Like subscribing to a newsletter.

How do I find my friends?

By their name. Everyone picks a name scoped to their location — like `alice@geo:us/oregon/portland` or `ravi@geo:india/mumbai`. 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 discover each other automatically over radio or WiFi — no names needed.

Is there a "feed" like Instagram or Twitter?

Yes, but you control it. There are five types of feeds:

- **Follow feed**: You follow people. Their posts appear in chronological order.
- **Geographic feed**: Everything from your neighborhood, city, or region.
- **Interest feed**: Everything tagged with a topic you care about (gaming, science, cooking).

- **Intersection feed**: Combine geographic + interest ("Portland Pokemon" = posts tagged with both).
- **Curated feed**: A human you trust picks the best content and publishes a list. Like a magazine editor.

There is no algorithm deciding what you see. No ads. No engagement optimization. Your feed is what you chose to follow.

How does browsing work?

Every post has two layers:

1. **Free preview**: A headline, short summary, and blurry thumbnail. You see these for free — browse as much as you want.
2. **Full content**: The actual article, image, video, or file. You pay a tiny fee to open it, and a share goes to the author.

This means you never pay for content you don't want. You browse previews like window shopping and only pay when you open something.

Can I browse websites?

Yes. People host websites on Mehr without needing a server or domain name. You visit them by name (`mysite@geo:us/oregon/portland`) or by direct link. Popular sites load fast because copies are cached throughout the network automatically.

Creating Content

How do I post something?

Write your post, add a headline and summary, optionally attach images or files, and publish. Your app creates a free preview (the envelope) and the paid content (the post) automatically.

You choose who sees it:

- **Geographic scope**: Tag your city, region, or neighborhood. Appears in geographic feeds.
- **Interest scope**: Tag a topic. Appears in interest feeds for that topic.
- **Both**: Tag a city AND a topic. Appears in both feeds.
- **No scope**: Only your trusted peers see it. The most private option.

Does it cost money to post?

Yes — a tiny amount. Every post is stored on the network, and storage costs MHR. This is the anti-spam mechanism: posting costs tokens, so flooding the network with garbage is economically irrational.

Within your trust network (friends and neighbors), posting is free.

Can I earn from my content?

Yes. When someone pays to read your full post, a portion of their fee goes back to you — this is called **kickback**. You set the percentage when you publish (default is about 50%).

Popular content that earns more kickback than it costs to store becomes **self-funding** — it lives as long as people read it, at no cost to you. Content nobody reads expires when you stop paying for storage.

What kinds of content can I publish?

Anything: text posts, photo essays, music albums, video courses, scientific papers, games, software, podcasts. The same envelope/post system works for all content types. The preview shows whatever makes sense (track listing for music, abstract for papers, screenshots for games).

What about curators?

Anyone can be a curator. You create a curated feed — a list of the best posts you've found — and publish it. Others subscribe to your feed. When they read posts you recommended, the original authors earn kickback AND you earn a separate fee for the curation. Two people get paid: the creator and the curator.

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).
- **On a voice call while moving?** The call hands off between nodes with less than a second of interruption. Quality may change but the call doesn't drop.

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 — a trust neighborhood. Nobody "creates" it or "runs" it — it emerges from real-world relationships.

Each person tags themselves with where they are (e.g., Portland, Oregon) and what they're into (e.g., gaming, science). These tags — called scopes — are how feeds and names work. No authority approves your tags. Communities converge on naming through social consensus, the same way they do today.

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) tagged with your geographic scope, and it's visible to your neighborhood. Buyers contact you directly. Payment can happen in person, through an external service, or through MHR escrow.

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@geo:us/or/portland`)
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.

Can I store my files on the network?

Yes. Mehr provides decentralized cloud storage — like Dropbox, but your files are encrypted on your device before being stored across multiple mesh nodes. No cloud provider has access to your files. Your devices sync automatically through the mesh. You can share files with specific people by granting them a decryption key.

If you don't want to deal with tokens, a gateway operator can offer cloud storage as a fiat-billed service — same experience as any cloud storage app, but backed by the mesh.

Can I earn by sharing my storage?

Yes — and it's one of the easiest ways to start earning MHR. Any device with spare disk space can offer storage services. You configure how much space to share, storage nodes advertise their availability, and clients form agreements with you. You earn μMHR for every epoch your storage is used. No special hardware needed — a Raspberry Pi with a USB drive works fine.

What happens when I move around?

Your device roams seamlessly. Mehr identity is your cryptographic key, not a network address. When you walk from WiFi to LoRa range to another WiFi node, your connections don't drop — traffic shifts to the best available transport in under a second. Apps adapt to link quality (images become previews on slow links, full quality returns on fast links). You can even plug an ethernet cable into different ports at different locations and stay connected with zero configuration.

Privacy and Safety

Is it private?

Yes. Messages are end-to-end encrypted. Social posts can be public (scoped) or neighborhood-only (unscoped). 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.

What about illegal or harmful content?

There is no central moderator. Instead, content governance is distributed:

- **Every node decides for itself** what to store, relay, and display. No node is forced to host or forward content it objects to.
- **Trust revocation** is the enforcement mechanism. If your community discovers you're producing harmful content, they remove you from trusted peers — cutting off your free relay, storage, credit, and reputation.
- **Economics limits abuse**: posting costs money, content starts local (doesn't go global without genuine demand), and there's no algorithm to amplify engagement.
- **Curators filter quality**: most readers follow curated feeds, not raw unfiltered streams.

This is the same tradeoff every free society makes: individual freedom with social consequences. No central authority decides what's allowed, but communities enforce their own norms.

Economy

How does money work on Mehr?

MHR is the network's internal token. Think of it like arcade tokens — valuable inside the arcade (network services), designed to be spent.

- **You earn MHR** by relaying traffic, storing data, or providing other services
- **You spend MHR** when your messages cross through untrusted infrastructure, or when you read paid content
- **Content creators earn MHR** through kickback — a share of what readers pay
- **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. But because it buys real services (bandwidth, storage, compute, content), it has real value — and people will likely trade it informally. This is fine. The network's health doesn't depend on preventing exchange; it works as a closed-loop economy regardless.

Can I buy MHR instead of earning it?

Yes. If someone sells you MHR they earned through relay work, you can spend it on the network. The seller earned those tokens through real service — the network benefited. You're indirectly funding infrastructure. This is no different from buying bus tokens.

What if I don't want to run a relay? Can I just pay to use the network?

Yes. **Gateway operators** handle this. A gateway is a regular node that accepts fiat payment (subscription, prepaid, or pay-as-you-go) and gives you network access in return. From your perspective, you sign up, pay a monthly bill, and use the network — just like a phone plan. You never see or touch MHR tokens.

The gateway adds you as a trusted peer and extends credit, so your traffic flows through them for free. The gateway handles MHR costs on your behalf. Multiple gateways compete in any area, so pricing stays competitive. You can switch gateways at any time — your identity is yours, not the gateway's.

See Gateway Operators for details.

Can I get rich from MHR?

That's not the point. MHR is designed to be spent on services, not hoarded. There's no ICO and no hidden allocation — the genesis gateway receives a transparent, disclosed allocation visible in the ledger from day one. Tail emission (0.1% annual) mildly dilutes idle holdings. Lost keys permanently remove supply. The economic incentive is to earn and spend, not to accumulate.

Licensing and Digital Assets

Can I sell licenses for my work on Mehr?

Yes. Mehr has a built-in digital licensing system. You publish a **LicenseOffer** alongside your asset (photo, music, software, dataset) specifying terms — price, whether derivatives are allowed, whether commercial use is permitted, and how many licenses can be issued. Buyers pay you directly (in MHR or fiat) and receive a **LicenseGrant** signed by both parties.

How does license verification work?

A LicenseGrant is cryptographically signed by both the licensor and licensee. Anyone can verify it by checking the Ed25519 signatures — no network connection needed. When someone uses a licensed asset in a derivative work, they include the LicenseGrant hash in their post's references. Readers can follow the chain: derivative work → LicenseGrant → LicenseOffer → original asset.

Can licenses be enforced?

Not at the protocol level. Mehr proves a license exists (or doesn't) — it doesn't prevent unlicensed use. This is the same as the real world: copyright exists whether or not someone violates it. Enforcement happens through social reputation (community trust) and legal systems (courts). The cryptographic proof makes disputes straightforward to resolve.

Do licenses work outside of Mehr?

Yes. A LicenseGrant contains public keys and signatures that can be verified with standard cryptographic tools — no Mehr software needed. A website, archive, or court can verify license authenticity from the grant alone. The rights described in the license apply wherever the parties intend them to, not just on the Mehr network.

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)
Content creators earn	Platform takes most/all revenue	Direct kickback to creator (~50%)

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.

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. Here, Mehr provides communication where there was none, or shares 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

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.

How is this different from Mastodon/Bluesky?

Mastodon and Bluesky are decentralized social networks that still require internet access and depend on servers run by someone. On Mehr:

	Mastodon/Bluesky	Mehr
Requires internet	Yes	No — works on radio alone
Requires servers	Yes (someone hosts instances)	No — content lives on mesh nodes
Content moderation	Server admin decides	Each node decides for itself
Posting cost	Free	Small fee (anti-spam)
Creator revenue	None built-in	Kickback on every read
Works offline	No	Yes — local mesh continues

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, LTE-M, NB-IoT, 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
LTE-M	0.375-1 Mbps	Via carrier	Full	Licensed LPWAN; better building penetration than LoRa, carrier-managed
NB-IoT	0.02-0.25 Mbps	Via carrier	Half	Licensed LPWAN; extreme range and battery life, carrier-managed
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

```
graph LR
    RS["Remote Sensor"] ⟵ "LoRa (10 kbps)" ⟶ BN["Bridge Node"]
    BN ⟵ "WiFi (100 Mbps)" ⟶ GW["Gateway"]
```

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 **$O(\log^2 N)$ 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 **$O(\log^2 N)$ 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 **$O(\log^2 N)$**. 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.

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

### 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>


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

# 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. DNS is used only for initial [genesis gateway discovery](#economics-
mhr-token--genesis-gateway-discovery), not for protocol operation.

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

## 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. DNS is
used only for initial genesis gateway discovery, not for protocol operation.
- **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 (demand-backed)**: 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 — preventing grinding. However, VRF alone does
not prevent traffic fabrication between colluding nodes. The actual Sybil defense is
[demand-backed minting](#economics-payment-channels--demand-backed-minting-eligibility):
VRF wins only count for minting if the packet traversed a funded payment channel, and
[revenue-capped minting](#economics-payment-channels--revenue-capped-minting) ensures self-
dealing is always unprofitable (spending Y MHR on fake traffic yields at most 0.5Y in
minting).
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.

<a id="protocol-security--cryptographic-primitives-summary"></a>

## 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 |

<a id="protocol-security--hash-algorithm-split"></a>

### 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.

<a id="protocol-security--ed25519-to-x25519-conversion"></a>

### 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: Genesis service allocation + demand-backed proof-of-service mining (no ICO) Genesis allocation: Disclosed amount to genesis gateway operator (see Genesis below) 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: halving_shift = min(e / 100_000, 63) // clamp to prevent undefined behavior epoch_reward(e) = max(10^12 >> halving_shift, // 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

Implementation note: the shift operand MUST be clamped to 63 (max for u64). At epoch 6,400,000 (~year 1218), unclamped shift = 64 which is undefined behavior on most platforms. Clamping to 63 yields 0 (10^12 >> 63 = 0), so the tail floor takes over — correct behavior.

```
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.

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

### 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.

<a id="economics-mhr-token--why-relay-only-minting"></a>

## Why Relay-Only Minting

Only relay earns minting rewards. Storage and compute earn through bilateral payments, not
minting. This is deliberate.
```

Service minting analysis:

Relay: ✓ Minting reward (VRF lottery, demand-backed) VRF prevents grinding — exactly one output per (relay, packet) pair. But VRF alone does NOT prevent traffic fabrication: a Sybil attacker can fabricate traffic between colluding nodes and run the VRF lottery on fake packets. The actual Sybil defense is demand-backed minting: VRF wins only count for minting if the packet traversed a funded payment channel. Fabricating funded-channel traffic requires spending real MHR, and revenue-capped minting ensures the attacker always loses money (see Revenue-Capped Minting below).

Storage: ✗ No minting reward Storage proofs can be gamed: store your own garbage data, respond to your own challenges, claim minting. A bilateral StorageAgreement requires TWO signatures — but a Sybil node could sign both sides.

Compute: ✗ No minting reward Compute is demand-driven. "Mining" computation without a requester would incentivize wasted work. Unlike relay (which serves others), un-requested computation serves nobody.

```
Storage and compute don't need minting because they bootstrap through the free tier and
bilateral payments:
```

Bootstrap sequence by service type:

Phase 0: FREE TIER ├── Relay: Trusted peers relay for free (works immediately) ├── Storage: Trusted peers store each other's data for free └── Compute: Nodes run their own contracts locally

Phase 0.5: GENESIS SERVICE GATEWAY ├── Genesis gateway receives transparent MHR allocation ├── Gateway offers real services for fiat (relay, storage, compute) ├── Consumer fiat → MHR credit extensions → funded channels └── Real relay demand enters the network for the first time

Phase 1: DEMAND-BACKED RELAY MINTING ├── Funded-channel traffic triggers VRF lottery ├── VRF wins on funded channels earn minting rewards ├── Revenue-capped minting prevents self-dealing (see below) └── MHR enters circulation backed by real demand

Phase 2: SPENDING ├── Relay earners spend MHR on paid storage agreements ├── Relay earners spend MHR on compute delegation └── Storage/compute providers now have MHR income

Phase 3: MATURE ECONOMY ├── Bilateral payments dominate all services ├── Minting becomes residual (decaying emission) └── Service prices emerge from supply/demand

```
Relay is the right bootstrap mechanism because it's the most universal service — every node
relays. A $30 solar relay earns minting rewards by forwarding packets, then spends those
tokens on storage and compute from more capable nodes. The minting subsidy flows from the
most common service to the rest of the economy.

**Sharing storage is another low-barrier entry point.** Any device with spare disk space
can offer [cloud storage](#) and earn MHR through bilateral payments. While storage doesn't
earn minting rewards, it earns directly from users who need their files stored and
replicated. The marginal cost is near zero (idle disk space), so even modest demand
generates income. For users who want to participate in the economy without running relay
infrastructure, storage is the simplest starting point.

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

## Economic Architecture
```

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

```mermaid
graph LR
    subgraph TRUST["TRUST NETWORK (free)"]
        Alice["Alice"] <-->|"free"| Bob["Bob"]
        Alice <-->|"free"| Dave["Dave"]
    end

    subgraph PAID["PAID ECONOMY (MHR)"]
        Bob -->|"relay"| Carol["Carol"]
        Carol -->|"pays relay fee"| Services["Storage\nCompute\nContent"]
        Bob -->|"lottery win?"| Mint["Mint +\nChannel debit"]
        Mint --> Services
    end
~~~

Free Tier (Trust-Based)

- Traffic between trusted peers 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
- Relay nodes earn MHR probabilistically — same expected income, far less overhead
- Settled via 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

Demand-Backed Proof-of-Service Mining (MHR Genesis)

The stochastic relay lottery serves a dual purpose: it determines who earns and how much, while the **funding source** depends on the economic context:

1. **Minting (subsidy, demand-backed)**: Each epoch, the emission schedule determines the minting ceiling. Actual minting is distributed proportionally to relay nodes based on their accumulated VRF lottery wins during that epoch — but only wins on packets that traversed a **funded payment channel** are minting-eligible. Free-tier trusted traffic does not earn minting rewards. This demand-backed requirement ensures minting reflects real economic activity, not fabricated traffic.

2. **Channel debit (market)**: When a relay wins the lottery and has an open payment channel 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 >> (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
```

Genesis Service Gateway

The bootstrapping problem is solved by a **Genesis Service Gateway** — a known, trusted operator that provides real services for fiat and bootstraps the MHR economy with genuine demand:

1. **Transparent allocation**: The genesis gateway operator receives a disclosed MHR allocation. No hidden allocation, no ICO — the amount is visible in the ledger from epoch 0.
2. **Competitive fiat pricing**: The gateway offers relay, storage, and compute at market-competitive fiat prices (see Initial Pricing below).
3. **Funded channels**: Consumer fiat payments are converted to MHR credit extensions, creating funded payment channels. This generates the first real relay demand on the network.
4. **Demand-backed minting**: Real relay traffic through funded channels triggers the VRF lottery. Winning relays earn minting rewards backed by actual economic activity.

5. **MHR circulation**: Minted MHR enters circulation — relay operators can spend it on storage, compute, or other services.
6. **Decentralization**: As more operators join and offer competing services, the genesis gateway becomes one of many providers. The economy transitions from gateway-bootstrapped to fully market-driven.

Bootstrap Sequence

1. Genesis gateway receives transparent MHR allocation, begins offering fiat-priced services
2. Nodes form local meshes (free between trusted peers, no tokens)
3. Consumers pay fiat to genesis gateway → funded channels created
4. Funded-channel traffic triggers demand-backed relay minting (VRF-based)
5. Lottery wins on funded channels accumulate as service proofs; epoch minting distributes MHR to relays
6. Relay nodes open payment channels and begin spending MHR on services
7. More operators join, offer competing services, prices fall toward marginal cost
8. Market pricing emerges from supply/demand

Trust-Based Credit

Trusted peers can vouch for each other 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 a configurable ratio (default 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.

Revenue-Capped Minting

The emission schedule sets a ceiling, but actual minting per epoch is capped at a fraction of real relay fees collected. This makes self-dealing always unprofitable:

```
Revenue-capped minting formula:

  effective_minting(epoch) = min(
      emission_schedule(epoch),                   // halving ceiling (10^12 >> shift)
      minting_cap × total_channel_debits(epoch)    // 0.5 × actual relay fees
  )

  minting_cap = 0.5  (minting can never exceed 50% of relay revenue)
```

Why this makes self-dealing unprofitable:

An attacker who pays fiat to acquire MHR, then spends it on fake relay traffic to their own Sybil nodes, always loses money:

```
Self-dealing attack analysis:

  1. Attacker pays $X fiat → gets Y MHR
  2. Attacker spends Y MHR on relay fees (fake traffic through own nodes)
  3. Maximum minting across ALL relays in the epoch = 0.5 × total_channel_debits
  4. Even if attacker captures 100% of all minting: gets back at most 0.5 × Y MHR
  5. Net result: spent Y, received ≤ 0.5Y → net loss of ≥ 0.5Y

  This holds regardless of epoch, traffic volume, or attacker's share of the network.
  The minting_cap guarantees self-dealing is unprofitable at every scale.
```

What happens to "unminted" emission:

- During early bootstrap, total relay fees are small, so actual minting is well below the emission schedule
- The difference is NOT minted — it is simply not created (supply grows slower)
- As traffic grows, actual minting approaches the emission schedule ceiling
- In mature economy, the cap is rarely binding (relay fees far exceed the emission schedule)

This changes the supply curve: instead of predictable emission, supply growth tracks actual economic activity. Early supply grows slowly (good — prevents speculation without real usage), mature supply follows the emission schedule.

Initial Pricing

The genesis gateway prices services at or slightly above market competitors. This is deliberate — the goal is fair pricing with operational margin, not undercutting.

```
Initial pricing strategy:

  Principle: Price at market rate with overhead, NOT undercutting.

  The genesis gateway publishes maximum prices (ceilings). These serve as a
  ceiling that competitors can undercut as they join. The gateway can initially
  run on AWS/cloud infrastructure — it needs margin to cover that cost.

  Service             Market Benchmark             Genesis Ceiling
  ────────────────────────────────────────────────────────────────────
  Storage             AWS S3: $0.023/GB/mo         ~$0.02/GB/mo
  Internet gateway    ISP: $30-100/mo              ~$30/mo
  Compute             AWS Lambda: ~$0.20/1M req    At market
  Relay (per-packet)  Bundled in gateway price     ~5 μMHR
  Rationale:
  - Storage: At market, not below — no reason to subsidize
  - Gateway: Match ISP rate; value is privacy/resilience, not cheapness
  - Compute: No reason to undercut cloud pricing initially
  - Relay: Derived from gateway fiat price ÷ expected packet volume
```

How prices fall over time:

```
Price evolution:

  Genesis:     Gateway sets ceiling (market rate + overhead)
  Growth:      New providers enter, set prices ≤ ceiling to attract users
  Maturity:    Competition drives prices toward marginal cost
               (Mehr's marginal cost is low — spare bandwidth/disk on existing devices)
```

The genesis gateway doesn't need to be cheapest. It needs to be **trusted, available, and fairly priced**. Price competition comes from the market, not from subsidized undercutting. The gateway's fiat-to-MHR conversion rate becomes the initial exchange rate for MHR.

Genesis Gateway Discovery

New nodes discover the genesis gateway through DNS:

```
Genesis gateway discovery:

  1. Well-known DNS domain resolves to genesis gateway IP(s)
  2. Hardcoded fallback list in daemon binary (in case DNS is unavailable)
  3. DNS is for initial contact only — once connected, gossip takes over
  4. Multiple DNS records for redundancy (A/AAAA records)

  Note: DNS is used ONLY for initial genesis gateway discovery,
  not for ongoing protocol operation. See roadmap Milestone 1.2.
```

This ties into the existing bootstrap mechanism (Milestone 1.2 in the roadmap), elevating DNS from "optional" to the primary method for locating genesis gateways.

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 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.

Fiat Exchange

MHR has no official fiat exchange rate. The protocol includes no exchange mechanism, no order book, no trading pair. But MHR buys real services — bandwidth, storage, compute, content access — so it has real value. People will trade it for fiat currency, whether through informal markets, OTC trades, or external exchanges.

This is expected and not inherently harmful.

Why exchange doesn't break the system:

```
graph TD
    Operator["Operator earns MHR"] ⟶ Sells["Sells for fiat"]
    Sells ⟶ Pays["Pays electricity bill"]
    Sells ⟶ Buyer["Buyer gets MHR"]
    Buyer ⟶ Spends["Spends on network services"]
    Operator ⟶ RealWork["Network received\nreal work (relay)"]
    Buyer ⟶ RealDemand["Network received\nreal demand (usage)"]
```

1. **Purchased MHR is legitimate.** If someone buys MHR with fiat instead of earning it through relay, the seller earned it through real work. The network benefited from that work. The buyer funds network infrastructure indirectly — identical to buying bus tokens.

2. **MHR derives value from utility.** Its value comes from the services it buys, not from artificial scarcity. If the service economy is healthy, MHR has value regardless of exchange markets.

3. **Hoarding is self-correcting.** Someone who buys MHR and holds it is funding operators (paying fiat for earned MHR) while removing tokens from circulation. Remaining MHR becomes more valuable per service unit, incentivizing earning through service provision. Tail emission (0.1% annual) mildly dilutes idle holdings.

What could go wrong:

Risk	Mitigation
Deflationary spiral (hoarding prevents spending)	Tail emission; free tier ensures basic functionality regardless
Speculation (price detaches from utility)	Utility value creates a floor; MHR has no use outside the network
Regulatory attention	Protocol doesn't facilitate exchange; users must understand their jurisdiction

Internal price discovery still works as designed — service prices float based on supply and demand:

```
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 fiat. They need to know: "Can I afford this service?" — and the answer is usually yes, because they earn MHR by providing services. The economy is circular even if some participants enter through fiat exchange.

Gateway Operators (Fiat Onramp)

The Genesis Service Gateway is the first instance of this pattern. The same mechanics — trust extension, credit lines, fiat billing — apply to all subsequent gateway operators. As more gateways join, the economy decentralizes and pricing becomes competitive.

Not everyone wants to run a relay. Pure consumers — people who just want to use the network — should be able to pay with fiat and never think about MHR. **Gateway operators** make this possible.

A gateway operator is a trusted intermediary who bridges fiat payment and MHR economics. The consumer interacts with the gateway; the gateway interacts with the network. This uses existing protocol mechanics — no new wire formats or consensus changes.

```
graph LR
    subgraph Consumer
        Signup["Signs up\n(fiat payment)"]
        Uses["Uses network\n(messages, content,\nstorage, etc.)"]
        Bill["Monthly fiat bill"]
    end

    subgraph Gateway
        Trust["Adds consumer to trusted_peers\nExtends credit via CreditState"]
        Relay["Gateway relays for free\n(trusted peer = free)"]
        Earns["Gateway earns MHR through\nrelay + receives fiat\nfrom consumers"]
    end

    subgraph Network
        Paid["Paid relay\nto wider network\n(gateway pays MHR)"]
    end

    Signup --> Trust
    Uses --> Relay
    Relay --> Paid
    Bill --> Earns
```

How it works:

1. **Sign-up**: Consumer pays the gateway in fiat (monthly subscription, prepaid, pay-as-you-go — the gateway chooses its business model)
2. **Trust extension**: Gateway adds the consumer to `trusted_peers` and extends a credit line via CreditState. The consumer's traffic through the gateway is free (trusted peer relay)
3. **Network access**: The consumer uses the network normally. Their traffic reaches the gateway for free, and the gateway pays MHR for onward relay to untrusted nodes
4. **Settlement**: The gateway earns MHR through relay minting + charges fiat to consumers. The spread between fiat revenue and MHR costs is the gateway's margin

The consumer never sees MHR. From their perspective, they pay a monthly bill and use the network. Like a mobile carrier — you don't think about interconnect fees between networks.

```
Trust-based gateway mechanics:

  Gateway's TrustConfig:
    trusted_peers: { consumer_1, consumer_2, ... }
    cost_overrides: { consumer_1: 0, consumer_2: 0 }  // free for consumers

  Gateway's CreditState per consumer:
    credit_limit: proportional to fiat subscription tier
    rate_limit: prevents abuse (e.g., 10 MB/epoch for basic tier)

  Consumer's view:
    - No MHR wallet needed
    - No payment channels
    - No economic complexity
    - Just "install app, sign up, use"
```

Why this works without protocol changes:

Mechanism	Already Exists
Free relay for trusted peers	Trust Neighborhoods
Credit extension	CreditState
Rate limiting	Per-epoch credit limits in CreditState
Abuse prevention	Gateway revokes trust on non-payment (fiat side)

Gateway business models:

Model	Description	Consumer Experience
Subscription	Monthly fiat fee for a usage tier	Like a phone plan
Prepaid	Buy credit in advance, use until depleted	Like a prepaid SIM
Pay-as-you-go	Fiat bill based on actual usage	Like a metered utility
Freemium	Free tier (rate-limited) + paid upgrade	Like free WiFi with premium option

Gateway incentives:

- Gateways earn relay minting rewards (they relay traffic between consumers and the wider network)
- Gateways earn fiat from consumer subscriptions
- Gateways with many consumers become valuable relay hubs — more traffic = more lottery wins = more MHR minting
- Competition between gateways drives prices toward cost (standard market dynamics)

Risks and mitigations:

Risk	Mitigation
Gateway goes down	Consumer can switch gateways or run their own node. No lock-in — identity is self-certifying
Gateway censors	Consumer switches gateway. Multiple gateways compete in any area with demand
Gateway overcharges	Market competition. Consumers compare pricing. Low switching cost
Consumer abuses gateway	Gateway revokes trust, cuts off credit. Fiat non-payment handled off-protocol

Gateways are not privileged protocol participants. They are regular nodes that choose to offer a service (fiat-to-network bridging) using standard trust and credit mechanics. Anyone can become a gateway operator — the barrier is having enough MHR to extend credit and enough fiat customers to sustain the business.

Economic Design Goals

- **Utility-first**: MHR is designed for purchasing services. Fiat exchange may emerge but the protocol's health doesn't depend on it, and the internal economy functions as a closed loop for participants who never touch fiat.
- **Transparent genesis**: Disclosed genesis allocation to the gateway operator, visible in the ledger from epoch 0. No ICO, no hidden allocation, no insider advantage.
- **Demand-backed minting**: Funded payment channels required for minting eligibility. Fabricated traffic through unfunded channels earns nothing. Revenue-capped emission guarantees self-dealing is always unprofitable.
- **Spend-incentivized**: Tail emission (0.1% annual) mildly dilutes idle holdings. Lost keys (~1–2% annually) permanently remove supply. MHR earns nothing by sitting still — only by being spent on services or lent via trust-based credit.
- **Partition-safe**: The economic layer works correctly during network partitions and converges when they heal
- **Minimal overhead**: Stochastic rewards 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.

Long-Term Sustainability

Does MHR stay functional for 100 years?

Economic Equilibrium

```
Supply dynamics over time:

  Year 0-10:   High minting emission, rapid supply growth
               Lost keys: ~1-2% annually (negligible vs. emission)
               Economy bootstraps

  Year 10-30:  Minting decays significantly (many halvings)
               Lost keys accumulate (~10-40% of early supply permanently gone)
               Effective circulating supply stabilizes

  Year 30+:    Tail emission ≈ lost keys
               Roughly constant effective supply
               All income is from bilateral payments + residual minting
```

The tail emission exists specifically for this: it ensures relay operators always have a minting incentive, even centuries from now. Lost keys and tail emission create a rough equilibrium — new supply enters through service, old supply exits through lost keys. Neither grows without bound.

Technology Evolution

Challenge	Mehr's Answer
New radio technologies	Transport-agnostic — any medium that moves bytes works
Post-quantum cryptography	KeyRotation claims enable key migration; new algorithms plug into existing identity framework
Hardware evolution	Capability marketplace adapts — nodes advertise what they can do, not what they are
Protocol upgrades	Open question — no formal governance mechanism yet. Communities can fork; the trust graph is the real network, not the protocol version

What Doesn't Change

The fundamental economic model — free between trusted peers, paid between strangers — is as old as human commerce. It doesn't depend on any specific technology, cryptographic primitive, or hardware platform. As long as people want to communicate and are willing to help their neighbors, the model works.

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

```
graph TD
    A["Packet arrives at relay node"] ⟶ B["Forward packet"]
    B ⟶ C["VRF Lottery: VRF(relay_key, packet_hash)"]
    C ⟶ D{"output < target?"}
    D -- "YES (1/100)" ⟶ E["Win! 500 μMHR"]
    D -- "NO (99/100)" ⟶ F["Nothing (no overhead)"]
    E ⟶ G["Channel debit (sender pays)"]
    E ⟶ H["Mining proof (epoch minting)"]
```

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), which reuses the relay's existing Ed25519 keypair. VRF proof size is 80 bytes, included only in winning lottery claims (not in every packet).

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.

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.

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

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
```

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. 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.

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.

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:

```
graph LR
    Alice --> Bob --> Carol --> Dave
    Bob -. "lottery?" .-> Bob
    Carol -. "lottery?" .-> Carol
```

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** (demand-backed): The VRF proof is accumulated as a service proof entitling the relay to a share of the epoch's minting reward — but only if the packet traversed a funded payment channel. Free-tier trusted traffic does not earn minting rewards. This demand-backed

requirement ensures minting reflects real economic activity. Minting 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.

Demand-Backed Minting Eligibility

A VRF lottery win is **minting-eligible** only if the packet that triggered it traversed a funded payment channel with the upstream sender. This is the core defense against Sybil traffic fabrication:

```
Minting eligibility rule:

  VRF win on packet P at relay R:
    IF upstream channel(sender, R) is funded (balance > 0):
      → Win counts toward R's epoch minting share
    ELSE (free-tier trusted traffic, or no funded channel):
      → Win does NOT count toward minting

  Why:
    Without this rule, a Sybil attacker can fabricate traffic between
    colluding nodes, run the VRF lottery on fake packets, and claim
    minting rewards for zero-cost "work."

    With this rule, generating minting-eligible traffic requires
    spending real MHR through funded channels — and revenue-capped
    minting ensures the attacker always loses money.
```

Free-tier trusted traffic: Trusted peers relay for free — this is unchanged. Free relay is a benefit of the trust network, not a minting mechanism. It was never intended to earn minting rewards.

Channel-funded payments (mechanism 1): Unaffected. When a relay wins the lottery and has a funded channel with the sender, the channel debit happens regardless. Demand-backed minting only affects mechanism 2 (minting proofs).

Revenue-Capped Minting

Even with demand-backed minting eligibility, an attacker could spend MHR on funded channels to generate minting-eligible traffic. Revenue-capped minting ensures this is **always unprofitable**:

```
Revenue-capped minting formula:

  effective_minting(epoch) = min(
      emission_schedule(epoch),                    // halving ceiling
      minting_cap × total_channel_debits(epoch)    // 0.5 × actual relay fees
  )
  minting_cap = 0.5
```

Formal proof of unprofitability:

```
Self-dealing attack:

  1. Attacker spends Y MHR on relay fees (channel debits) for fake traffic
  2. total_channel_debits increases by Y (attacker's contribution)
  3. Maximum possible minting = minting_cap × total_channel_debits
  4. Even if attacker is the ONLY relay and captures 100% of minting:
     attacker receives ≤ 0.5 × total_channel_debits
  5. Attacker's best case: spent Y, received 0.5 × Y = net loss of 0.5 × Y

  With other honest relays in the network, the attacker receives even less
  (their share of 0.5 × total_debits, proportional to their wins).

  The attack is unprofitable at ALL scales — whether Y is 1 MHR or 1 billion MHR.
```

Impact on supply curve:

- Early network (low traffic): actual minting is well below the emission schedule. Supply grows slowly, tracking real economic activity.
- Growing network: actual minting approaches emission ceiling as relay fees increase.
- Mature network: revenue cap is rarely binding (relay fees far exceed the emission schedule). Supply follows the standard halving schedule.

This means supply growth is demand-responsive rather than fixed. Early supply grows slowly (good — prevents speculation), and mature supply follows the emission schedule.

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) of the gossip bandwidth budget even on the most constrained links.

Trusted Peers: Free Relay

Nodes relay traffic for trusted peers 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.

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.

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.

Account State

```
graph TD
    subgraph NodeA["Node A"]
        A_epoch["epoch_bal: 200"]
        A_earned["Δ earned: 300"]
        A_spent["Δ spent: 0"]
        A_balance["balance: 500"]
    end

    subgraph NodeB["Node B"]
        B_epoch["epoch_bal: 200"]
        B_earned["Δ earned: 100"]
        B_spent["Δ spent: 0"]
        B_balance["balance: 300"]
    end

    NodeA --> SR["SettlementRecord<br/>(both signatures)"]
    NodeB --> SR
    SR --> Gossip["Gossiped to network"]
    Gossip --> Merge["Each receiving node<br/>validates & merges<br/><br/>epoch_balance:
frozen<br/>delta GCounters: merge<br/>via pointwise max<br/>(no conflicts ever)"]
```

```
AccountState {
    node_id: NodeID,
    epoch_number: u64,            // which epoch this state is relative to
    epoch_balance: u64,           // frozen balance at last epoch compaction
    delta_earned: GCounter,       // post-epoch earnings (per-node entries, merge =
pointwise max)
    delta_spent: GCounter,        // post-epoch spending (same structure)
    // Balance = epoch_balance + value(delta_earned) - value(delta_spent)
    settlements: GSet<SettlementHash>,  // dedup set (post-epoch only)
}
```

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

Why Separate epoch_balance from Deltas?

The `epoch_balance` is a frozen scalar from the authoritative epoch snapshot. The `delta_earned` and `delta_spent` GCounters track only post-epoch activity using per-node entries. This separation is critical for partition safety — see GCounter Rebase for the full analysis.

Balance is always derived: `balance = epoch_balance + value(delta_earned) - value(delta_spent)`. It is never stored directly.

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 delta_spent / party_b's delta_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.

Gossip Bandwidth

With stochastic relay rewards, 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

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 winning epoch's `epoch_balance` snapshot is adopted and the losing partition's settlements are recovered via settlement proofs (see Partition-Safe Merge Rules). Individual balance correctness is preserved — 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 demand-backed VRF lottery wins this
epoch
                                         // (only wins where packet traversed a funded
channel)
    sample_proofs: Vec<VRFProof>,        // 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.

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 (see GCounter Rebase)
    account_snapshot: Map<NodeID, epoch_balance>,

    // 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 },
}
```

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.

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.

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.

GCounter Rebase

GCounter `delta_earned` and `delta_spent` grow monotonically between epochs. Over very long timescales (centuries), high-throughput nodes could approach the u64 maximum (1.84 × 10^19) due to money velocity: the same tokens are earned, spent, earned again, each cycle growing both delta counters.

Epoch compaction solves this. At each epoch, the snapshot freezes the balance and resets the deltas:

```
GCounter rebase at epoch compaction:

  Before epoch:
    Alice: epoch_balance = 200,000    delta_earned = {Y: 3,000,000, Z: 1,800,000}
    delta_spent = {W: 4,600,000, V: 200,000}
    Balance = 200,000 + 4,800,000 - 4,800,000 = 200,000

  After epoch snapshot (rebased):
    Alice: epoch_balance = 200,000    epoch_number incremented
    delta_earned = {}  (zeroed)
    delta_spent = {}   (zeroed)
    Balance = 200,000 (unchanged)

  Post-epoch settlements apply on top:
    Alice earns 50,000 (processed by node Y) → delta_earned = {Y: 50,000}
    Alice spends 30,000 (processed by node Z) → delta_spent = {Z: 30,000}
    Balance = 200,000 + 50,000 - 30,000 = 220,000 ✓
```

Without rebase, a node processing 10^10 μMHR/epoch of throughput would overflow u64 after ~~1.84 × 10^9 epochs (~~35,000 years). With rebase, delta counters never exceed one epoch's worth of activity — the protocol runs indefinitely.

Partition-Safe Merge Rules

The separation of `epoch_balance` from delta GCounters is critical for correctness during partition merges. When two copies of the same account are merged:

```
CASE 1: Same epoch_number, same epoch_balance (normal operation)
  Standard CRDT merge:
    epoch_balance: unchanged (identical on both sides)
    delta_earned: GCounter pointwise max
    delta_spent:  GCounter pointwise max
    settlements:  GSet union

  This is the common case — both nodes are in the same partition
  or have received the same epoch snapshot.

CASE 2: Same epoch_number, DIFFERENT epoch_balance (concurrent partition compaction)
  Two partitions independently compacted to the same epoch number
  but processed different pre-rebase settlements, producing different
  epoch_balance values.

  Resolution:
    1. The epoch with the higher settlement count wins (existing rule)
    2. Winning epoch's account_snapshot provides epoch_balance for ALL accounts
    3. Winning partition's delta GCounters are kept as-is
    4. Losing partition's delta GCounters are discarded
    5. Losing partition's post-epoch settlements that are NOT in the winning
       epoch's bloom filter are submitted as settlement proofs during the
       verification window, which re-applies them to the delta GCounters
    6. Losing partition's PRE-epoch settlements that are NOT in the winning
       epoch's bloom filter are ALSO submitted as settlement proofs —
       these add the amounts that were absorbed into the losing partition's
       epoch_balance but lost when the winning partition's higher/lower
       epoch_balance was adopted

CASE 3: DIFFERENT epoch_numbers (one partition is ahead)
  The higher epoch_number wins entirely.
    epoch_number: higher value
    epoch_balance: from the higher-epoch partition
    delta_earned: from the higher-epoch partition
    delta_spent:  from the higher-epoch partition
  The lower-epoch partition's settlements are recovered via
  settlement proofs against the winning epoch's bloom filter.
```

Why this is safe: The delta GCounters use per-node entries (each processing node writes only to its own entry). Within a single partition, standard CRDT merge (pointwise max) is always correct. Across partitions with conflicting epochs, the settlement proof mechanism — which checks against the winning epoch's bloom filter, NOT the GSet — recovers any settlements that were lost during epoch_balance adoption. The bloom filter check is critical: a settlement may be in the merged GSet (from the losing partition's contribution) but NOT reflected in the winning epoch's delta GCounters, so the GSet must not be used for dedup during settlement proof processing.

Settlement proof dedup rule: During the verification window, settlement proofs are checked against the **winning epoch's bloom filter only**. The live GSet is NOT consulted. After successful re-application, the settlement hash is added to the GSet to prevent future re-processing during normal (non-verification-window) operation.

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 in epoch_balance, discard
- **Absent from filter**: New settlement, apply to delta GCounters on top of epoch_balance. If within the verification window, submit as a settlement proof.

Important: During the verification window after a partition merge, settlement proofs are checked against the **winning epoch's bloom filter only** — NOT the merged GSet. This is because a settlement may exist in the merged GSet (contributed by the losing partition) but not be reflected in the delta GCounters (because the losing partition's deltas were discarded during conflict resolution). The bloom-filter-only check ensures such settlements are correctly re-applied.

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) / (ln2)² ≈ 19.2 bits at p = 0.0001
  k = -log₂(p) ≈ 13.3, rounded to 13
  For 10,000 settlements: m = 192,000 bits = 24 KB
  For 1M settlements: m = 19.2M bits ≈ 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 || epoch_balance)`, 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 24 bytes per entry (16-byte NodeID + 8-byte epoch_balance), this is ~24 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 × 24 bytes = 1.2 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,
    epoch_balance: u64,
    merkle_siblings: Vec<Blake3Hash>,  // 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_epoch_balance: u64,                     // frozen balance at this epoch
    partner_epoch_balances: Vec<(NodeID, 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.

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.

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.

The Trust Graph

Each node maintains a set of trusted peers:

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

    // 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>,

    // Transitive credit ratio for friends-of-friends (default 0.10)
    transitive_credit_ratio: f32,           // range [0.0, 0.50], default 0.10
    transitive_ratio_overrides: Map<NodeID, f32>,  // per-peer override

    // Self-assigned scopes — geographic and interest (purely informational)
    scopes: Vec<HierarchicalScope>,     // max 1 Geo + up to 3 Topic; total ≤ 1 KB

    // DEPRECATED: use scopes instead. Kept for backward compatibility.
    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. Cost overrides are unidirectional — they apply to outbound traffic pricing from the configuring node only.

How Communities Emerge

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

```
graph LR
    Alice["Alice"] ⟷|trust| Bob["Bob"]
    Bob ⟷|trust| Carol["Carol"]
    Alice ⟷|trust| Dave["Dave"]
    Bob ⟷|trust| Dave
    Carol ⟷|trust| Dave
```

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

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.

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.

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): configurable ratio of direct credit line (default 10%, max
50%)
  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	Configurable % of direct limit (default 10%, max 50%)	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 transitive_credit_ratio × 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
```

Hierarchical Scopes

Nodes self-assign **scopes** — hierarchical namespaces that describe where they are and what they care about. Scopes replace the flat `community_label` with a structured system that supports both place-based communities and interest communities.

```
HierarchicalScope {
    scope_type: enum {
        Geo,    // place hierarchy (physical or virtual)
        Topic,  // interest/community hierarchy
    },
    segments: Vec<String>,    // hierarchical path, max 8 levels, max 32 chars each
}

Scope constraints:
  Per node (TrustConfig):    max 1 Geo + up to 3 Topic (4 total)
  Per content (PostEnvelope): max 1 Geo + up to 3 Topic (4 total)
  Total scope bytes:          ≤ 1,024 bytes (UI shows remaining space)

Rationale:
  - You are physically in one place at a time
  - Voting is geo-scoped — multiple geo scopes would enable double-voting
  - RadioRangeProof verifies one location
  - Interests are multi-dimensional — multiple topics are natural
  - Hierarchical prefix matching means a single deep tag covers multiple query levels
  - 3 topics is expressive enough: most users have 1-2 primary interests
  - 1 KB cap keeps scope data small for constrained devices (ESP32)
```

Scopes are **hierarchical namespaces**, similar to URLs. The `segments` are arbitrary strings — they can describe physical locations, virtual spaces, organizations, or anything else. The `scope_type` signals **propagation intent**, not physicality: `Geo` means "this is a place where members are dense and nearby each other" (whether physically or virtually), while `Topic` means "this is an interest that spans across places."

Geo Scopes

Geo scopes describe **places** — physical locations, virtual spaces, or any community with dense, place-like membership:

```
Geo scope examples:

  Physical locations:                    Virtual places:

  north-america                          cyberspace
  └── us                                 └── guild-wars
      ├── oregon                         │   ├── server-42
      │   ├── portland                   │   └── server-7
      │   │   ├── hawthorne ◄── Alice    │
      │   │   └── pearl     ◄── Bob     └── discord
      │   ├── eugene                         └── mehr-dev ◄── Dave
      │   └── bend
      └── california                     organizations
          └── ...                        └── university
                                             └── mit
  asia                                       └── csail ◄── Eve
```

```
    └── iran
        └── tehran
            └── district-6   ←── Carol
```

```
Physical:  Geo("north-america", "us", "oregon", "portland", "hawthorne")
Virtual:   Geo("cyberspace", "guild-wars", "server-42")
Org:       Geo("organizations", "university", "mit", "csail")
```

The hierarchy is **bottom-up** — neighbors form the base, cities emerge from connected neighborhoods, regions from connected cities. This applies equally to physical and virtual places: you join a game server first, then discover the broader game community, then the gaming umbrella. The pattern is the same — local connections aggregate into larger structures.

Interest Scopes

Interest scopes describe communities of shared interest that span geography:

```
Dave sets:   Topic("gaming", "pokemon", "competitive")
Eve sets:    Topic("science", "physics", "quantum")
Frank sets:  Topic("music", "jazz")
```

Interest communities are **sparse** — not everyone in Portland cares about Pokemon. A Pokemon community might span Portland, Tokyo, and Berlin with nothing in between. This is the opposite of geographic scopes, which are **dense** (most people are physically somewhere).

Scope Matching

Subscriptions and queries can match at any level of the hierarchy:

Pattern	Matches
`Geo("north-america", "us", "oregon", "portland")` exact	Only Portland
`Geo("north-america", "us", "oregon")` prefix	Portland, Eugene, Bend, and everything in Oregon
`Geo("north-america", "us")` prefix	All US scopes
`Topic("gaming")` prefix	Pokemon, Minecraft, and all gaming sub-topics
`Topic("gaming", "pokemon")` exact	Only Pokemon, not gaming broadly

Properties

Scopes retain all the properties of `community_label`:

- **Self-assigned** — no one approves your scope claims, no authority enforces them
- **Not authoritative** — scopes carry no protocol-level privileges (cannot grant access, waive fees, or modify trust)
- **Not unique** — multiple disjoint clusters can use the same scope
- **Free-form strings** — all segments are arbitrary strings. Communities converge on naming through social consensus (e.g., "portland" not "pdx"), the same way they do today. No ISO codes, no standardized taxonomy, no gatekeeping.
- **Used by services** — MHR-Name scopes names by geographic scope, MHR-Pub supports `Scope(match)` subscriptions, MHR-DHT uses scopes for content propagation boundaries, and the Social layer uses scopes for geographic and interest feeds

Geo Scope Verification

Geo scopes can optionally be **verified** — see Identity Claims for the full verification protocol. Verification methods vary by the kind of place:

Place Type	Verification Method	Precision
Physical neighborhood	RadioRangeProof — if you can hear a node's LoRa beacon, you're within physical range	~1–15 km
Physical city/region	Bottom-up aggregation of verified neighborhood claims	Aggregated
Virtual space	Application-specific (e.g., server-signed attestation, invite-chain, admin vouch)	Varies
Organization	Peer attestation from existing verified members	Social

Interest scopes are **never verified** — anyone can declare interest in Pokemon. Verification matters for geo scopes that carry governance weight (see Voting) — a node cannot vote on Portland issues without a verified Portland-area presence claim. Unverified geo scopes are still useful for content routing and feed subscriptions; they just don't carry voting rights.

Wire Format

Designed for constrained devices:

Field	Size	Description
`scope_type`	1 byte	0 = Geo, 1 = Topic
`segment_count`	1 byte	Number of path segments (max 8)
`segments`	variable	Length-prefixed UTF-8 (1-byte length + content per segment)

Maximum size per scope: 2 + 8 × 33 = 266 bytes. Total scope data is capped at **1,024 bytes** (1 KB). With max 4 scopes (1 Geo + 3 Topic), typical usage is well under the cap — e.g., `Geo("us", "oregon", "portland")` (15 bytes) + `Topic("gaming", "pokemon")` (18 bytes) + `Topic("music", "jazz")` (16 bytes) = 49 bytes. The cap only binds when using deep hierarchies (8 segments × 32 chars) across multiple scopes.

Backward Compatibility

The `community_label` field is retained for backward compatibility. New nodes populate both:

```
Migration:
  community_label: "portland-mesh"
    → scopes: [Geo("portland")]

  Old nodes: read community_label, ignore scopes
  New nodes: read scopes, fall back to community_label if scopes is empty
```

Geo vs. Topic: Two Dimensions of Community

	Geo Scopes (Places)	Topic Scopes (Interests)
Density	Dense — members are nearby each other	Sparse — members are scattered
Propagation	Bottom-up through proximity	Wide, across places
Verification	RadioRangeProof (physical), attestation (virtual), or none	None needed (self-declared)
Content cost	Cheap locally, expensive globally	Depends on relay distance
Voting	Enables scoped voting (if verified)	No voting implications
Physical examples	`Geo("north-america", "us", "oregon", "portland")`	`Topic("gaming", "pokemon")`
Virtual examples	`Geo("cyberspace", "guild-wars", "server-42")`	`Topic("science", "physics")`
Multiplicity	One per node/content	Up to 3 (within 1 KB total)

A single post can have **one** geographic scope and **multiple** interest scopes. A post tagged `Geo("portland") + Topic("gaming", "pokemon")` appears in both the Portland local feed and the global Pokemon feed. Intersection queries ("Portland Pokemon") are resolved client-side by filtering on both scopes.

Comparison: Explicit Zones vs. Trust Neighborhoods

Aspect	Explicit Zones	Trust Neighborhoods
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@geo:portland` (hierarchical scopes)
Sybil resistance	Admission policy	Trust is social and economic (you absorb their debts)
UX complexity	Create, join, configure	Add contacts

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 by configurable ratio (default 10%, max 50%) 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.

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

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.

The Apartment Building Scenario

Consider a typical apartment building in Denmark:

```
graph TD
    subgraph current["Current Model"]
        ISP1["ISPs"]
        A1["Apt 1\n200 kr/month"]
        A2["Apt 2\n200 kr/month"]
        A3["..."]
        A50["Apt 50\n200 kr/month"]
        ISP1 -->|"own router,\nown subscription"| A1
        ISP1 --> A2
        ISP1 --> A3
        ISP1 --> A50
        TOTAL1["50 × 200 kr = 10,000 kr/month\nUtilization per connection: < 5%"]
    end

    subgraph mehr["Mehr Model"]
        ISP2["ISPs"]
        GW1["Gateway Node 1\n200 kr/month"]
        GW2["Gateway Node 2\n200 kr/month"]
        R1["Apt 1\npays MHR"]
        R2["Apt 2\npays MHR"]
        R3["..."]
        R47["Apt 47\npays MHR"]
        ISP2 --> GW1
        ISP2 --> GW2
        GW1 -->|"WiFi mesh"| R1
        GW1 --> R2
        GW2 --> R3
        GW2 --> R47
        TOTAL2["2-3 gateways = 400-600 kr/month\nISP revenue drops ~94%"]
    end
```

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.

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

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.

How You Earn on Mehr

Every node earns proportionally to the value it provides:

Relay Earnings

The simplest way to earn. Any node that forwards packets for non-trusted traffic participates in the stochastic relay lottery. 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
```

Storage Earnings

Nodes with disk space earn by storing data for the network via MHR-Store:

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

Compute Earnings

Nodes with CPUs or GPUs earn by executing contracts and offering inference via 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

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),
    },
}
```

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.

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

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. No central coordination required.

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.

Discovery Rings

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.

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 (log₂-encoded μMHR/byte)
    avg_cost: u16,      // average cost across providers (log₂-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 log₂ formula:
  encoded = round(16 × log₂(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.

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.

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.

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.

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.

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.

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

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.

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.

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<ChannelState>,  // 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

Capability Agreements

When a requester finds a suitable provider through 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`

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),
}
```

Key Properties

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

Payment-Linked

Every agreement references an existing payment channel between the two parties. Payment flows automatically as the service is delivered.

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 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
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"

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<u8>>,     // 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, trusted peers often offer discounted or free services.

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.

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.

Note: Delivery receipts prove that packets were delivered, not that the traffic represents legitimate demand. A Sybil attacker can fabricate traffic between colluding nodes and produce valid delivery receipts. The economic defense against this is demand-backed minting — VRF wins only count for minting if the packet traversed a funded payment channel, and revenue-capped minting ensures self-dealing is always unprofitable.

Storage Verification

Method: Merkle-proof challenge-response (see MHR-Store 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 — the consumer reconstructs the lost shard from erasure-coded replicas and stores it on a replacement node.

Compute Verification

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

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.

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.

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.

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

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").

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.

Data Objects

```
DataObject {
    hash: Blake3Hash,                // content hash = address
    content_type: enum { Immutable, Mutable, Ephemeral },
    owner: Option<NodeID>,           // for mutable objects
    created: Timestamp,
    ttl: Option<Duration>,           // 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)
    },
}
```

Content Types

Immutable

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

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)
```

Ephemeral

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

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<ShardInfo>,// if part of an erasure-coded set

    // Revenue sharing — for content that earns from reader access
    kickback_recipient: Option<NodeID>,  // original author (receives share of retrieval
fees)
    kickback_rate: u8,                   // 0-255: author's share of retrieval fee
(rate/255)
```

```
    signatures: (Sig_Provider, Sig_Consumer),
}
```

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.

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.

Free Storage Between Trusted Peers

Just like relay traffic, 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.

Revenue Sharing (Kickback)

When content is published for public consumption (social posts, media, curated feeds), the **original author** can earn a share of retrieval fees through the kickback mechanism.

How It Works

```
Author publishes post → creates StorageAgreement with kickback fields:
    kickback_recipient = Author's NodeID
    kickback_rate = 128  (roughly 50% of retrieval fees go to author)

Reader pays storage node to retrieve post:
    Retrieval fee: 100 μMHR
    Storage node keeps: 100 × (255 - 128) / 255 ≈ 50 μMHR
    Storage node forwards: 100 × 128 / 255 ≈ 50 μMHR → Author

Settlement: via the existing payment channel between storage node and author
```

Incentive Alignment

- **Storage nodes** are incentivized to host popular content because they earn the non-kickback portion of every retrieval fee
- **Authors** are incentivized to create content people want to read because they earn kickback from every reader
- **Readers** pay the same retrieval fee regardless — the kickback split is between author and storage node
- **Curators** who reference posts in curated feeds drive traffic to original authors, earning kickback on their own curation feed while generating kickback for the original authors too

Kickback Rate

The `kickback_rate` field is a `u8` (0–255):

Rate	Author Share	Storage Node Share	Typical Use
0	0%	100%	No kickback (pure storage)
64	~25%	~75%	Low author share, high storage incentive
128	~50%	~50%	Balanced split
192	~75%	~25%	High author share
255	100%	0%	Maximum author share (storage node earns only per-epoch fee)

If `kickback_recipient` is `None`, there is no kickback — the storage node keeps all retrieval fees. This is the default for non-social content (private data, infrastructure objects, etc.).

Self-Funding Content

When kickback revenue exceeds storage cost, content becomes **self-funding**. The author can reinvest kickback to extend storage agreements, or an automated RepairAgent can do it:

```
Self-funding threshold:
    If kickback_per_epoch > cost_per_epoch:
        Content is self-sustaining - it lives as long as people read it
    If kickback_per_epoch < cost_per_epoch:
        Author must subsidize - content expires if author stops paying
```

Popular content that crosses the self-funding threshold climbs the propagation hierarchy automatically. Unpopular content stays local or expires. No algorithm decides what lives and what dies — economics does.

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.

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
```

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.

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.

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
```

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.

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
```

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.

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.

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
```

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 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.

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.

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.

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

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<u32> }
    Provider responds with only the requested chunks, avoiding
    retransmission of already-received data.
```

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

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 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.

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.

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.

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
```

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

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.

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, 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:

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.

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.

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

Trusted Relay Shortcut

If the client's relay is in its trust graph, 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.

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

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.

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 (deprecated)
        Scope(ScopeMatch),      // any publication matching a hierarchical scope
    },
    delivery: enum {
        Push,                   // immediate, full payload
        Digest,                 // batched summaries, periodic
        PullHint,               // hash-only notification
    },
}

ScopeMatch {
    scope: HierarchicalScope,   // from trust-neighborhoods
    match_mode: enum {
        Exact,                  // this scope level only
        Prefix,                 // this scope and all children
    },
}
```

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 (deprecated — use Scope)
Scope	Watch all activity matching a hierarchical scope — geographic or interest

Delivery Modes

Push

Full payload delivered immediately when published. Best for high-bandwidth links where real-time updates matter.

Use on: WiFi, Ethernet, Cellular

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

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

Envelope-Aware Delivery

For Social content, MHR-Pub notifications carry the PostEnvelope — the free preview layer — rather than the full post:

Delivery Mode	What's Delivered	Reader Cost
Push	Full PostEnvelope (~300-500 bytes)	Free
Digest	Batched envelopes (headline + hash per item)	Free
PullHint	Post hash only (32 bytes)	Free (envelope fetched on demand)

In all modes, the reader browses envelopes at zero cost. Fetching the full SocialPost content is a separate, paid retrieval from the storage node. This separation means even LoRa nodes on PullHint subscriptions can browse headlines and decide what's worth fetching over a higher-bandwidth link later.

Scope Subscriptions

Scope subscriptions are the primary mechanism for geographic feeds, interest feeds, and community content discovery. They build on Hierarchical Scopes to enable structured content routing.

Geographic Feeds

Subscribe to content from a geographic area at any level of the hierarchy:

```
// All posts from Portland
subscribe(Scope(Geo("north-america", "us", "oregon", "portland"), Exact), Push);

// All posts from anywhere in Oregon
subscribe(Scope(Geo("north-america", "us", "oregon"), Prefix), Digest);
```

Geographic scope subscriptions naturally bias toward local content — nearby nodes have cheaper relay paths and higher cache density, so local content arrives faster and costs less.

Interest Feeds

Subscribe to content by topic at any level of the hierarchy:

```
// All Pokemon content globally
subscribe(Scope(Topic("gaming", "pokemon"), Prefix), Digest);

// Only competitive Pokemon
subscribe(Scope(Topic("gaming", "pokemon", "competitive"), Exact), Push);
```

Interest subscriptions are **sparse** — they span geography. A Pokemon subscription connects Portland, Tokyo, and Berlin through interest relay nodes that bridge geographic clusters.

Intersection Feeds

A client can subscribe to both a geographic and interest scope simultaneously and filter locally for the intersection:

```
// Subscribe to Portland content AND Pokemon content
subscribe(Scope(Geo("north-america", "us", "oregon", "portland"), Exact), Push);
subscribe(Scope(Topic("gaming", "pokemon"), Prefix), Push);

// Client-side: show posts that appear in BOTH feeds
// Result: Portland Pokemon community
```

Intersection is always client-side — the protocol delivers by individual scope, and the application composes.

Scope Routing

When a node publishes with scopes, notifications propagate to subscribers at each level:

```
Post published with scopes:
  Geo("north-america", "us", "oregon", "portland", "hawthorne")
  Topic("gaming", "pokemon")

Notifications delivered to:
  Scope(Geo("...", "hawthorne"), Exact)     ✓  exact match
  Scope(Geo("...", "portland"), Prefix)      ✓  portland covers hawthorne
  Scope(Geo("...", "oregon"), Prefix)        ✓  oregon covers portland
  Scope(Topic("gaming", "pokemon"), Exact)   ✓  exact match
  Scope(Topic("gaming"), Prefix)             ✓  gaming covers pokemon
  Scope(Geo("...", "seattle"), Exact)        ✗  wrong city
  Scope(Topic("science"), Prefix)            ✗  wrong topic
```

Delivery Mode by Scope Level

Applications should select delivery mode based on both link quality and scope breadth:

Scope Level	Typical Volume	Recommended Default
Neighborhood	Low	Push
City	Moderate	Push or Digest
Region	High	Digest
Country/Global	Very high	PullHint
Narrow interest topic	Low-moderate	Push
Broad interest topic	High	Digest

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<u8>,              // 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 possible.

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.

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
```

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.

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.

Compute Delegation

If a node can't execute a contract locally, it delegates to a capable neighbor via the capability marketplace:

```
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.

Opaque Compute: Hardware-Accelerated Services

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. The pattern is **opaque compute**: input goes in, output comes out. The protocol does not sandbox, inspect, or guarantee the compute method — the node can use GPU, NPU, FPGA, or any hardware.

```
A GPU/NPU 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. Trust comes from reputation, not verification of the compute method.

Hardware examples:

Accelerator Type	Examples
GPU	NVIDIA RTX series, AMD Radeon
NPU	Apple Neural Engine, Qualcomm Hexagon, MediaTek APU
FPGA	Xilinx, Intel/Altera
TPU	Google Edge TPU

Result Verification for Opaque Compute

Since opaque compute provides no built-in execution guarantee, consumers choose a verification strategy based on their trust requirements and budget:

Strategy	How It Works	Cost	Trust Level
1. Reputation (default)	Node builds reputation through consistent outputs. Bad outputs → trust removal → lost income stream.	None (built into trust system)	Moderate
2. Redundant execution	Client sends same input to 2–3 nodes. Majority agreement = accepted result.	2–3x compute fees	High
3. Spot-checking	Client occasionally sends inputs with known outputs. Wrong answer → node flagged, agreement terminated.	~5% overhead	Moderate–High
4. Cryptographic verification (future)	ZK proofs of correct inference (active research area). Not practical for large models today.	TBD	Highest

Verification is a **consumer-side choice**, not protocol enforcement. Most consumers rely on reputation (the default). High-value or adversarial workloads use redundant execution or spot-checking.

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

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.

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
    },
}
```

Tier 0: No Privacy (Default)

The compute node receives plaintext input, executes, and returns the result. Verification is via result hash 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.

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.

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.

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.

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
Delivery notifications	MHR-Pub
Transport encryption	Link-layer encryption (Reticulum-derived)
End-to-end encryption	E2E encryption

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

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.

Group Messaging

Group messages use shared symmetric keys managed by an MHR-Compute contract:

```
GroupState {
    group_id: Blake3Hash,
    members: Set<NodeID>,
    current_key: ChaCha20Key,        // current group symmetric key
    key_epoch: u64,                   // increments on every rotation
    admin: NodeID,                    // creator; can add/remove members
    co_admins: Vec<CoAdminCertificate>,  // 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)
}
```

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)

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

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.

Social

A decentralized content distribution network built on Mehr primitives. No central servers, no algorithmic recommendations, no ads. Authors pay to publish — skin in the game. Readers pay bandwidth to access — infrastructure sustains itself. Popular content self-funds and propagates wider. Unpopular content expires. Economics replaces algorithms.

While "social" implies short text posts, the same architecture handles **any content type**: music albums, scientific papers, video courses, games, software, journalism, podcasts — anything that can be stored as a DataObject. The envelope/post split, kickback economics, and propagation rules are content-agnostic.

Content Type	Envelope Shows	Paid Content	Typical Kickback
Text post	Full text (under 280 chars)	Full text + links	Low (cheap to read)
Photo essay	Summary + blurhash thumbnails	Full-resolution images	Moderate
Music album	Track listing + artist + duration	Audio files	High (large files)
Video course	Lesson titles + descriptions	Video files	High
Scientific paper	Title + abstract + authors	Full PDF	Moderate
Game / software	Name + description + screenshots	Binary + assets	High
Podcast episode	Title + show notes	Audio file	Moderate
Curated collection	Curator notes per item	References to originals	Curator earns on collection; authors earn on items

Architecture

Every publication on Mehr has two layers: a **free envelope** that propagates everywhere (browsable at zero cost), and **paid content** that requires retrieval fees. Users browse envelopes to decide what's worth accessing, then pay only for content they actually want.

```
graph TD
    subgraph FREE["FREE LAYER (PostEnvelope)<br/>Propagates via MHR-Pub to all scope
subscribers<br/>~300-500 bytes — fits in a single LoRa frame"]
        Headline
        Summary
        Blurhash
        Scopes["Scopes, metadata"]
    end

    subgraph PAID["PAID LAYER (SocialPost)<br/>Fetched on demand — reader pays relay
fees<br/>Size proportional to content"]
        FullText["Full text"]
        Media["Media (images, video, audio)"]
        Links
    end

    Headline -->|post_id| FullText
    Summary -->|post_id| FullText
    Blurhash -->|post_id| FullText
    Scopes -->|post_id| FullText

    FullText -->|Kickback| Author
    Media -->|Kickback| Author
    Links -->|Kickback| Author
```

PostEnvelope (Free Layer)

The envelope is a lightweight, separate DataObject that propagates freely across the mesh. It contains everything a reader needs to decide whether to fetch the full post:

```
PostEnvelope {
    post_id: Option<Blake3Hash>,            // stable ID of the SocialPost (None for boost-
only envelopes)
    author: NodeID,
    headline: Option<String>,               // title (~100 chars, author-set)
    summary: Option<String>,                // author-written preview (None for boosts —
use the original's)
    media_hints: Vec<MediaHint>,            // lightweight descriptions of attachments
    scopes: Vec<HierarchicalScope>,         // max 1 Geo + up to 3 Topic; total ≤ 1 KB
    reply_to: Option<Blake3Hash>,           // post_id of parent (threading)
    boost_of: Option<Blake3Hash>,           // post_id of boosted post
    references: Vec<Blake3Hash>,            // post_ids of related posts (bidirectional
content graph)
    content_size: u32,                      // full post size in bytes (0 for boost-only)
    created: Timestamp,
    sequence: u64,                          // monotonic version counter (incremented on
edit)
    kickback_rate: u8,                      // author's desired share of retrieval fees (0-
255)
    signature: Ed25519Sig,                  // signed by author (proves authenticity)
}

MediaHint {
    content_type: String,                   // "image/jpeg", "video/mp4", etc.
    size: u32,                              // bytes
    blurhash: Option<String>,               // visual placeholder (~30 bytes)
    alt_text: Option<String>,               // accessibility description
}
```

Envelope size: ~300–500 bytes. Fits in a single LoRa frame with `min_bandwidth: 0`. A reader on a LoRa-only node can browse headlines, summaries, and blurhash thumbnails without paying anything.

SocialPost (Paid Layer)

The full post is a mutable DataObject containing the actual content. Fetching it costs retrieval fees:

```
SocialPost {
    post_id: Blake3Hash,                    // stable ID: Blake3(author ‖ created ‖ nonce)
    author: NodeID,
    content: PostContent {
        text: Option<String>,               // full post body (UTF-8)
        media: Vec<Blake3Hash>,             // references to media DataObjects
        links: Vec<String>,                 // external URLs (for internet-connected nodes)
    },
    sequence: u64,                          // monotonic version counter (0 on first
publish)
    edited: Option<Timestamp>,              // None on first publish, Some on edits
    signature: Ed25519Sig,
}
```

The SocialPost is lean — scopes, timestamps, and metadata live on the envelope. The post contains only the content that costs money to retrieve. Both the envelope and the post are mutable DataObjects addressed by `(author, post_id)`. The `post_id` is a stable identifier generated at creation time (`Blake3(author ‖ created ‖ nonce)`) that never changes, even when the content is edited.

Profile

A mutable DataObject containing identity information:

```
UserProfile {
    node_id: NodeID,
    display_name: String,
    bio: Option<String>,
    avatar: Option<Blake3Hash>,             // reference to image DataObject
    scopes: Vec<HierarchicalScope>,         // from TrustConfig
    claims: Vec<Blake3Hash>,                // references to IdentityClaims
    sequence: u64,                          // monotonic version counter
    signature: Ed25519Sig,
}
```

Feed Types

Mehr social supports five feed types. All feeds are assembled **locally** — no server decides what you see.

```
graph TD
    Follow["1. FOLLOW<br/>Specific users you choose<br/>Node(alice)"]
    Geographic["2. GEOGRAPHIC<br/>Content from a place:<br/>neighborhood → city →
region"]
    Interest["3. INTEREST<br/>Content by topic:<br/>pokemon, physics, jazz"]
    Intersection["4. INTERSECTION<br/>Client-side filter on BOTH:<br/>'Portland Pokemon'
= geo ∩ topic"]
    Curated["5. CURATED<br/>Human editor selects best content<br/>Readers subscribe to
curator's feed<br/>Two kickback flows: curator + original author"]

    Geographic --> Intersection
    Interest --> Intersection
```

1. Direct Follow Feed

Follow specific users. Unchanged from the basic MHR-Pub model.

```
// Follow Alice — see everything she posts
subscribe(Node(alice_node_id), Push);
```

This is the foundation. You follow people you trust, and you see their posts in reverse-chronological order.

2. Geographic Feed

Subscribe to content from a geographic area at any level of the scope hierarchy:

```
// Everything from my neighborhood
subscribe(Scope(Geo("north-america", "us", "oregon", "portland", "hawthorne"), Exact),
Push);

// Everything from Portland (all neighborhoods)
subscribe(Scope(Geo("north-america", "us", "oregon", "portland"), Prefix), Digest);

// Everything from Oregon
subscribe(Scope(Geo("north-america", "us", "oregon"), Prefix), PullHint);
```

Geographic feeds are the **local newspaper** — events, news, discussions relevant to where you physically are. Content is cheapest and fastest at the neighborhood level, progressively more expensive at higher scopes.

3. Interest Feed

Subscribe to content by topic, independent of geography:

```
// All Pokemon content globally
subscribe(Scope(Topic("gaming", "pokemon"), Prefix), Digest);

// Only competitive Pokemon
subscribe(Scope(Topic("gaming", "pokemon", "competitive"), Exact), Push);

// All science content
subscribe(Scope(Topic("science"), Prefix), PullHint);
```

Interest feeds are **sparse** — they connect people across geography. A Pokemon feed connects Portland, Tokyo, and Berlin. Content propagates through interest relay nodes that bridge geographic clusters, but only after local validation — the author's local community must engage with the content (boost, retrieve, or curate) before interest relays forward it globally.

4. Intersection Feed

Combine geographic and interest scopes client-side:

```
// Subscribe to Portland AND Pokemon
subscribe(Scope(Geo("...", "portland"), Exact), Push);
subscribe(Scope(Topic("gaming", "pokemon"), Prefix), Push);

// Client-side: show only posts that appear in BOTH feeds
// Result: Portland Pokemon community
```

The protocol delivers by individual scope. The application composes intersection feeds locally by filtering posts that match multiple subscriptions. This keeps the protocol simple while enabling powerful queries.

5. Curated Feed

Follow a **curator** — a human who selects the best content from a broader scope:

```
CuratedFeed {
    curator: NodeID,
    name: String,                    // "Portland's Best", "Quantum Physics Weekly"
    description: String,
    entries: Vec<CuratedEntry>,      // max 256 entries per feed page
    scope: HierarchicalScope,        // what this feed covers
    updated: Timestamp,
    sequence: u64,                   // monotonic version counter
    kickback_rate: u8,               // curator's share of retrieval fees
    signature: Ed25519Sig,
}

CuratedEntry {
    post_id: Blake3Hash,                    // stable ID of original post
    added: Timestamp,
    note: Option<String>,                   // curator's commentary
}
```

A single CuratedFeed holds at most **256 entries**. For larger archives, the curator publishes multiple feed pages as separate DataObjects, each covering a time period or sub-topic. This keeps individual feed objects small enough for constrained devices to fetch and parse.

How curation works:

1. Alice follows 200 people and reads the Portland geographic feed daily
2. She publishes a `CuratedFeed` selecting the best posts — "Portland Daily Digest"
3. Bob subscribes to Alice's curated feed instead of following 200 people
4. Bob fetches the CuratedFeed DataObject — Alice earns kickback on it (the curation list has its own `kickback_rate`)
5. Bob's client fetches the **PostEnvelopes** for each curated entry — free, showing headlines + summaries + Alice's curator notes
6. Bob taps posts that interest him — fetches the full SocialPost, and the **original author** earns kickback (the post has its own `kickback_rate`)

These are **two independent kickback flows** on two different DataObjects — the curator's rate and the author's rate don't interact. Bob pays once per DataObject he retrieves, and each DataObject's kickback goes to its respective creator.

The browsing experience: Bob sees Alice's curator notes ("Must-read thread on the new bike lanes") alongside each post's envelope (headline, summary, blurhash). He can scroll the entire curated feed for nearly free — only paying when he opens a full post. This makes curated feeds the most bandwidth-efficient way to discover content.

The curation hierarchy:

```
graph TD
    Producers["Producers (authors)<br/>Create content, earn kickback from readers"]
    Curators["Curators (human editors)<br/>Select best content, earn from subscribers"]
    MetaCurators["Meta-curators (curators of curators)<br/>Select best curators, earn
from subscribers"]
    Readers["Readers<br/>Pay bandwidth, choose what to subscribe to"]

    Producers --> Curators
    Curators --> MetaCurators
    MetaCurators --> Readers
```

Every level has skin in the game:

- Producers pay to post (anti-spam)
- Curators pay to store their curated feed (anti-spam for curation)
- Readers pay bandwidth (infrastructure sustains itself)
- Kickback flows backward at every level

Publishing Flow

When an author creates a post, two mutable DataObjects are published:

```
1. Author writes post content + sets headline/summary
2. Client generates a stable post_id: Blake3(author ‖ created ‖ random_nonce)
3. Client creates SocialPost (paid layer):
     → stored as mutable DataObject keyed by (author, post_id)
     → sequence: 0 (first version)
     → only fetched when a reader requests the full content
4. Client creates PostEnvelope (free layer) with same post_id:
     → stored as mutable DataObject with min_bandwidth: 0
     → sequence: 0 (first version)
     → propagates via MHR-Pub to scope subscribers
     → no storage agreement needed within trust network
```

The `post_id` is a stable identifier that never changes — it's the address of both the envelope and the post across all edits. The envelope costs almost nothing to store and propagate (under 500 bytes). The full post costs proportional to its size. Authors pay for content storage, not for letting people know the content exists.

Editing Posts

Authors can edit their posts by publishing new versions of both the SocialPost and PostEnvelope:

```
Editing flow:
  1. Author modifies content (and optionally headline/summary)
  2. Client publishes updated SocialPost:
       → same post_id, same (author, post_id) key
       → sequence: previous + 1
       → edited: current timestamp
  3. Client publishes updated PostEnvelope (if headline/summary changed):
       → same post_id, same key
       → sequence: previous + 1
  4. MHR-Store propagates the update (highest sequence wins)
  5. MHR-Pub notifies scope subscribers of the updated envelope
```

Edit properties:

- **Version history is not preserved** by default. Mutable DataObject semantics: the highest sequence number replaces the previous version. Storage nodes only keep the latest version.
- **Replies, boosts, and references are stable.** They reference the `post_id`, not the content. An edited post doesn't break its reply chains or reference graph.
- **Clients can show edit status.** The `edited` timestamp on SocialPost tells readers the post was modified. Clients may display "edited" alongside the post.
- **No edit limit.** Authors can edit as many times as they want. Each edit increments `sequence` and costs a storage update (negligible).
- **Boosts of edited posts** reflect the latest version. A reader fetching a boosted post always gets the current content.

Content Economics

```
graph LR
    Author
    Storage["Storage Node"]
    Reader

    Author -->|"1. Pay storage"| Storage
    Storage -->|"2. Envelope propagates (free)"| Reader
    Reader -->|"3. Browse headlines,<br/>summaries (free)"| Reader
    Reader -->|"4. Fetch full post (paid)<br/>relay fees"| Storage
    Storage -->|"5. Kickback"| Author

    style Author fill:#f9f,stroke:#333
    style Reader fill:#bbf,stroke:#333
    style Storage fill:#bfb,stroke:#333
```

Browse Before You Pay

Envelopes propagate freely through MHR-Pub notifications. Readers browse without spending:

```
Reader's feed experience:
  1. Receive PostEnvelopes via MHR-Pub subscription (free)
  2. Browse headlines, summaries, blurhash thumbnails (free)
  3. See content_size and estimated retrieval cost (free)
  4. Tap to fetch full SocialPost + media (paid)

LoRa-only node:
  → Sees all envelopes (text headlines + blurhash)
  → Cannot fetch large media (min_bandwidth too low)
  → Can still fetch text-only posts (small enough for LoRa)

WiFi node:
  → Sees all envelopes
```

```
    → Fetches full posts + images on demand
    → Fetches video only on high-bandwidth links
```

This solves the "pay before you know what you're getting" problem. The envelope is the shopfront window; the full post is what you pay for inside.

Author Pays to Post

Every post is a DataObject that requires a storage agreement. Posting costs money:

```
Cost to post (neighborhood scope):
    Text-only (~200 bytes):     ~1-5 μMHR per epoch
    With image (~50 KB):        ~50-100 μMHR per epoch
    With video (~5 MB):         ~500-1000 μMHR per epoch

Within trust network: free (no agreement, no payment)
```

This is the **anti-spam mechanism**. Every post costs tokens. Spam is economically irrational because you're paying for content nobody will read. No ML moderation needed — posting costs money, and money is limited.

Readers Pay for Bandwidth

When someone outside your trust network retrieves your post, they pay relay fees:

```
Reader cost:
    Within trust network:       free
    Cross-trust retrieval:      ~5 μMHR per packet per hop
    Typical 5-hop path:         ~25 μMHR per retrieval

Total reader cost for a text post: ~25-75 μMHR
Total reader cost for an image post: ~250-750 μMHR
```

Kickback to Author

When readers pay to retrieve content, a portion flows back to the author via the kickback mechanism:

```
Reader pays 100 μMHR to storage node
    → Storage node keeps (255 - kickback_rate) / 255 of retrieval fee
    → Storage node forwards kickback_rate / 255 to author

At default kickback_rate of 128 (~50%):
    → Storage node keeps ~50 μMHR
    → Author receives ~50 μMHR
```

Self-Funding Content

Popular content earns more kickback than it costs to store. It becomes **self-sustaining** and propagates upward through the scope hierarchy automatically:

Popularity	Scope	Funding Model
Unread	Neighborhood	Author pays entirely
Local interest	City	Kickback offsets some storage cost
Regional hit	Region	Kickback exceeds storage cost (self-funding)
Viral	Country/Global	Self-funding at all levels

Content that nobody reads expires when the author stops paying. Content that everyone reads lives forever, funded by its own readership. No platform decides — economics decides.

Why No Likes or Upvotes

Mehr has no reactions, likes, upvotes, or downvotes. This is deliberate.

The problem with reactions: Any free reaction can be Sybil-farmed (create 50 accounts, like your own post 50 times). Any paid reaction where the money returns to the sender can be self-tipped in a loop (post → tip own post → kickback returns the tip). Even burn-to-signal (destroy tokens to upvote) creates a vanity metric that people optimize for — the same engagement treadmill that centralized platforms run on.

Mehr's quality signals are economic, not social:

Signal	What It Means	Sybil-Resistant?
Retrieval count	Real people paid real tokens to read this	Yes — each retrieval costs the reader relay fees that don't return to the author
Self-funding threshold	Kickback exceeds storage cost — readership sustains the content	Yes — requires many distinct paying readers
Scope promotion	Content bubbled up from neighborhood to city/region	Yes — driven by retrieval demand from geographically distributed nodes
Curator inclusion	A human with skin in the game selected this as worth reading	Yes — curator's reputation is on the line
Boost count	Multiple people amplified this to their followers	Partially — boosts are free envelopes, but boosting Sybil content wastes your followers' attention

An author self-retrieving their own post pays relay fees that don't come back via kickback (the storage node keeps its share, and the author is both payer and kickback recipient — net loss). A

Sybil cluster retrieving each other's posts bleeds tokens on relay fees with no external demand to sustain it. The economics make fake popularity expensive.

What readers see instead of like counts: retrieval-driven propagation scope (neighborhood → city → region tells you how widely read something is), curator endorsements, and boost count from people in their trust graph.

Clickbait Resistance

The envelope's `headline` and `summary` are author-written and free-form. An author can write a misleading preview. The protocol does not attempt to enforce summary accuracy — there is no way to mechanically verify that a summary "fairly represents" an image, video, or audio post. Instead, clickbait is handled by the same economic and social systems that handle everything else:

Why clickbait is less profitable on Mehr than on ad-supported platforms:

	Ad-Supported Platform	Mehr
Cost to post	Free	Author pays storage
Revenue model	Infinite recurring ad impressions	One-time retrieval fee per reader
Amplification	Algorithm promotes high-engagement content	No algorithm — only boosts and curators
Second wave	Algorithm keeps serving it to new victims	Zero boosts + zero curation = no second wave
Reader cost	Free (attention only)	μMHR per retrieval — small but nonzero

Defense layers:

1. **Local-first propagation** — Content doesn't spread globally on publication. Geographic content starts at neighborhood scope and promotes upward only when retrieval demand justifies it. Interest content starts in the author's local cluster and only propagates to distant clusters when locally validated — at least one boost, multiple distinct retrievals, or curator inclusion. A garbage post tagged with a popular topic never leaves the author's neighborhood because nobody locally endorses it.

2. **Curators are the quality filter** — Raw geographic and interest feeds are unfiltered. Curated feeds are human-verified. For expensive content (video, large media), readers naturally prefer curator-endorsed content over unknown authors. A curator who includes clickbait loses subscribers and kickback revenue.

3. **Client-side author reputation** — After fetching from an author and finding garbage, the client locally downgrades that author. Future envelopes from flagged authors can be marked with a

warning or hidden entirely. This is purely local — no protocol change, no centralized blocklist.

4. **Economic self-correction** — Clickbait earns from the first wave of disappointed readers. But with zero boosts, zero curator inclusion, and zero re-reads, the content never self-funds, never promotes to wider scope, and expires when the author stops paying storage. The clickbait author's profit is capped at one wave of retrieval fees minus ongoing storage costs.
5. **Low stakes per read** — A text post retrieval costs ~25–75 μMHR. Being clickbaited once is cheap. Expensive content (video at ~1000+ μMHR) is where the risk is higher — but that's exactly where readers naturally rely on curators rather than browsing raw feeds.

Clickbait on Mehr is a **diminishing-returns attack**: it works once per reader, generates no organic amplification, and the author pays ongoing storage for content that nobody will read a second time.

Threading and Interaction

Replies

Replies reference the parent post via `reply_to` on the envelope:

```
Reply to a post:
    PostEnvelope {
        reply_to: Some(parent_post_id),
        summary: "Great point!",
        ...
    }
    SocialPost {
        content: PostContent { text: "Great point! Here's why..." },
        ...
    }
```

Thread assembly is local — each client collects reply envelopes by querying the DHT for envelopes referencing a given hash. Threads are assembled client-side in chronological order. The envelope's summary is enough to display the thread tree — full posts are fetched only when a reader opens a specific reply.

Clients should limit thread traversal depth (recommended: 64 levels). Deeply nested reply chains beyond the limit are still accessible — the client just stops auto-fetching and shows a "load more" prompt. At mesh scale, threads rarely go deep; the economics of reply storage naturally limits chain length.

Boosts

A boost (repost) references the original via `boost_of` on the envelope:

```
Boost a post:
    PostEnvelope {
        post_id: None,                        // no SocialPost — boost is envelope-only
        boost_of: Some(original_post_id),
        summary: None,                        // original's envelope has the summary
        content_size: 0,
        ...
    }
```

Boosts are envelope-only — `post_id` is None and no SocialPost is created. When a reader fetches the boosted content, the original author receives kickback — not the booster. Boosts are pure amplification without capturing revenue.

References

References declare that a post is **related to** other posts — without threading (reply) or amplification (boost). They create a queryable content graph: the DHT can answer "what other posts reference this one?", enabling discovery of related discussions, counterarguments, and follow-ups across communities.

```
Reference other posts:
    PostEnvelope {
        references: [post_id_a, post_id_b, post_id_c],
        headline: "Why the bike lane debate misses the point",
        summary: "Alice, Bob, and Carol each analyzed the new bike lanes...",
        ...
    }
```

What references enable:

- **Related discussions**: Query the DHT for all envelopes where `references` contains a given hash. A reader viewing a popular post can discover every post that references it — counterarguments, analyses, translations, remixes.
- **Cross-community linking**: A Portland post referenced by a Tokyo post creates a connection between geographic communities. Interest relay nodes can surface cross-references.
- **Knowledge webs**: Scientific papers referencing other papers, course lessons linking to prerequisites, music remixes pointing to originals — any content type benefits from declared relationships.

How clients render references is application-dependent. A client might:

1. Show referenced envelopes as linked cards below the post (free — envelope fetch)
2. Show a "referenced by N posts" count with expandable list
3. Build a graph visualization of connected posts

4. Ignore references entirely (minimal client)

When a reader fetches a referenced post's full content, the **referenced author** gets kickback — same as any retrieval.

	Reply	Boost	Reference
Creates new content	Yes	No (envelope-only)	Yes
Relationship	Vertical (parent → child)	Amplification (repost)	Horizontal (related posts)
Protocol-level field	`reply_to` on envelope	`boost_of` on envelope	`references` on envelope
Queryable via DHT	"What replied to X?"	"Who boosted X?"	"What references X?"
Who earns kickback on original	Original author	Original author	Original author
Multiple targets	No (one parent)	No (one target)	Yes (any number)

Scoped Content

Every post declares its scopes — where it's relevant and how it propagates:

Local Post (Geographic Only)

```
PostEnvelope {
    scopes: [Geo("north-america", "us", "oregon", "portland")],
    headline: "Hawthorne Farmers Market",
    summary: "Farmers market on Hawthorne this Saturday! Fresh produce, live music...",
    ...
}
```

Envelope appears in Portland geographic feeds. Propagates cheaply within Portland. May bubble up to Oregon if popular.

Interest Post (Topic Only)

```
PostEnvelope {
    scopes: [Topic("gaming", "pokemon", "competitive")],
    headline: "VGC Meta Analysis",
    summary: "New VGC meta analysis after the latest patch...",
    media_hints: [MediaHint { content_type: "image/png", size: 85000, ... }],
    ...
}
```

Envelope appears in Pokemon interest feeds globally. Readers see the headline and summary for free. The 85KB image is only fetched (and paid for) when a reader opens the full post. Propagates through interest relay nodes. No geographic bias.

Cross-Scoped Post (Geographic + Interest)

```
PostEnvelope {
    scopes: [
        Geo("north-america", "us", "oregon", "portland"),
        Topic("gaming", "pokemon"),
    ],
    headline: "Portland Pokemon League Meetup",
    summary: "Meetup next Friday at Pioneer Square — all skill levels welcome",
    ...
}
```

Envelope appears in both Portland geographic feeds and Pokemon interest feeds. Portland Pokemon intersection subscribers see it automatically. Note: up to 3 Topic scopes are allowed (e.g., `Topic("gaming", "pokemon") + Topic("events", "meetups")`), but only 1 Geo scope per post, with total scope data capped at 1 KB — see scope constraints.

Neighborhood-Only Post (Private)

```
PostEnvelope {
    scopes: [],     // no declared scopes
    summary: "Block party at our place this weekend — neighbors only!",
    ...
}
```

Visible only within trust neighborhood. No propagation beyond trusted peers. The most private form of social posting. Since the audience is trusted peers only, the envelope and full post are both free to access.

Privacy depends on MHR-Pub scope routing: an envelope with no declared scopes matches no `Scope(...)` subscriptions, so it is never forwarded beyond direct MHR-Pub gossip between trusted peers. Nodes only gossip unscoped envelopes to their trusted peer set.

Media Tiering

The envelope/post split naturally creates a tiered browsing experience. Envelopes carry blurhash thumbnails via `MediaHint`; full media lives in the paid SocialPost as separate DataObjects with `min_bandwidth` constraints:

Layer	Content	Size	Cost	Available On

Envelope	Headline + summary	~300-500 bytes	Free	Everywhere, including LoRa
Envelope	Blurhash thumbnails (in MediaHint)	~30-64 bytes each	Free	Everywhere, including LoRa
Post	Full text body	~200 bytes - 10 KB	Retrieval fee	Everywhere
Post	Compressed image	~50 KB	Retrieval fee	WiFi and above
Post	Full resolution image	~500 KB	Retrieval fee	WiFi and above
Post	Video	1+ MB	Retrieval fee	High-bandwidth links only

The application decides what to fetch based on current link quality:

```
// Envelopes always arrive via MHR-Pub (free)
display(envelope.headline, envelope.summary, envelope.media_hints);

// On user tap — fetch full content
let link = query_link_quality(storage_node);

if link.bandwidth_bps < 1000 {
    // LoRa: text content only
    fetch(post.content.text);
} else if link.bandwidth_bps < 100_000 {
    // Moderate: text + compressed images
    fetch(post.content.text);
    fetch(post.content.media, max_size: 50_000);
} else {
    // High bandwidth: everything
    fetch(post.content.text);
    fetch(post.content.media);
}
```

Privacy

- **Public posts** (scoped): Propagate per declared scopes and propagation economics. Anyone can pay to read them.
- **Neighborhood-only posts** (unscoped): Visible only within trust neighborhood. Gossip only between trusted peers. The most private social posting.
- **Interest-only posts**: No geographic scope. Propagate through interest graph only — no geographic trail.

- **No social graph leakage**: Following is a local MHR-Pub subscription. No central server has a copy of the social graph. Only the subscriber and the publisher know about the subscription.
- **Unfollowing is silent**: Stop subscribing. No notification, no record.

Comparison: Economics vs. Algorithms

	Centralized Social	Mehr Social
Spam prevention	AI moderation (arms race)	Posting costs tokens (economically irrational)
Content ranking	Opaque algorithm optimizing engagement	Economic signal (what people pay to read)
Content lifespan	Platform decides	Funded by readership
Monetization	Ads; platform takes 100%	Direct author-reader kickback; no middleman
Censorship	Platform discretion	No central point of control
Feed curation	Algorithmic, engagement-optimized	Human curators with skin in the game
Cost to read	"Free" (you're the product)	μMHR for strangers; free for friends
Infrastructure	Company-owned data centers	Self-sustaining through service fees

Voice

Voice communication in Mehr ranges from push-to-talk over LoRa to full-duplex calls over WiFi, adapting to available bandwidth.

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

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.

Opus on WiFi

On higher-bandwidth links, Opus provides near-CD-quality voice with full-duplex operation (both parties can talk simultaneously).

Encryption

Voice streams are **end-to-end encrypted** using the standard E2E encryption 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.

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. The protocol has no concept of "voice" — it routes bytes. The application decides how to adapt between bandwidth tiers.

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.

Naming (MHR-Name)

MHR-Name provides human-readable names scoped to hierarchical scopes. There is no global namespace — names resolve locally based on proximity and trust.

Scope-Based Names

Names follow the format `name@scope`:

```
maryam@geo:tehran
alice@geo:portland
relay-7@geo:backbone-west
pikachu-fan@topic:gaming/pokemon
```

The scope portion uses the HierarchicalScope format. Geographic scopes are prefixed with `geo:`, interest scopes with `topic:`.

Resolution works by proximity:

1. Parse the scope from the name (e.g., `geo:portland`)
2. Search nearby nodes whose scopes match
3. Within that cluster, resolve the name via local naming records
4. Use the returned NodeID for routing

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.

Scope-based names solve this:

- Names are locally consistent within their scope
- Different communities can have different "alice" users — they are `alice@geo:portland` and `alice@geo:tehran`
- No global coordination needed
- Scopes are self-assigned, not centrally managed

Name Registration

Names are registered locally by announcing a name binding:

```
NameBinding {
    name: String,                        // "alice"
    scope: HierarchicalScope,            // Geo("portland") or Topic("gaming", "pokemon")
    node_id: NodeID,
    signature: Ed25519Signature,
}
```

Name bindings propagate via gossip within the matching scope. Conflicts (two nodes claiming the same name in the same scope) 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-Scope Resolution

To resolve a name in a different scope:

1. Query propagates toward nodes whose scopes match the target
2. Nearest matching cluster responds with the name binding
3. Result is cached locally with a TTL

Because scopes are self-assigned, resolution finds the **nearest** cluster with that scope. This is usually what you want — `alice@geo:portland` resolves to the Portland cluster nearest to you.

Collision across regions: Two different cities named "portland" (e.g., Portland, Oregon and Portland, Maine) would both match `geo:portland`. Proximity-based resolution handles this naturally — you'll reach whichever Portland is closer in the mesh. To disambiguate explicitly, use a longer scope path: `alice@geo:us/oregon/portland` vs. `alice@geo:us/maine/portland`.

Backward Compatibility

The old `name@community-label` format maps to the new scope format:

```
alice@portland-mesh  →  alice@geo:portland
relay-7@backbone-west  →  relay-7@geo:backbone-west
```

Nodes running older software continue to use the flat `community_label` field. New nodes check both `scopes` and `community_label` for resolution.

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

Community Applications

Community applications — forums, marketplaces, and wikis — are built on 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

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)

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.

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.

Hosting a Website

A "website" on Mehr is a collection of DataObjects — HTML, CSS, JavaScript, images — stored in MHR-Store and addressed by content hash or human-readable MHR-Name.

Static Site

```
Site structure:
  root = DataObject {
      hash: Blake3("index.html contents"),
      content_type: Immutable,
      payload: Inline("<html>...links to sub-objects...</html>"),
      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.

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

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_id_1, post_id_2, ...]),
      // Owner can update the post list by publishing a new version
      // Each individual post is immutable — only the feed index changes
  }
```

Hosting a Social Feed

A social feed is an append-only log of posts, served via MHR-Pub:

```
Your feed:
  1. Profile: Mutable DataObject (name, bio, avatar hash)
  2. Posts: Mutable DataObjects (editable, versioned via sequence number)
  3. Feed index: Mutable DataObject listing post IDs in order
  4. Subscriber notifications via MHR-Pub
```

Publishing a Post

```
Publishing flow:
  1. Create a mutable DataObject for the post content (keyed by post_id)
  2. Store it in MHR-Store with replication
  3. Update your feed index (mutable) to include the new post_id
  4. MHR-Pub notifies all subscribers of the update
```

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.

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.

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

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 for details.

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. Consumers find your service, form agreements, and pay via payment channels — all handled by the protocol.

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

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

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

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

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

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.

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

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/NPU Workstation)

- ML inference fees (highest per-invocation revenue)
- Heavy compute services
- Includes: GPU workstations, servers with NPU/TPU, FPGA accelerators
- 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 (GPU/NPU)
```

Each delegation is a bilateral capability agreement with payment flowing through channels.

Implementation Roadmap

The Mehr implementation follows a **server-first** strategy: get the protocol running on well-connected Linux servers over traditional internet, prove the core services work, then extend to phones and mesh radio. The protocol spec is comprehensive because it needs to be — but implementation is ruthlessly phased. Each phase delivers something people can use, not just something that passes tests.

```
graph LR
    P1["<b>Phase 1: Linux Server Node (MVP)</b><br/>TCP/IP transport<br/>Storage + DHT<br/>Trust graph<br/>Free tier only<br/><i>Users: server operators</i>"]
    P2["<b>Phase 2: Economics + Social</b><br/>Payment channels<br/>VRF lottery<br/>CRDT ledger<br/>Social feeds<br/><i>Users: economy bootstraps</i>"]
    P3["<b>Phase 3: Mobile + Mesh</b><br/>Phone apps<br/>LoRa relay nodes<br/>WiFi/BLE mesh<br/>Gateway operators<br/><i>Users: mobile communities</i>"]
    P4["<b>Phase 4: Full Ecosystem</b><br/>Advanced compute<br/>Licensing<br/>Onion routing<br/>Protocol bridges<br/><i>Users: mature ecosystem</i>"]

    P1 --> P2 --> P3 --> P4
```

Principle: Start where the resources are. Servers have bandwidth, uptime, storage, and compute. Debug the protocol on reliable hardware over reliable links, then extend to constrained devices and radio. The free tier (trusted peer communication) is a complete product on its own. MHR tokens, economics, and advanced features come only after there are real nodes generating real traffic. Token follows utility, never leads it.

Phase 1: Server Node (MVP)

Focus: A Linux daemon that lets servers join a decentralized network, providing compute, storage, and relay services over standard internet connections. The free tier only — no tokens, no payment.

Why server first: Servers have public IPs (or easily configured port forwarding), reliable uptime, abundant bandwidth, and real storage and compute resources to offer. TCP/IP transport is trivial compared to radio. This is the fastest path to a working protocol — debug routing, gossip, storage, and DHT on hardware that can actually run them well, without fighting radio propagation, phone OS restrictions, or constrained-device limitations. The `no_std` constraint for ESP32 can be added later; starting with the full Rust standard library makes development significantly faster.

Milestone 1.1: Core Protocol Library (Rust)

- `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)
- TCP transport interface (outbound connections, bidirectional links, keepalive)
- Announce generation and forwarding with Ed25519 signature verification

Acceptance: Two Linux nodes establish an encrypted link over TCP, exchange announces, and forward packets. Unauthenticated nodes are rejected.

Milestone 1.2: Bootstrap + Peer Discovery

- DNS-based genesis gateway discovery (well-known domain resolves to genesis gateway IPs — primary bootstrap method)
- Hardcoded bootstrap node list as fallback (known IP:port pairs, including genesis gateway addresses)
- Peer exchange protocol (connected peers share their known peer lists)
- Outbound-only NAT traversal (nodes behind NAT connect outbound to bootstrap nodes; TCP connection is bidirectional once established)
- Peer persistence (remember previously connected peers across restarts)

Acceptance: A new node discovers the genesis gateway via DNS and connects to the network within 30 seconds. After 3 gossip rounds, the node has discovered peers beyond the genesis gateway. Fallback to hardcoded list works when DNS is unavailable. A node behind NAT connects outbound and participates fully as a relay. Restarting a node reconnects to previously known peers without hitting the bootstrap list.

Milestone 1.3: 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)
- Announce propagation rules (event-driven + 30-min refresh, hop limit, expiry, link failure detection)

Acceptance: A 10-node network 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.

Milestone 1.4: Trust Graph + Free Relay

- `TrustConfig` implementation (trusted peers, cost overrides, scopes)
- Free relay logic (sender trusted AND destination trusted → no lottery, no channels)
- Adding/removing trusted peers

Acceptance: Trusted peers relay traffic for free with zero economic overhead. The full relay stack works with zero tokens in circulation.

Milestone 1.5: MHR-Store

- `DataObject` types (Immutable, Mutable, Ephemeral)
- Storage agreements (bilateral, free between trusted peers initially)
- 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)
- Chunking (4 KB chunks, parallel retrieval, resumable transfers)

Acceptance: A node stores a DataObject with replication factor 3 across the network. Proof-of-storage challenges pass. Erasure coding reconstructs from k of n chunks. Chunked transfer resumes after interruption.

Milestone 1.6: MHR-DHT + MHR-Pub

- DHT routing (XOR distance + cost weighting, α=0.7)
- k=3 replication with cost-bounded storage set
- Lookup and publication protocols
- Subscription types (Key, Prefix, Node, Scope)
- Delivery modes (Push, Digest, PullHint)
- Bandwidth-adaptive mode selection

Acceptance: A node publishes a DataObject and it's discoverable via DHT lookup from any node in the network. MHR-Pub delivers notifications to subscribers within 2 gossip rounds.

Milestone 1.7: Linux Daemon + CLI

- `mehrd` daemon (background process, systemd service file)
- Configuration file (bootstrap peers, listen address, storage path, trust config)
- CLI tool (`mehr`) for node management:
    - `mehr status` — node identity, connected peers, routing table summary
    - `mehr peers` — list connected peers with link quality metrics
    - `mehr trust add/remove <destination_hash>` — manage trusted peers

  - `mehr store put/get <file>` — store and retrieve data objects
  - `mehr dht lookup <key>` — query the DHT
- Logging and metrics (structured logs, Prometheus-compatible metrics endpoint)

Acceptance: A sysadmin can install the daemon, configure bootstrap peers, and join the network. The CLI provides full visibility into node state. The daemon runs unattended and recovers from restarts.

Phase 1 Deliverable

A network of Linux servers providing decentralized storage and relay. Install the daemon, configure a few bootstrap peers, and your server joins the network — contributing storage, relay bandwidth, and DHT capacity. Trust your friends' servers for free relay. This is the foundation everything else builds on.

Target audiences: homelabbers, self-hosting enthusiasts, VPS operators, privacy-conscious sysadmins, distributed systems developers.

Phase 2: Economic Layer + Social

Focus: MHR token genesis, payment infrastructure, and social features. With real servers generating real traffic and providing real storage, economics can be validated against actual usage patterns.

Milestone 2.1: 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 2.2: CRDT Ledger + Epoch Compaction

- `AccountState` (GCounter for earned/spent, GSet for settlements)
- GCounter merge (pointwise max per-node entries)
- GCounter rebase at epoch compaction (prevents overflow from money velocity)

- Settlement flow (validation: 2 sig checks + Blake3 hash + GSet dedup, gossip forward)
- Balance derivation (`earned - spent`, reject negative)
- Epoch trigger logic (3-trigger: settlement count, GSet size, small-partition adaptive)
- Epoch lifecycle (Propose → Acknowledge at 67% → Activate → Verify → Finalize)
- Merkle-tree snapshot (full tree — servers have the memory for it)
- `BalanceProof` generation and verification

Acceptance: A 20-node network triggers epochs correctly. Balances converge across the network. GCounter rebase keeps counters bounded.

Milestone 2.3: Token Genesis + Demand-Backed Mining

- Emission schedule implementation (10^12 μMHR/epoch, halving every 100,000 epochs, shift clamp at 63)
- Tail emission floor (0.1% of circulating supply annually)
- Genesis allocation to genesis gateway operator (transparent, visible in ledger from epoch 0)
- Demand-backed minting eligibility (VRF wins count only on funded-channel traffic)
- Revenue-capped minting (`effective_minting = min(emission_schedule, 0.5 × total_channel_debits)`)
- `RelayWinSummary` aggregation per epoch (demand-backed wins only)
- Mint distribution proportional to verified VRF lottery wins
- Channel-funded relay payments (coexist with minting)

Acceptance: The first epoch mints MHR and distributes it to relay nodes. Genesis allocation is visible in the ledger. Distribution is proportional to demand-backed wins. Minting on unfunded-channel traffic is rejected. Revenue cap limits minting to 50% of channel debits. Minting and channel payments coexist. Token supply follows the emission schedule.

Milestone 2.4: Reputation + Credit

- `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%)
- Transitive credit (direct: full, friend-of-friend: 10%, 3+ hops: none)
- `CreditState` tracking per grantee
- Credit rate limiting per trust distance and per epoch

Acceptance: Reliable nodes build reputation. Credit extends through trust graph. A friend-of-friend gets exactly 10% of the direct credit line. Default handling absorbs debt correctly.

Milestone 2.5: Paid Storage + Kickback

- StorageAgreement with payment channel integration (pay-per-duration)
- Kickback fields (revenue sharing between storage node and content author)
- Self-funding content detection (kickback exceeds storage cost)
- Content propagation through scope hierarchy

Acceptance: Storage nodes earn for hosting data. Kickback flows correctly on retrieval. Self-funding content persists without author payment.

Milestone 2.6: Social Layer

- `PostEnvelope` (free layer) + `SocialPost` (paid layer) — mutable DataObjects
- `UserProfile` (display name, bio, avatar, scopes, claims)
- Hierarchical scopes (Geo + Topic) with scope matching
- Five feed types: follow, geographic, interest, intersection, curated
- `CuratedFeed` with curator kickback
- Publishing flow (post_id generation, envelope propagation)
- Editing (mutable DataObject semantics, sequence versioning)
- Replies, boosts, references
- MHR-Name (scope-based naming, conflict resolution, petnames)

Acceptance: A user publishes a post tagged with geographic and interest scopes. The post's envelope appears in subscribers' feeds. Readers pay to fetch full content. Kickback flows to author. Curated feeds work end-to-end.

Milestone 2.7: MHR-Compute

- 47-opcode MHR-Byte interpreter implementation in Rust
- Cycle cost enforcement
- Resource limit enforcement (max_memory, max_cycles, max_state_size)
- WASM execution environment (Wasmtime, Light + Full tiers)
- Compute delegation via capability marketplace
- Reference test vector suite for cross-platform conformance

Acceptance: A compute contract executes on a server node. MHR-Byte and WASM produce identical results for the same inputs. Compute delegation routes requests to capable nodes. Cycle cost metering terminates runaway contracts.

Milestone 2.8: Test Network

- Deploy a 20-50 node test network across multiple server operators

- Instrument for: routing convergence, gossip bandwidth, storage reliability, economic dynamics, social UX
- Run for at least 8 weeks
- Document: failure modes, parameter tuning, real-world performance, economic balance

Acceptance: Test network operates continuously for 8 weeks. Storage proofs pass reliably. Token economy reaches equilibrium. Published test report with metrics.

Phase 2 Deliverable

A functioning decentralized economy on Linux servers. MHR tokens enter circulation through proof-of-service. Server operators earn by relaying traffic and hosting storage. Content creators earn through kickback. Compute delegation works. The economic layer is live, validated on real servers with real traffic.

Phase 3: Mobile + Mesh

Focus: Bring the proven protocol to phones and mesh radio. The protocol is already battle-tested on servers — now extend it to constrained devices and transport-independent operation.

Milestone 3.1: `no_std` Core Library

- Factor the core protocol library into `no_std`-compatible crate
- Separate transport-specific code (TCP) from protocol logic
- Verify all crypto operations work without `std` (Ed25519, X25519, ChaCha20, Blake3)
- Sparse Merkle-tree snapshots for constrained devices (under 5 KB)

Acceptance: The core protocol library compiles for `no_std` targets. All protocol-level tests pass on both `std` and `no_std` builds.

Milestone 3.2: Phone Apps

- Android app (Kotlin/Rust FFI) — Android first for broader device support and sideloading
- iOS app (Swift/Rust FFI) — follows Android
- Contact management (add trusted peers via QR code, NFC, or manual key entry)
- Messaging UI (conversations, groups, media sending adapted to link quality)
- E2E encrypted messaging (store-and-forward, offline delivery)
- Group messaging with co-admin delegation
- Voice on WiFi links (Opus codec)
- Connect to server network over internet (phone as a light client)

- WiFi Direct and BLE transport for local phone-to-phone mesh
- Background mesh relay (phone relays traffic while in pocket)
- Multi-transport handoff (internet ↔ WiFi Direct ↔ BLE, seamless)

Acceptance: A non-technical user can install the app, add a friend via QR code, and exchange messages. The app connects to the server network over internet for relay and storage. Phone-to-phone WiFi Direct messaging works without internet. Voice calls work on WiFi links.

Milestone 3.3: LoRa Transport

- LoRa interface implementation (SX1276/SX1262 via `no_std` Rust)
- Off-the-shelf hardware support:
  - Heltec WiFi LoRa 32 (~$15)
  - LILYGO T-Beam (~$25, with GPS)
  - RAK WisBlock (~$30, modular)
  - RNode (Reticulum-native)
- LTE-M and NB-IoT interface support (carrier-managed LPWAN)
- Multi-interface bridging (phone WiFi ↔ LoRa relay ↔ WiFi ↔ server network)
- Solar relay firmware (ESP32 L1: transport, routing, gossip — runs on $30 solar kit)
- Congestion control tuning for constrained links (CSMA/CA, backpressure)

Acceptance: A LoRa relay extends the network to areas without internet. A phone sends a message that hops: phone → WiFi → LoRa relay → WiFi → server → destination. Solar relay runs unattended for 30+ days.

Milestone 3.4: Gateway Operators

- Gateway trust-based onboarding (add consumer to trusted_peers, extend credit)
- Fiat billing integration (subscription, prepaid, pay-as-you-go — gateway's choice)
- Cloud storage via gateway (consumer stores files, gateway handles MHR)
- Gateway-provided connectivity (ethernet ports, WiFi access points)

The genesis service gateway is the first instance of this pattern — it bootstraps the economy in Phase 2. This milestone generalizes gateway mechanics for any operator to deploy.

Acceptance: A consumer signs up with a gateway, pays fiat, and uses the network without seeing MHR. Traffic flows through gateway trust. Consumer can switch gateways without losing identity.

Milestone 3.5: Mesh Test Networks

- Deploy 3-5 physical test networks (urban, rural, campus, event)

- Each network: 10-50 nodes (phones + LoRa relays + server backbone) across at least 2 transports
- Instrument for: routing convergence, gossip bandwidth, storage reliability, mesh UX
- Run for at least 4 weeks per network
- Document: failure modes, parameter tuning, real-world performance

Acceptance: Test networks operate continuously for 4 weeks. Users report messaging and mesh features work reliably. Published test report with metrics.

Phase 3 Deliverable

The full network: servers as backbone, phones as endpoints, mesh radio for off-grid. Phone users get encrypted messaging and social feeds backed by the server network's storage and compute. LoRa relays extend coverage to areas without internet. Gateway operators bridge fiat consumers to the mesh economy. The same protocol runs from $30 ESP32 to datacenter servers.

Phase 4: Full Ecosystem

Focus: Advanced capabilities, application richness, and ecosystem growth.

Milestone 4.1: Identity + Governance

- Identity claims and vouches (GeoPresence, CommunityMember, KeyRotation, Capability, ExternalIdentity)
- RadioRangeProof (geographic verification via LoRa beacons)
- Peer attestation and transitive confidence
- Vouch lifecycle (create, gossip, verify, renew, revoke)
- Voting prerequisites (geographic eligibility from verified claims)

Milestone 4.2: Rich Applications

- Voice (Codec2 on LoRa, Opus on WiFi, bandwidth bridging, seamless handoff)
- Digital licensing (LicenseOffer, LicenseGrant, verification chain, off-network verifiability)
- Cloud storage (client-side encryption, erasure coding, sync between devices, file sharing)
- Forums (append-only logs, moderation contracts)
- Marketplace (listings, escrow contracts)

Milestone 4.3: Interoperability + Privacy

- Third-party protocol bridges (SSB, Matrix, Briar) — standalone gateway services with identity attestation
- Onion routing implementation (`PathPolicy.ONION_ROUTE`, per-packet layered encryption)
- Private compute tiers (Split Inference, Secret Sharing, TEE)

Milestone 4.4: Ecosystem Growth

- Developer SDK and documentation
- Community-driven capability development
- Hardware partnerships and reference design refinement (informed by real deployment data)
- Custom hardware (only if demand justifies — let usage data guide form factors)

Phase 4 Deliverable

A full-featured decentralized platform with rich applications, privacy-enhanced routing, identity governance, 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 (added in Phase 3)
- Strong ecosystem for cryptography and networking
- Single codebase from microcontroller to server
- FFI to Kotlin (Android) and Swift (iOS) for phone apps

Phase 1-2 use the full standard library. The `no_std` factoring happens at the start of Phase 3 when embedded targets are introduced.

Platform Targets

Platform	Implementation	Phase
Linux server / desktop	Rust native daemon + CLI (full node)	Phase 1
Raspberry Pi / Linux SBC	Rust native (bridge, gateway, storage)	Phase 1-2
Android phone	Rust core + Kotlin UI via FFI	Phase 3
iOS phone	Rust core + Swift UI via FFI	Phase 3
ESP32 + LoRa	Rust `no_std` (L1 relay)	Phase 3

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

Bootstrap Strategy

New nodes discover the network through a layered bootstrap mechanism:

1. **Hardcoded bootstrap list**: The daemon ships with a list of known bootstrap node IP:port pairs. These are well-connected, high-uptime servers operated by early network participants.
2. **DNS bootstrap**: A well-known domain resolves to current bootstrap node IPs. This allows updating the bootstrap list without software releases.
3. **Peer exchange**: Once connected to any node, the gossip protocol discovers the rest of the network. Connected peers share their known peer lists.
4. **Peer persistence**: Previously connected peers are remembered across restarts, so a node that has been online before rarely needs the bootstrap list.

The bootstrap list is a starting point, not a dependency. After initial connection, the announce mechanism and gossip protocol handle all peer discovery. Bootstrap nodes have no special protocol role — they are ordinary nodes that happen to be well-known.

NAT Traversal

Most servers targeted in Phase 1 have public IPs or easily configured port forwarding. For nodes behind NAT:

- **Outbound TCP connections** traverse NAT automatically. A node behind NAT connects outbound to a known peer; the TCP connection is bidirectional once established. This covers the vast majority of home server setups.
- **Relay forwarding**: A node that cannot accept inbound connections is still reachable — traffic routes through peers that have direct links to it. This is standard Mehr relay operation.
- **UPnP/NAT-PMP** (optional): Automatic port forwarding on consumer routers, attempted on startup.
- **UDP hole punching** (Phase 3+): For phone-to-phone mesh scenarios where both parties are behind NAT.

No STUN/TURN servers are required. The Mehr relay mechanism itself provides the relay function that TURN would otherwise fill.

Test Network Strategy

Real distributed test networks, not simulations:

- Phase 1 test networks are server-only (TCP/IP over internet)
- Phase 2 test networks enable economics and measure token dynamics on real traffic
- Phase 3 test networks add phones and LoRa relays, validating the full transport range
- Each test network should represent a different deployment scenario
- Test networks validate both the protocol and the user experience

Why This Order

The previous roadmap (phone-mesh-first) would require solving radio transport, constrained-device limitations, phone OS restrictions, and `no_std` compatibility before any protocol logic could be tested. This roadmap gets the protocol running first on hardware where development is fast, then extends to harder targets:

Phase	What Users Get	What the Network Gets
1	Decentralized storage + relay on Linux servers	Real nodes, real traffic, battle-tested protocol
2	Token economy, social feeds, compute	Real economic data, real market pricing
3	Phone apps, mesh radio, off-grid operation	Mobile users, mesh coverage, transport diversity
4	Rich apps, privacy, interoperability	Mature ecosystem

Each phase is viable on its own. Phase 1 is a useful product even if phones never ship — a decentralized storage and relay network for server operators. Phase 2 adds a functioning economy. Phase 3 extends to mobile and mesh. No phase depends on token speculation or hardware manufacturing for its value.

Key advantage: By the time Phase 3 tackles constrained devices and radio, the protocol is already proven. Bugs in routing, gossip, storage, DHT, and economics have been found and fixed on servers where debugging is easy. The `no_std` factoring is a well-scoped engineering task applied to known-good code, not a simultaneous protocol design and embedded engineering challenge.

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
Packet format + CompactPathCost	Complete (wire format specified)	Network Protocol

Routing + Announce propagation	Complete (scoring, announce rules, expiry, failure detection)	Network Protocol
Gossip protocol	Complete (bloom filter, bandwidth budget, 4-tier)	Network Protocol
Congestion control	Complete (3-layer, priority levels, backpressure)	Network Protocol
Trust neighborhoods	Complete (free relay, credit, scopes)	Trust & Neighborhoods
MHR-Store	Complete (agreements, proofs, erasure coding, repair)	MHR-Store
MHR-DHT + MHR-Pub	Complete (routing, replication, subscriptions)	MHR-DHT, MHR-Pub
VRF lottery + Payment channels	Complete (RFC 9381, difficulty formula, channel lifecycle)	Payment Channels
CRDT ledger + Settlement	Complete (validation rules, GCounter merge, GSet dedup, rebase)	CRDT Ledger

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 as the initial transport implementation; treat it as a swappable layer
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
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 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: Scope-Based, No Global Namespace

Chosen	Hierarchical-scope-based names (e.g., `alice@geo:us/oregon/portland`)
Alternatives	Global names via consensus, flat community labels (`alice@portland-mesh`)
Rationale	Global consensus contradicts partition tolerance. Flat community labels were replaced by hierarchical scopes — geographic (`Geo`) and interest (`Topic`) — which provide structured resolution. Names resolve against scope hierarchy: `alice@geo:portland` queries Portland scope first, then broadens. Two different cities named "portland" are disambiguated by longer paths (`alice@geo:us/oregon/portland` vs `alice@geo:us/maine/portland`). Proximity-based resolution handles most cases naturally. 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
Ratio	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~~

nale	~~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
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. Compute execution is opaque — the protocol does not verify the compute method, only the input/output. Verification strategies (reputation, redundant execution, spot-checking) are consumer-side choices, not protocol enforcement.

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)
Ratio	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~~

nale	~~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

Chos	Initial reward of 10^12 µMHR per epoch, discrete halving every 100,000 epochs, tail emission floor at 0.1% of circulating supply annualized

en	
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, the winning epoch's `epoch_balance` snapshot is adopted and the losing partition's settlements are recovered via settlement proofs, preserving individual balances. 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.

Genesis Model: Transparent Service Gateway

Chosen	Genesis Service Gateway + demand-backed minting + revenue-capped emission
Alternatives	Pure proof-of-service mining (no genesis), airdrop, ICO, DAO treasury
Rationale	VRF prevents grinding (one output per relay-packet pair) but does not prevent traffic fabrication: a Sybil attacker can fabricate traffic between colluding nodes and run the VRF lottery on fake packets. A genesis service gateway bootstraps the economy with real demand: a known trusted operator provides real services for fiat, creating funded payment channels that generate legitimate relay traffic. Revenue-capped minting (see below) ensures that even paying through the gateway and self-dealing is unprofitable. The genesis allocation is transparent (visible in ledger from epoch 0). DNS provides initial gateway discovery; the hardcoded bootstrap list serves as fallback.

Revenue-Capped Minting

Chosen	`effective_minting = min(emission_schedule, 0.5 × total_channel_debits)`
Alternatives	Uncapped emission (vulnerable to self-dealing), fixed low emission (doesn't scale with network growth), proof-of-stake gating (concentrates power)
Rationale	The revenue cap guarantees that self-dealing is unprofitable at all traffic levels: an attacker spending Y MHR on relay fees can receive at most 0.5 × Y in minting rewards, regardless of their share of the network. This holds for any Y, any epoch, and any number of Sybil nodes. The cap also makes supply growth demand-responsive: during low-usage periods, actual minting is well below the emission schedule (supply grows slowly, preventing speculation). As the network matures and relay fees increase, the cap becomes non-binding and supply follows the standard halving schedule. Fixed low emission was rejected because it doesn't scale — a fixed amount becomes either too generous at low traffic or too stingy at high traffic. Proof-of-stake gating was rejected because it concentrates minting power among large holders.

Configurable Transitive Credit Ratio

Chosen	User-configurable transitive credit ratio (0–50%, default 10%)
Alternatives	Fixed protocol constant (10%), fully uncapped user choice
Rationale	Direct credit is already configurable per-peer; transitive ratio should be too. The 50% ceiling prevents naive users from excessive exposure to friend-of-friend defaults. Different communities have different risk appetites — a tight-knit village may want 30% transitive credit, while an urban mesh with loose trust edges may prefer 5%. Per-peer overrides (`transitive_ratio_overrides`) allow fine-grained control. A fixed 10% was too rigid; fully uncapped allows users to set 100%, which could cascade defaults through the trust graph.

Single Geo Scope Per Node/Content

Chosen	Max 1 Geo scope + up to 3 Topic scopes per node/content; total scope data ≤ 1 KB

Alternatives	Unconstrained (up to 8 of either type), total byte budget only (no count limit), no geo scopes on content
Rationale	Physical presence is singular — you are in one place at a time. Voting is geo-scoped, and multiple geo scopes would enable double-voting on governance issues. RadioRangeProof verifies one location; multiple geo claims would require multiple independent proofs. Interests are multi-dimensional but hierarchical prefix matching means a single deep tag (e.g., `Topic("gaming", "pokemon", "competitive")`) covers multiple query levels — 3 topic slots is expressive enough for most users. The 1 KB byte cap (down from ~2.1 KB for 8 scopes) halves the scope footprint in announces and envelopes, meaningful for ESP32 and LoRa. A pure byte budget (no count limit) was rejected because it creates a perverse incentive: deep, specific tags cost more bytes, penalizing the most useful scopes. A hard count + byte cap gives predictable allocation for constrained devices while letting the UI show "bytes remaining" (like character counts).

Per-Hop Independent Relay Rewards

Chosen	Per-hop independent rewards (VRF lottery per relay, no end-to-end coordination)
Alternatives	End-to-end payment (sender pays all relays in one transaction), hybrid (per-hop with end-to-end settlement)
Rationale	End-to-end payment requires the sender to know the full path, which breaks sender anonymity — packets carry no source address by design. Per-hop stochastic rewards already achieve ~0.3% bandwidth overhead; end-to-end would save negligible additional bandwidth. Per-hop uses simple bilateral channels; end-to-end requires multi-hop payment routing (Lightning-style complexity). Each hop is independent — no coordination failure cascade. If one relay goes offline, only that hop is affected; upstream and downstream relays continue operating on their own bilateral channels. The hybrid approach adds complexity without meaningful benefit over pure per-hop.

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
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
3	Mutable Object Fork Detection Reporting	5-step protocol: record, block 24h, gossip advisory, 7-day dedup, resolution via KeyCompromiseAdvisory	MHR-Store
4	CongestionSignal on Multi-Interface Nodes	Replaced link_id with scope enum (ThisLink/AllOutbound); 3 bytes total	Network Protocol
5	DHT Rebalancing Timing	2 gossip rounds convergence on join; 6 additional missed rounds before re-replication; graceful degradation	MHR-DHT
6	Beacon Collision Handling	Density adaptation: interval doubles at 50% utilization, triples at 75%; Ring 1 gossip provides redundancy	Discovery
7	Fragment Reassembly	30s per-chunk timeout, exponential backoff (3 retries), 5-min overall, resumable via ChunkRequest	MHR-Store
8	Transitive Credit Rate-Limiting	Per-epoch, per-grantee tracking via CreditState struct; epoch-boundary reset; vouching peer absorbs defaults	Trust Neighborhoods

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
CompactPathCost normalization ambiguity	Normalization uses decoded values, not log-encoded wire values	Network Protocol

Agreement lifecycle (expiry, renewal, billing)	Defined Active/Expired/Grace/Closed states, per-capability expiry behavior, renewal protocol	Agreements
Settlement timing (eager vs lazy)	Settlements to CRDT ledger are not per-win; created on cooperative close, dispute, or periodic finalization	Payment Channels
Channel sequence semantics	Sequence is monotonic version number; replay protection via final_sequence comparison	Payment Channels
CapabilitySummary cost encoding	Identical $\log_2$ formula to CompactPathCost; 0x0000=free, 0xFFFF=unknown	Discovery
Merkle root trust for constrained nodes	Signed by proposer; trusted peers accept immediately; untrusted requires 2-source quorum	CRDT Ledger
Reputation initialization values	New peers start at 0; referrals capped at 5000; first-hand replaces after 5 interactions	Security

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
2	Presence Beacon Capability Bitfield	8 assigned bits (relay, gateway, storage, compute-byte, compute-wasm, pubsub, dht, naming); bits 8-15 reserved	Discovery
3	Ring 1 Capability Aggregation Format	`CapabilitySummary` struct: 8 bytes per type (type, count, min/avg cost, min/max hops)	Discovery
4	DHT Metadata Format	`DHTMetadata` struct: 129 bytes (key, size, content_type, owner, ttl, lamport_ts, Ed25519 signature)	MHR-DHT
5	Negotiation Protocol Wire Format	Single-round take-it-or-leave-it; 30-second timeout; `CapabilityRequest` + `CapabilityOffer` with nonce replay prevention	Agreements

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

2	Reputation gossip vs. first-hand only	Bounded 1-hop trust-weighted referrals, capped at 50%, advisory only	Design Decisions
3	Onion routing for high-threat environments	Per-packet layered encryption, opt-in via PathPolicy, 3 hops default, 21% overhead on LoRa	Design Decisions
4	MHR-Byte full opcode specification	47 opcodes in 7 categories, reference interpreter in Rust, ESP32-calibrated cycle costs	Design Decisions
5	Bootstrap emission schedule parameters	10^12 μMHR/epoch initial, discrete halving every 100K epochs, 0.1% tail floor	Design Decisions
6	Protocol bridge design (SSB, Matrix, Briar)	Standalone gateway services, identity attestation, bridge operator pays Mehr costs	Design Decisions
7	Formal verification targets	TLA+ priority-ordered: CRDT merge, payment channels, epoch checkpoints; composition deferred	Design Decisions

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
ChaCha20 nonce handling	64-bit counter per session key, zero-padded to 96 bits	Security
Session key rotation timing	Local monotonic clock, either side can initiate	Security
KeyCompromiseAdvisory replay	Added monotonic `sequence` field	Security
Difficulty target formula	Local per-link computation, `win_prob = target_updates / observed_packets`	Payment Channels
Settlement validation	Every node validates (2 sig checks + hash + GSet lookup), invalid dropped silently	CRDT Ledger
Active set definition	Nodes appearing as party in at least 1 settlement in last 2 epochs	CRDT Ledger
Bloom filter construction	k=13 Blake3-derived hash functions, 19.2 bits/element at 0.01% FPR	CRDT Ledger
Relay compensation tracking	RelayWinSummary with spot-check proofs, challengeable during grace period	CRDT Ledger

Roaming cache fingerprint stability	60% overlap threshold for area recognition	Discovery
Credit-based fast start staleness	Rate-limited grants, 2× balance requirement for staleness tolerance	Discovery
Trust relationship revocation	Asymmetric, revocable at any time, downgrades stored data priority	Trust Neighborhoods

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 overhead budget	≤10% of link bandwidth	Bandwidth Budget
CompactPathCost size	6 bytes (constant)	Network Protocol
MehrExtension magic byte	`0x4E` ('N')	Network Protocol
Destination hash size	16 bytes (128-bit)	Network Protocol
Smallest MHR unit	1 μMHR	MHR Token
Supply ceiling	2^64 μMHR (asymptotic)	MHR Token
Default relay lottery probability	1/100	Stochastic Rewards
Payment channel state size	200 bytes	Payment Channels
Dispute challenge window	2,880 gossip rounds (~48h)	Payment Channels
Channel abandonment threshold	4 epochs	Payment Channels
Epoch trigger: settlement count	≥10,000 batches	CRDT Ledger
Epoch trigger: GSet memory	≥500 KB	CRDT Ledger
Epoch acknowledgment threshold	67% of active set	CRDT Ledger
Epoch verification window	4 epochs after activation	CRDT Ledger

Bloom filter FPR (epoch)	0.01%	CRDT Ledger
DHT replication factor	k=3	MHR-DHT
DHT XOR weight (w_xor)	0.7	MHR-DHT
Storage chunk size	4 KB	MHR-Store
Presence beacon size	20 bytes	Discovery
Presence beacon interval	10 seconds	Discovery
Transitive credit limit	10% per hop, max 2 hops	Trust & Neighborhoods
Max scopes per node	8	Trust & Neighborhoods
Max scope depth	8 segments	Trust & Neighborhoods
Max scope segment length	32 characters	Trust & Neighborhoods
Vouch expiry (default)	30 epochs	Identity & Claims
Kickback rate range	0–255 (u8)	MHR-Store
Default kickback rate	128 (~50%)	Content Propagation
IdentityClaim min size	~100 bytes	Identity & Claims
Vouch size	113 bytes	Identity & Claims
Geo verification: min vouches	3 (for Verified level)	Voting
Protocol version encoding	1 byte (major 4 bits, minor 4 bits)	Versioning
Extended version escape	Major = 15 → read u16 pair from TLV	Versioning
Current protocol version	`0x10` (v1.0)	Versioning
Emission halving shift clamp	max 63 (prevents UB at epoch 6.4M+)	MHR Token
Max curated feed entries	256 per page	Social
LicenseOffer min size	~160 bytes	Digital Licensing
LicenseGrant size	226 bytes	Digital Licensing
Max custom license terms	1024 characters	Digital Licensing

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

Layer Dependency Graph

```
graph TD
    L6["Layer 6: Applications<br/>Messaging, Social, Identity, Voice, Naming, Voting, Licensing, Cloud Storage, Roaming, Forums, Hosting"] --> L5
    L5["Layer 5: Service Primitives<br/>MHR-Store, MHR-DHT, MHR-Pub, MHR-Compute"] --> L4
    L4["Layer 4: Capability Marketplace<br/>Discovery, Agreements, Verification"] --> L3
    L3["Layer 3: Economic Protocol<br/>MHR Token, Stochastic Rewards, CRDT Ledger, Trust Neighborhoods, Propagation"] --> L2
    L2["Layer 2: Security<br/>Link encryption, E2E encryption, Authentication, Key management"] --> L1
    L1["Layer 1: Network Protocol<br/>Identity, Addressing, Routing, Gossip, Congestion Control"] --> L0
    L0["Layer 0: Physical Transport<br/>LoRa, WiFi, Cellular, LTE-M, NB-IoT, Ethernet, BLE, Fiber, Serial"]
```

Each layer depends **only** on the layer directly below it. Applications never touch transport details. Payment never touches routing internals.

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

Wire Format Summary

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)
```

Mehr Announce Extension

```
[MAGIC 1B: 0x4E] [VERSION 1B] [CompactPathCost 6B] [TLV extensions...]
Minimum: 8 bytes. Carried in announce DATA field.
```

CompactPathCost

```
[cumulative_cost 2B] [worst_latency_ms 2B] [bottleneck_bps 1B] [hop_count 1B]
Total: 6 bytes (constant regardless of path length)
```

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
```

Specification Sections

Spec Section	Documentation Page
0. Design Philosophy	Introduction
1. Layer 0: Physical Transport	Physical Transport
2. Layer 1: Network Protocol	Network Protocol

3. Layer 2: Security	Security
4. Layer 3: Economic Protocol	MHR Token, Stochastic Relay Rewards, CRDT Ledger, Trust & Neighborhoods, Content Propagation, Content Governance, Real-World Economics
5. Layer 4: Capability Marketplace	Overview, Discovery, Agreements, Verification
6. Layer 5: Service Primitives	MHR-Store, MHR-DHT, MHR-Pub, MHR-Compute
7. Layer 6: Applications	Messaging, Social, Identity, Voice, Naming, Community Apps, Voting, Digital Licensing, Cloud Storage, Roaming, Hosting
8. Hardware Reference	Reference Designs, Device Tiers
9. Implementation Roadmap	Roadmap
10. Design Decisions	Design Decisions
11. Protocol Versioning	Versioning
12. Open Questions	Open Questions

Version

Version	Status
v1.0	**Current**

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.