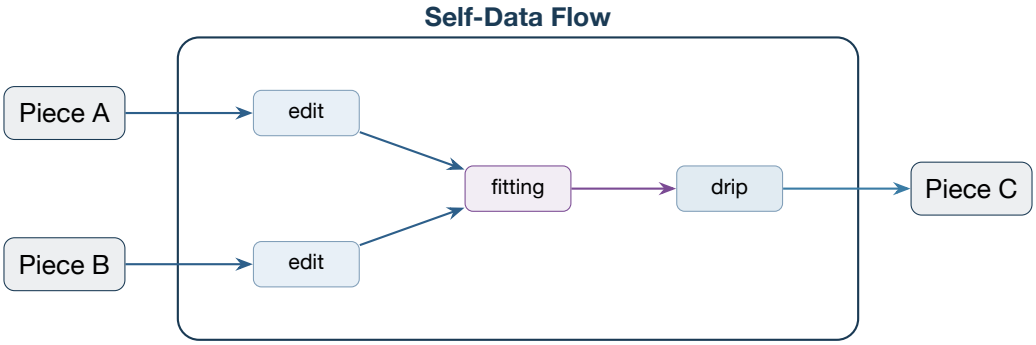
Rim Self-Data Protocol

Self-Data Flows

Architecture, Concepts, and Design



Soradyne Project

Draft — February 12, 2026

Reference implementation: `soradyne_core` (Rust)

Contents

1 Introduction

1.1 What is Self-Data?

Self-data is data that belongs to a person and exists on their devices. It is not hosted by a third party on their behalf; it lives where they put it. The Rim Self-Data Protocol defines how self-data moves between devices, how conflicts are resolved when multiple devices edit the same data, and how the data is stored, secured, and retrieved.

The protocol assumes maximum heterogeneity among participating devices. A single session might involve a phone, a laptop, a VR headset, an ESP32 microcontroller, and a robot arm driven by Python on a Raspberry Pi—each a *piece* in the protocol's terminology (§11). Each piece may run different operating systems, communicate over different transports (Bluetooth Low Energy, TCP, bespoke hardware interfaces), and have vastly different storage and compute capabilities. The protocol defines agreements about *what* must happen, and leaves *how* to the implementations on each device.

1.2 Design Philosophy

- **User-owned**: Data lives on devices the user controls. Third-party servers may participate as roles in a flow, but are never the sole authority.
- **Peer-to-peer**: No distinguished leader. Any device can make edits offline; all devices converge when they communicate.
- **Device-heterogeneous**: The protocol does not assume homogeneous platforms. Implementations exist per-language, per-platform, and conform to versioned design documents rather than to a single reference binary.
- **Local-first**: Operations succeed locally and synchronize later. Network partitions degrade gracefully; they never block local work.
- **Application-defined**: The protocol provides infrastructure. Applications define what their data looks like, how conflicts resolve, and what trade-offs they accept. The protocol does not make these decisions for them.
- **Body-proximate, BLE-first**: The protocol treats Bluetooth Low Energy as the prototypical network implementation, not TCP/IP. This is a deliberate inversion of the usual assumption that sockets or HTTP are basic while wireless variations are exotic. The goal is to be as locality-, body-, fashion-, and hardware-centered as possible. IP-based transports (TCP, UDP, WebSockets) are available as extensions, but the baseline design targets the constraints and affordances of BLE: short range, low bandwidth, broadcast-capable, battery-efficient, and physically proximate.

1.3 Scope of This Document

This document defines **Self-Data Flows**: the core abstraction by which data is created, moved, transformed, stored, and queried across devices in the Rim protocol.

It covers:

- The conceptual model: flows, streams, flow roles, fittings, policies

- Data geometry: how the spatial, temporal, and structural character of data is described so that flows can operate on it uniformly
- Worked examples across diverse data types: images, video, sound, spatial measurements, photo albums, structured records, and task graphs
- The implementer's perspective: how to define new flow types, how versioning works, how to extend the system
- Security and memorization: policies for data at rest, dissolution, robustness
- Device topology: how devices are grouped (capsules), tracked in real time (ensembles), and integrated at the hardware level (parures)

1.4 Relationship to Existing Code

The reference implementation in `soradyne_core` (Rust) includes early versions of many concepts described here. Where the code and this document diverge, the document represents the intended design. Specifically:

- `DataChannel<T>` (formerly `SelfDataFlow<T>`) is a concrete implementation of a stream, not a flow.
- `ConvergentDocument<S>` is a tool for implementing a drip stream's backing store, not the drip itself.
- The `Flow` trait, `FlowRegistry`, and `FlowConfigStorage` in `flow_core.rs` reflect the bootstrap-from-UUID model described here and are directionally correct.

2 Self-Data Flows

Self-Data Flow

A self-data flow is a persistent, typed, UUID-identified, authenticated instance of a bundle of streams, created from a schema for building streams, along with a collection of policies for how serialized data is moved and transformed between streams, policies for delegating who performs transmissions and transformations, and policies for exceptions.

New streams may be created according to the schema after the bundle is created.

2.1 Identity: The UUID

Every flow instance has a UUID. This is the stable handle that applications use. Everything else—the type, the configuration, the participating devices, the network addresses—is looked up from the UUID or discovered through policies that the UUID leads to.

An application that wants to read or write data opens a flow by UUID. It assumes the flow has a particular type and therefore a particular interface. If the assumption is wrong, it gets an error. If no one is implementing the flow's streams, it gets an error per the flow's error policy.

2.2 Flow Types

A flow type is a versioned design document that specifies:

1. What streams the flow has (names, cardinalities, categories)
2. What flow roles are needed to operate the flow
3. What policies govern error handling, delegation, discovery, memorization
4. What interface the flow presents to applications
5. What serialization formats are used

Flow types are defined at the **application level**. The protocol provides the infrastructure; applications (or domain-specific standards) define the types.

2.2.1 Type Identity via Content Hash

A flow type's definition can be serialized as a canonical JSON (or similar structured text) document. This document is sorted deterministically and hashed. The hash *is* the type's identity. Human-readable names and version strings are convenience tags associated with the hash, but the hash is authoritative.

Type Identity

```
type_definition_text = canonical_sort(serialize(flow_type_definition))
type_hash = sha256(type_definition_text)

# Tags (convenience, not authoritative):
#   "rim.inventory.v1" -> type_hash
#   "rim.giantt.v2" -> type_hash
```

Any method can be used to retrieve the text of a type definition, and easily confirmed even if it came from an untrusted source, because the hash can be recomputed.

2.2.2 Type Scope

Some flow types are **device-specific**: defined by a particular device's software, never needing cross-vendor compatibility. A robot arm's joint-state flow type is defined by its manufacturer's software. Only that software (and tools designed to work with that device) ever need to know the type definition.

Other flow types are **gestalt**: messaging, photo albums, notes. These represent ideas about data that are more fundamental than any particular implementation. Their flow type definitions must be public, versioned, and stable, so that any application can implement them and interoperate.

In either case, the type is set at compile time by the application, not negotiated at runtime by the transport layer.

2.3 Flow Configuration

A flow's configuration is the set of parameters specific to one instance of a type. While the type defines *what streams exist and how they behave in general*, the configuration defines *instance-specific details*: which parties participate, what network addresses to try, what storage quotas apply, etc.

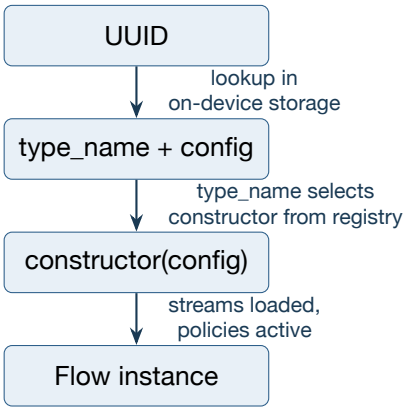
Configuration Example

```
{
  "id": "a1b2c3d4-...",
  "type": "sha256:abc123...",     // or "rim.inventory.v1"
  "params": {
    "initial_parties": ["device_A", "device_B"],
    "storage_policy": "local_only",
    "max_history_bytes": 10485760
  }
}
```

Configuration lives in on-device storage. Currently it must be distributed ahead of time. The near-term evolution path:

1. **Now**: Configuration files distributed manually or bundled with the app
2. **Soon**: An initiating party broadcasts configuration during a setup round
3. **Later**: Configuration discoverable via static hosts, DHT, or other mechanisms

2.4 Bootstrap Sequence



A new device joining a flow needs only the configuration. It does not even need the type map if it can assume the type is correct (because flows are defined at the application level—the app already knows what type it expects).

From the application's perspective, the entire bootstrap is:

Application Bootstrap

```
let flow = registry.load(uuid, &storage)?;
let state = flow.stream("current_state")?.read()?;
// done. UI populates, edits work.
```

3 Streams

Streams are the basic I/O primitive of flows. A flow contains streams. Applications read from and write to streams. Everything that enters or leaves a flow goes through a stream.

3.1 What a Stream Is

A stream is a named, typed channel within a flow through which serialized data moves. Streams have:

- A **name**, unique within the flow, given in the schema
- A **cardinality**: singleton (one per flow), per-party (one per participant), or unbounded (created as needed, e.g., threads or topics)
- A **category hint**: drip, jet, or unspecified
- An **interface**: read, write, subscribe

The category hint is a shorthand for the stream's behavioral contract, not a type enforced by the system. "Drip" and "jet" do not appear in function signatures. They appear in flow type definitions as familiar labels that let readers skip past long formal names.

3.2 Drips: Convergent Streams

Drip

A drip is a stream that provides eventually consistent, authoritative data. Any holder of a drip should give the same answer after a settling time. Drips are slow, deliberate, and represent consensus.

Drips are typically backed by some form of convergent data structure (CRDT, operational transform, or similar). The backing implementation is injected by the parties responsible for the drip, not specified by the stream abstraction itself.

Examples of drips:

- The current merged state of a task graph (Giantt)
- The current structure of a photo album (order, crops, annotations)
- The current consensus alignment of a coordinate system network (Nestbox)
- The current merged state of a personal inventory

3.3 Jets: Fast Streams

Jet

A jet is a stream that provides fast, possibly lossy, real-time data. Jets prioritize promptness over completeness or consensus.

Jets carry live interaction data: cursor positions, typing indicators, sensor readings, video frames, audio samples. Data on a jet may arrive out of order, may be incomplete, and is not expected to be the same from every observer's perspective.

Examples of jets:

- Who is currently viewing or editing what in a shared photo album
- Live audio streams during a group call
- Real-time motor encoder readings from a robot arm
- IMU and camera data from a phone participating in spatial alignment

3.4 Stream Lifecycle

Streams defined in the schema exist conceptually from the moment the flow is created. Whether data is returned from a stream depends on whether any party has implemented it. If no party is serving a stream, requests to it activate the flow's error policy (which might return "data not available," retry, fall back to another party, etc.).

New streams can be created according to the schema's rules (e.g., a per-party stream is created when a new party joins). Streams are not deleted, but may become inactive if all parties stop serving them. The flow's policies determine what happens when inactive streams are queried.

4 Data Geometry

Different kinds of data have fundamentally different spatial, temporal, and structural character. A protocol that aims to handle images, sound, 3D measurements, task graphs, and binary blobs through the same flow abstraction must have a way to describe these differences so that operations like *query*, *subset*, *transform*, and *merge* can be defined uniformly.

This section defines the vocabulary for describing data geometry. These concepts appear in flow type definitions and determine what operations streams support.

4.1 Query Coordinates

Query coordinates are the dimensions along which you *request* data from a stream. They define the "address space" of the data.

Data type	Query coords	Type	Notes
Binary blob	offset	discrete (integer)	One coordinate. All-or-nothing also valid.
Sound	time	continuous (float)	One coordinate. Subsetting = time range.
Image	x, y (or u, v)	discrete or continuous	Two ordered coordinates.
Video	x, y, time	mixed	Spatial discrete, temporal continuous.
3D measurements	x, y, z	continuous	Transforms apply. May include time.
Text	line, offset	discrete (integer)	Two coordinates, both ordered.
Task graph	item ID	discrete (unordered)	One coordinate, a set of IDs.
Inventory	item ID	discrete (unordered)	One coordinate, a set of IDs.
Photo album	position	discrete (ordered)	One coordinate: slot in order.

4.1.1 Properties of Query Coordinates

Each query coordinate has properties that determine what operations are meaningful:

- **Discrete vs. continuous**: Can you request data at integer indices only (pixels, bytes, lines), or at arbitrary real-valued positions (time in seconds, spatial position in meters)?
- **Ordered vs. unordered**: Does the coordinate have a natural ordering (time, position, line number) or is it a set of identifiers (item IDs, UUIDs)? Ordered coordinates support range queries; unordered coordinates support set membership queries.
- **Bounded vs. unbounded**: Is there a known extent (image width, file length) or can the coordinate space grow indefinitely (append-only log, growing inventory)?
- **Homogeneous vs. inhomogeneous**: Are the coordinates in a projective space (camera image coordinates often are) where scaling matters, or in an affine space where it does not?

4.2 Data Coordinates

Data coordinates are the dimensions of the *values* returned at each query point.

Data type	Data coords	Count	Notes
Binary blob	byte value	1	Technically a bit, practically a byte.
Grayscale image	intensity	1	
RGB image	R, G, B	3	
RGBA image	R, G, B, A	4	
Multispectral image	wavelength bands	n	JWST: 29 bands.
Sound (mono)	amplitude	1	
Sound (stereo)	L, R amplitude	2	
Sound (surround)	channel amplitudes	n	5.1, 7.1, Atmos, etc.
3D position	x, y, z	3	Often with covariance.
6D pose	x, y, z, roll, pitch, yaw	6	Or quaternion: 7.
Text	character	1	Unicode codepoint.
Task graph item	structured record	varies	Title, status, priority, …

4.2.1 Properties of Data Coordinates

- **Discrete vs. continuous**: Byte values are discrete (0–255); spatial positions are continuous (floats).
- **Scalar vs. structured**: A pixel's RGB is a flat vector; a task graph item is a structured record with named fields of different types.
- **Uncertain vs. exact**: Spatial measurements carry covariance matrices quantifying uncertainty. Pixel values are exact (within quantization).

4.3 Transforms

When query coordinates live in a geometric space (2D, 3D, projective), data can be requested through a transform. An image flow could accept a 2×2 matrix and start/end vectors, returning any rotation, scale, flip, and crop of the image. A 3D measurement flow could accept a rigid-body transform to return data in a different coordinate system.

Transforms are part of the query interface, not the data itself. They allow the same underlying data to be accessed from different perspectives without redundant storage.

> **Design Decision: Server-side vs. Client-side Transforms**
>
> For most data (images at typical resolutions, short audio clips), transforms are better applied on the receiving device. For very large data (gigapixel images, long high-rate sensor logs), supporting transforms in the query—so that only the relevant subset is transmitted—is important. Flow type definitions should specify which transforms the stream supports in queries.

4.4 Sparsity and Density

Some data fills its coordinate space densely (images: every pixel has a value). Other data is sparse (3D feature measurements: values exist at a few scattered points in a continuous space). The distinction matters for how data is stored, transmitted, and merged:

- Dense data benefits from array-based storage and compression.

- Sparse data benefits from indexed storage (coordinate $\to$ value maps).
- Merging two sparse datasets is a set union with conflict resolution at coincident points.
- Merging two dense datasets requires per-element conflict resolution or layer-based composition.

5 Flow Roles

5.1 What a Flow Role Is

A *flow role* is an abstract unit of responsibility within a flow. Flow roles are defined in the flow's type definition. Parties (devices, processes, services) *fill* roles at runtime.

Flow Role

A flow role is a named set of responsibilities defined by a flow type. A single device can fill many roles. Many devices can fill the same role. Roles are abstract; they do not imply separate hardware, separate processes, or even separate lines of code—though they might be any of those.

The key insight is that hundreds of roles might be filled by a single device, even for a single flow. A phone running an inventory app might simultaneously fill the roles of data source (user makes edits), memorization (stores edit history to local files), fitting (applies edits to produce current state), and data sink (displays current state in UI). These are all conceptually separate responsibilities that happen to be co-located.

5.2 Common Flow Role Types

5.2.1 Data Source

A data source produces new data and writes it to a stream. Examples:

- A user making edits in an app UI
- An LLM generating tool calls
- A sensor producing readings
- A CLI command modifying a task graph

5.2.2 Data Sink

A data sink consumes data from a stream. Examples:

- A UI displaying the current state of an inventory
- A robot arm consuming joint-angle commands
- A speaker playing back mixed audio

5.2.3 Memorization

A memorization flow role reliably stores and recalls data that was written to streams, so that it can be retrieved later. This is often implemented as local file I/O, but the flow only specifies the *what* (this data must be recallable under these conditions) and the *policy* (how robust, how secure, how long), not the *how*.

Memorization roles carry security policies:

- **Plaintext-safe**: The host is behind user accounts, passwords, and standard security. Data can be stored in plaintext on local drives.
- **Encrypted-at-rest**: The storage location is accessible to others. Data must be encrypted before writing.
- **Dissolution-required**: After a time horizon, an attacker would need to physically acquire a threshold number of devices to read the data. This triggers data dissolution (erasure coding across multiple devices).

Memorization roles also carry **robustness policies**:

- **Ephemeral**: If the holder disappears, the data is gone. Acceptable for transient caches and live-only jets.
- **Replicated**: Data is held by multiple memorization roles. If some disappear, others still have it.
- **Reallocating**: If holders disappear over time, the remaining holders proactively redistribute data to new roles with updated routes. This is the common policy for persistent self-data.

5.2.4 Curator

A curator grooms raw data for consumption. In the spatial measurement context (Section 7.4), curators remove erroneous measurements, denoise, and produce summarized measurement sets that aligners query to catch up. Curators are often co-located with the devices that produce the raw data, but this is a convenience, not a requirement.

5.2.5 Aligner

An aligner is a specialized role for spatial flows (Section 7.4). Aligners consume curated measurements and produce a consensus alignment (a drip) of coordinate systems. Multiple aligners processing the same data should converge to the same answer.

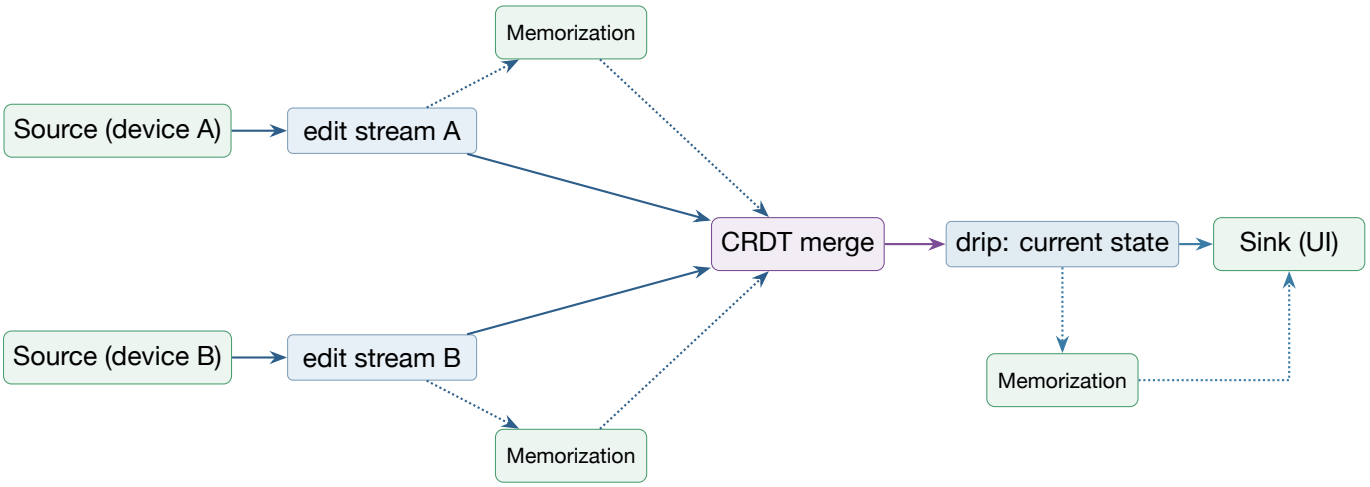
5.3 Fittings

Fitting

A fitting is a transformation that connects streams to each other within a flow. Fittings describe how data from one stream is processed and made available on another.

A fitting's functionality is real and implemented in real code—the parties running the code *are* implementing the fittings described in the flow's policies. But a fitting is not a first-class named entity at the same level as a stream. There is no fitting registry, no fitting UUID, no fitting interface that you call methods on. The fitting is implicit in its implementation: the code that reads from an edit stream, applies operations to a CRDT, and makes the result available on a drip stream *is* a fitting, but it does not need to know that about itself.

A fitting might be as simple as "apply each edit operation from the edit stream to a CRDT, making the result available on the drip stream," or as complex as "mix n audio input streams into a single output stream, applying gain normalization and latency compensation." In both cases, the fitting is a description of what the code does, not a separate object the code instantiates. Delegation policies (§6) determine which party runs the code that implements a given fitting, and failover policies determine what happens when that party disappears—but the fitting itself remains a noun for talking about transformations, not a type for instantiating them.



6 Policies

Policies are the rules encoded in a flow's type definition and configuration that govern how streams behave under various conditions. They are the "law" of the flow: once you have a flow, the policies call the shots.

6.1 Error Policies

What happens when a stream is queried but no party is implementing it? What happens when a memorization role can't recall data? What happens when a fitting fails?

Error policies specify:

- What abstract error to emit ("data not available," "uninitialized," "degraded")
- Whether to retry, and how
- Whether to fall back to alternative parties
- Whether to notify the application or handle silently

6.2 Delegation Policies

Who performs transmissions and transformations? Delegation policies specify:

- Which roles are responsible for which streams
- Whether responsibility can be transferred
- How new parties are assigned roles when they join
- How roles are redistributed when parties leave

6.3 Discovery Policies

How do pieces in a capsule find each other when they need to form an ensemble? Discovery policies specify:

- **Proximity broadcast**: Pieces advertise presence and capsule membership via encrypted BLE advertisements. Other pieces holding the capsule's key material can detect and respond. This is the baseline discovery mechanism.
- **Simple**: The flow configuration contains a static list of known pieces
- **Referral**: A known piece can provide the identities of others
- **Registry**: A remote directory service maintains a list of pieces (an IP-based extension for when proximity alone is insufficient)
- **DHT**: Distributed hash table lookup (another IP-based extension)

The policy also specifies what to do when discovery information is stale (retire the flow? attempt re-discovery? fall back to a different method?). Because the baseline is proximity broadcast, staleness often resolves itself: a piece that walks back into range is rediscovered automatically.

6.4 Memorization Policies

How is data stored at rest? Memorization policies combine security and robustness:

Policy aspect	Options
Security at rest	Plaintext-safe, encrypted-at-rest, dissolution-required
Robustness	Ephemeral, replicated, reallocating
Retention	Indefinite, time-bounded, size-bounded, policy-on-overflow
Provenance	Track origin device and timestamp, or not

6.5 Retirement and Refresh

Flows are persistent by nature, but they can be retired. Retirement means no new data is accepted on any stream, and existing data is subject to the memorization retention policy. Some flows benefit from being short-lived and easily refreshed (a quick spatial alignment session among 3 devices), while others are meant to last indefinitely (a personal photo album).

The flow's type definition specifies whether retirement is manual, automatic (time-based, inactivity-based), or not supported.

7 Data Type Examples

This section walks through how diverse types of data map onto the self-data flow abstraction. Each example defines the data geometry, the streams, the roles, and the key policies.

The examples are ordered from spatially rich to structurally rich, beginning with data types that have strong geometric character and progressing toward data types whose structure is more about relationships and records than about coordinates.

7.1 Images

An image has rich spatial structure. Rather than treating images as a single monolithic flow, the protocol decomposes image handling into three tiers of flows reflecting a key insight: raw image data, collaborative composition, and collection management have fundamentally different data characters and independent lifecycles. A photograph someone took and stores on their devices does not need multi-party consensus—it is spatial data you query at a resolution. A collaboratively edited composite (layers, transforms, annotations) does need consensus on the sequence of edits. A collection of composites (an album) needs consensus on membership and ordering. These are separate flows, linked by UUID references in a flow mesh (§10).

7.1.1 Data Geometry (Base Image)

- **Query coordinates**: 2. Either discrete (x, y) as integer pixel indices, or continuous (u, v) as floats in $[0, 1]$.
- **Data coordinates**: 3 for RGB, 4 for RGBA, 1 for grayscale, n for multispectral (up to 29 for some JWST images).
- **Ordered**: Yes, both query coordinates are ordered.
- **Dense**: Yes, every query point has a value.
- **Transform support**: The full query interface supports a 2×2 matrix and a pair of start/end vectors, expressing any rotation, scale, flip, and crop. Most transforms are better done on the receiving device, but for very large images, the query should support spatial subsetting. A valid starting implementation may use discrete resolution tiers (thumbnail, preview, large, full) as the query parameter, deferring continuous transform support to later iterations.

7.1.2 Three-Tier Decomposition

1. **Image flow** (query-response, neither jet nor drip). Stores raw image data and serves it at a requested resolution. Memorization-backed, spatially aware. Multiple pieces can memorize the same image; synchronization is "do you have this data?" not "what is the consensus state?" There is no drip because there is no multi-party state to converge—the image simply exists as spatial data.
2. **Image composite flow** (has a **drip**). Produces a rendered result from one or more image sources plus transforms, overlays, and annotations. The drip converges when multiple parties contribute edits—crops, rotations, layer operations, filter applications. A simple composite might contain only a genesis operation ("load from image flow X at resolution Y"); a rich composite might contain dozens of layer operations. The composite flow also returns non-image data (vector annotations, text overlays) alongside the rendered pixels.

3. **Album flow** (has a **drip**). A convergent collection of composite references. Each entry references a composite flow by UUID. The album's edits are structural: add, remove, reorder entries, set captions and tags. Image transforms (crop, rotate, filter) belong to the composite layer, not the album. See §7.5.

7.1.3 Image Flow Streams

- **Image data** (query-response, no category): Returns image data at a requested resolution. The resolution parameter is part of the query. This is neither a drip (no multi-party consensus to fuse) nor a jet (not continuous or lossy).
- **Metadata** (read-once, no category): Dimensions, format, original size, color space.

7.1.4 Image Composite Flow Streams

- **Drip** (`operations`): The convergent sequence of composition operations. When multiple parties contribute edits (transforms, layers, annotations), the drip ensures convergence via its backing CRDT.
- **Rendered** (query-response, no category): The composited image data plus extra data (annotations, vectors) at a requested resolution.
- **Jet** (`viewport`): Per-party, what region of the composite each participant is currently viewing. Optional; useful for collaborative editing UIs.

7.1.5 Key Policies

- Conflict resolution for concurrent composition edits: operation ordering via the drip's backing CRDT.
- Large images: memorization roles may store tiles or resolution tiers, serving subsets on query.
- The image flow and the composite flow referencing it have independent lifecycles: the same raw image can appear in many composites; retiring a composite does not affect the underlying image data.

7.2 Video

Video extends images with a time coordinate.

7.2.1 Data Geometry

- **Query coordinates**: 3. Spatial (x, y) plus temporal t (continuous).
- **Data coordinates**: Same as image (RGB, RGBA, etc.).
- **Dense in space, dense or sparse in time**: Every pixel in a frame has a value; frames may be keyframes with interpolation between them.

7.2.2 Streams

- **Drip** (`frame_state`): The consensus sequence of frames. Queryable by time range and spatial region.
- **Jets** (per-party): Live camera feeds or screen shares. Fast, lossy, per-observer.
- **Edit streams**: Cuts, splices, overlays, filters, applied to the temporal structure.

7.2.3 Relationship to Calls and Recordings

A video call is a self-data flow where each participant contributes a jet (their live camera feed). Each participant's local recording of the call is itself a data entity: the truth of what happened at one end, saved locally for reference by other flows (personal albums, file systems). The flow's drip—the consensus "what happened"—is not the same as any participant's live observations. It is the multi-screen, high-quality reconstruction assembled from all participants' locally stored data, whether as pointers into their local storage or as a literal instantiated file.

7.3 Sound

Sound has minimal spatial structure but strong temporal structure and strict latency requirements during playback.

7.3.1 Data Geometry

- **Query coordinates**: 1. Time t (continuous). Subsetting = time range.
- **Data coordinates**: n channels. Mono: 1. Stereo: 2. Surround: 5.1, 7.1, Atmos, etc.
- **Free parameters**: Gain, sampling frequency.
- **Ordered, dense in time** (at the sampling rate).

7.3.2 Streams

- **Drip** (`mixed_audio`): The consensus of what tracks should have been playing what when. Equivalent to a full multitrack project (Audacity-like).
- **Jets** (per-party): Live audio input. For a group call, these are the participants' microphone streams.
- **Edit streams**: Import clips, manipulate clips (trim, move, gain adjust), add/remove tracks. The drip reflects the consensus edit state; querying it causes mixing to happen at the specified sampling frequency.

7.3.3 Latency and Decompression

Sound must be incredibly prompt during playback. It is better to insert filler (silence, comfort noise, interpolation) than to break the stream and restart. This motivates:

- Jets return data in a format designed to be fed into decompression and smoothing.
- A trivial decompressor (passthrough) is a simple fitting from jet to OS audio output.

- Smarter decompressors may want richer data (prediction residuals, codebook info) from the jet to do more intelligible gap-filling. This is an open design question: how much metadata to include in the jet format.

Starting Big

It might be better to start big with sound, just as with text (Section 7.9): all sounds are playbacks of collections of tracks with clips. A phone call is clips continuously imported into separate tracks. Querying causes mixing. This avoids special-casing simple scenarios and immediately generalizes.

7.4 Spatial Measurements

Spatial measurement flows support the distributed, multi-device, multi-coordinate-system alignment problem. This is the domain of Nestbox, where devices with different sensors (cameras, IMUs, LIDAR, robot encoders) observe shared features in a physical space and must agree on how their coordinate systems relate.

7.4.1 Data Geometry

- **Query coordinates**: 3D (or 6D with velocity, or 2D for some sensors). Continuous. Subject to coordinate system transforms.
- **Data coordinates**: Multivariate Gaussian. Mean vector plus precision (inverse covariance) matrix. Dimensionality, cardinality, and whether the measurement is projective are all flexible.
- **Sparse**: Measurements exist at scattered points, not densely.
- **Uncertain**: Every measurement carries quantified uncertainty.

7.4.2 Why Precision Matrices

Precision matrices (inverse covariance) should be preferred over covariance matrices throughout. They save unnecessary inversions (many optimization algorithms work natively in information form), compose more naturally for independent measurements, and avoid numerical issues with near-singular covariances.

7.4.3 Coordinate Systems

Each device cares about certain coordinate systems. A flow bundles coordinate systems and defines transforms between them. The transforms are maintained by aligners and served as drips.

A device that can move but not measure (a basic robot arm) needs only to receive a transform from a drip to convert commands in one coordinate system to its local frame.

7.4.4 Streams

This flow type has an unusually large number of streams:

- **Drips**:
  - Per-aligner: current best-effort alignment (all should converge)
  - Curated measurements: the denoised, pruned measurement set
- **Jets**: Raw measurements from each device, streamed to aligners/curators
- **Historical streams**: Measurement histories, alignment state histories, for new aligners catching up

7.4.5 Flow Roles

- **Sensor**: Data source. Produces raw measurements in its local coordinate system.
- **Curator**: Grooms measurements. Removes outliers, denoises, summarizes. Often co-located with sensors due to data proximity.
- **Aligner**: Consumes curated measurements, produces consensus alignment. Multiple aligners should converge.
- **Consumer**: Reads alignment drips to get transforms. May be a robot arm that just needs to know how to convert coordinates.

7.4.6 Flow Boundaries

Should the alignment session be one big flow or many small ones? Key considerations:

- A "current effort with the devices that are here now" should be easy to retire or refresh after use.
- Long-lived persistent alignment data might live in a different flow.
- Many flows can form a mesh—not everyone needs to be on one huge global flow.
- A new flow can be announced briefly among, say, 3 devices.

See Section 10 for general principles.

7.4.7 Relationship to Nestbox Twigs

The existing Nestbox Twig system (Protocol Buffers, `SampleRouter`, dimension enums) implements exactly the sensor-to-aligner path described above, but with bespoke plumbing per use case. The Twig's `stream_id` and `coord_sys_id` map to stream identity within a flow. The `MeasurementSet` with its dimensions, covariance, transforms, and homogeneity flags maps to the data geometry vocabulary defined in Section 4. The `SampleRouter` is a fitting.

What Twigs lack is the flow-level abstraction: no UUID bundling streams, no policies for memorization/discovery/error, no schema declaring what streams exist. Every new device type requires new bespoke routing configuration. Self-data flows generalize the pattern.

7.5 Photo Albums

A photo album is a self-data flow whose data is a convergent collection of references to image composite flows (§7.1). Each album entry references a composite flow by UUID; the album reads from the composite to obtain image data for display, ignoring the composite's non-image data (annotations, vectors) unless the UI chooses to present it.

The album's concerns are structural: what composites are in the collection, in what order, with what captions and tags. Image-level transforms (crop, rotate, filter) belong to the composite layer—the same image can appear in two albums with different crops, because the crops are properties of different composite flows, not of the album entries.

7.5.1 Streams

- **Drip** (`album_state`): The current structure of the album. Queryable for the ordered list of entries, each entry's composite flow UUID, caption, tags, and album-level metadata. Backed by a ConvergentDocument.
- **Edit streams** (per-party): Add entry (referencing a composite flow), remove entry, reorder, set caption, add/remove tags. These are purely structural operations on the collection; no image data passes through the album's edit streams.
- **Jets** (per-party): Who is currently viewing what, who is typing a comment where. Prompt, non-consensus.

7.5.2 Flow Roles

- Standard: data source (editor), data sink (viewer), memorization (stores album state), fitting (applies structural edits to album CRDT).

7.6 Binary Data

A block of binary data is the simplest self-data flow. It has no *a priori* knowable internal structure; you can only have it all-or-nothing (or by byte offset).

7.6.1 Data Geometry

- **Query coordinates**: 1. Offset from start (discrete, integer).
- **Data coordinates**: 1. Byte value (discrete, 0–255).
- **Ordered, bounded** (known length).

7.6.2 Streams

- **Drip**: The most recent version with edits integrated. Conflict resolution is crude (e.g., if two parties insert at the same offset, one wins per policy—this is explicitly undefined/policy-dependent).
- **Edit streams**: Append, prepend, insert at offset, complete replacement (most common).

7.7 Structured Records: Inventory

A personal inventory is a collection of items with properties. It maps naturally to a ConvergentDocument-backed drip.

7.7.1 Data Geometry

- **Query coordinates**: 1. Item ID (discrete, unordered set).
- **Data coordinates**: Structured record (name, quantity, location, tags, custom fields). Variable per item.

7.7.2 Current Implementation and Migration Path

The current inventory app uses a Dart CRDT with file-based operation logs. The migration to a self-data flow means:

1. The flow UUID stands for one complete inventory.
2. The flow's type determines what modules are loaded for memorization (local file I/O for now) and fittings (CRDT apply for the drip).
3. The app reads from the drip (current inventory state) and writes to an edit stream (commands).
4. Memorization: edit histories are accumulated in per-stream files, as they are now. The only change is that the behavior is loaded as a module from the flow, so it could be swapped for dissolution, network storage, etc. without the app knowing.
5. Other devices' edits arrive as additional streams. Their histories are stored the same way. The fitting merges all streams into the drip.

Dart CRDT vs. Rust Port

Two paths for the CRDT implementation:

1. Port the Dart CRDT to Rust (like Giantt). Clean, everything in soradyne_core.
2. Keep the Dart CRDT; soradyne integration invokes it via callback/binding as the fitting implementation for the drip.

The decision depends on whether the Dart CRDT has features that would be expensive to reimplement, and whether the callback overhead is acceptable. For consistency with Giantt, porting to Rust is recommended.

7.8 Task Graphs: Giantt

A Giantt task graph is a collection of items with typed relations (REQUIRES, ANYOF, BLOCKS, etc.), time constraints, status, priority, and tags. It is a structured record flow with graph semantics.

7.8.1 Data Geometry

- **Query coordinates**: 1. Item ID (discrete, unordered set). But graph structure imposes a partial order via dependencies.
- **Data coordinates**: Structured record (title, status, priority, duration, time constraints, tags, charts, relations to other items).

7.8.2 Streams

- **Drip** (`graph_state`): The current merged task graph. Backed by a ConvergentDocument with GianttSchema.
- **Edit streams** (per-party): Add/remove items, set fields, add/remove relations. Each edit carries causal context (horizon).

7.8.3 Current Implementation

The Rust ConvergentDocument with GianttSchema already implements the drip backing. The Dart CLI and Flutter app produce edits. The migration wraps these in a flow so that the CRDT, the edit source, and the memorization are all accessed through the flow by UUID, with the fitting logic loaded from the flow's type.

7.9 Text

Line-based text can be managed by well-understood conflict-resolved diff methods.

7.9.1 Data Geometry

- **Query coordinates**: 2. Line number and character offset (both discrete, ordered). Or: 1, just byte offset.
- **Data coordinates**: 1. Character (Unicode codepoint).

7.9.2 Starting Big: File Trees

The default text flow type should represent an entire file tree: prefixed names mapped to IDs. A single-file flow contains one name (`/`) mapped to one ID (hash of flow ID). Multi-file flows map paths to IDs. All multi-diffs always apply atomically to all text they regard, like git.

7.9.3 Streams

- **Drip**: Merged text state. Queryable by file path, line range, offset.
- **Edit streams**: Groups of diffs (insertions, deletions, replacements) with causal context.

> **Text is De-Emphasized**
>
> Text-based CRDTs are a well-studied problem and an easy one to articulate. The Rim protocol's text support is important but not foundational. The examples above (images, sound, spatial measurements) are more representative of the protocol's intended scope and the kinds of problems it uniquely addresses.

7.10 Vector Images

Vector images are structured text (XML for SVG, AI format, etc.) but are not truly text—they have symmetries (reordering tags, grouping elements) that line-based diff does not respect.

7.10.1 Open Question

Should vector image flows be built on a structured-text flow that respects XML/HTML/JSON symmetries (reordering, nesting), with SVG as one application? Or should SVG flows be derived independently? The structured-text approach is more general but harder. The direct approach risks reinventing the wheel for each structured format.

Operations that would be faithful to vector image semantics: merge by groups, select by inclusive/exclusive box bounds, select by group or symbol identity.

8 The Implementer's Perspective

8.1 Defining a New Flow Type

To define a new flow type:

1. Write a design document specifying all streams, roles, policies, data geometry, and the application-facing interface.
2. Serialize the design document to a canonical form and compute its content hash. This hash is the type identity.
3. Tag the hash with a human-readable name and version string for convenience.
4. Implement the type's constructor, which accepts a `FlowConfig` and returns a `Flow` with the right streams and policies.
5. Register the constructor in the `FlowRegistry` under the type name (or hash).

The design document is the source of truth. Implementations in different languages conform to the design document, not to each other's code. The hash ensures that everyone agreeing on a type hash is agreeing on exactly the same specification.

8.2 Cross-Language Implementation

The Rim protocol is designed to be implemented in multiple languages. The Rust implementation in `soradyne_core` is the reference, but applications in Dart, Python, C#, Swift, etc. should be able to:

1. Provide a UUID to an interface object (in their language) backed by the Rim protocol.
2. Treat the resulting object as having a known type with a known interface.
3. Read from streams, write to streams, subscribe to updates.

This works because flow types are defined by design documents (the "PDF"), not by Rust structs. An implementer in Python reads the design document and implements the same streams, roles, and policies. The content hash of the type ensures compatibility can be verified.

8.3 Extending vs. Replacing

Many roles defined in flow type design documents are "drop-in": generic memorization, generic discovery, generic error handling. These are provided by the Soradyne distribution and can be used by any flow type.

Some flow types innovate on specific roles (a novel alignment algorithm, a specialized compression scheme) while keeping the generic roles for everything else. The plugin architecture supports this: register a flow type constructor that wires together standard modules for most roles and custom code for the novel parts.

Flows on minimal hardware (ESP32, embedded Linux) may use stripped-down implementations of standard roles. The flow type's design document describes the interface; the implementation is whatever fits on the device.

8.4 What Ships with Soradyne

The Soradyne distribution includes:

- Several built-in flow types (inventory, task graph, photo album, binary blob)
- Pre-made configurations for each type
- UUIDs that index those configurations
- A library of standard role implementations (local file memorization, in-memory streams, basic discovery, ConvergentDocument for drips)

A new device joining an existing flow needs only the configuration. After that, just UUIDs. "Load flow X" $\rightarrow$ UI populates, edits work.

9 Security and Memorization

9.1 Data at Rest

Every memorization role operates under a security policy specified by the flow. The three tiers:

1. **Plaintext-safe**: Host is behind standard security (user accounts, passwords). Data stored in plaintext.
2. **Encrypted-at-rest**: Storage location is accessible to others. Data encrypted before writing, keys managed by the flow's authentication system.
3. **Dissolution-required**: After a time horizon, an attacker must physically acquire a threshold number of devices to read the data. Implemented via data dissolution (erasure coding across devices using Reed-Solomon or similar).

9.2 Data Dissolution and Crystallization

Data dissolution splits data across multiple devices using erasure coding. No single device holds enough to reconstruct the data. Crystallization is the reverse: when enough devices come together, the data can be reconstructed.

The flow's memorization policy specifies:

- The threshold: how many devices are needed to crystallize
- The total: how many devices hold shards
- The time horizon: after how long dissolution is required
- Reallocation: what happens when shard-holding devices disappear

9.3 Robustness to Loss

The most common robustness policy for persistent self-data is **reallocating**: if holders of memorization roles disappear over time, the remaining holders proactively redistribute data to new roles with updated routes. This ensures that even if every original host were to disappear a few at a time over a long period, the data survives by continuously migrating to new hosts.

This is distinct from simple replication. Replication assumes a static set of holders. Reallocation assumes a dynamic set and actively manages the transitions.

9.4 Leakage Policies

Memorization policies must specify the level of leakage allowed: what happens if someone outside the Rim ecosystem looks at the stored data? This determines whether plaintext, encryption, or dissolution is required.

9.5 Authentication and Authorization

Every exchange of information through a flow confirms identity and authorization. This is part of the policies for streams and drips:

- Who is authorized to write to each stream

- Who is authorized to read from each stream
- How authorization is verified (signatures, tokens, challenge-response)
- What happens on authorization failure

10 Flow Boundaries

When should something be one flow vs. multiple flows? The guiding principle:

Flow Boundary Principle

A self-data flow's boundary is defined by the streams and drips that *must* be fitted back to each other invisibly from the perspective of users of the Rim protocol API. If interconnections can be managed by splitting into several flows, each managed independently by an application, and information shuttled between them in ways the application would naturally expect, then they should be separate flows.

Example: "Get a photo" should involve only one flow, even if Soradyne engages many devices in multiple steps to fetch it. "Move a photo to a new album that you got from an older one" naturally involves two flows at the application level.

10.1 Guidelines

- **Atomic unit of data**: If the data makes no sense without all its parts (an image's pixels, an inventory's items), it's one flow.
- **Independent lifecycle**: If parts of the data should be retirable independently (a temporary spatial alignment session vs. persistent calibration data), they should be separate flows.
- **Participant scope**: If not all participants need all data, separate flows avoid forcing everyone onto one big global flow. A mesh of many small flows is often better.
- **Application-natural boundaries**: If the application would naturally treat something as two separate entities (two albums, two inventories, two task lists), they should be separate flows.

10.2 Flow Mesh

Multiple flows can reference each other. A photo album flow contains references (by UUID or content hash) to image data that may live in separate binary-blob flows or image flows. This creates a mesh of flows, each independently managed, discoverable, and retirable, but linked by references.

The mesh is not a protocol-level construct—it emerges from applications storing references to other flows' UUIDs in their data. But the protocol should make this easy by ensuring that UUIDs are stable, discoverable, and sufficient to bootstrap access.

11 Device Topology: Capsules, Ensembles, and Pieces

Previous sections describe flows, streams, roles, and policies in terms of abstract *parties*—anything that can fill a role. This section specifies how devices are grouped, authorized, and tracked at the physical and network level. The terminology is drawn from fashion and jewelry, where items are curated into coordinated sets, worn in varying combinations, and sometimes sold as matched collections.

11.1 Cryptographic Identity as Foundation

Everything in the device topology—capsule membership, encrypted advertisements, pairing verification, flow authentication—depends on cryptographic device identity. This is not a feature that can be added later; it is the non-negotiable foundation that all other layers build on.

Each piece has an Ed25519 signing keypair (for authentication and signing operations) and an X25519 key-agreement keypair (for establishing shared secrets during pairing and encrypting communications). Together these form the cryptographic root of a piece's identity. The signing key proves "I am who I claim to be"; the key-agreement key enables "we can talk privately."

When a capsule is created, shared symmetric key material is established and distributed to all members during the pairing step. This key material enables encrypted BLE advertisements (so only capsule members can detect each other) and per-session encryption of data connections. Key material is bound to the capsule, not to individual connections.

These primitives—per-device identity keypairs and per-capsule shared keys—are the minimum viable cryptographic infrastructure. They must exist before any capsule can be built, any ensemble can form, or any flow can authenticate its participants. The protocol treats their implementation as a prerequisite, not an extension.

11.2 Pieces

A **piece** is a device considered as a member of one or more device groupings. The word is deliberately neutral: a phone is a piece, a laptop is a piece, an ESP32 is a piece, a robot arm is a piece. What makes something a piece (rather than just "a device") is that it participates in the Rim protocol's device topology—it has been authorized into at least one capsule.

A piece can belong to many capsules simultaneously. Its identity within the protocol is tied to its cryptographic device identity, not to any particular capsule membership.

11.3 Capsules

A **capsule** is a persistent, curated set of pieces that have been authorized to work together. The term is used in the fashion sense: a capsule wardrobe is a set of items designed or curated so they can all work together and mix-and-match. A device capsule is a set of devices authorized to see each other, coordinate, and route traffic among themselves when they happen to be nearby.

Capsules have the following properties:

- **Built incrementally**: Pieces are authorized into a capsule one or a few at a time, using trustworthy out-of-band linking strategies (QR codes, NFC tap, proximity-based challenge-response, or similar). The authorization step establishes mutual trust between the new piece and the existing capsule membership.
- **Relatively persistent**: A capsule is meant to be a stable, long-lived grouping. You build it over time as you acquire or designate devices for a purpose.

- **Retired whole**: In the style of proactive security, the preferred response to a compromised or lost piece is to retire the entire capsule and build a new one, re-authorizing each remaining piece. This is analogous to key rotation: you do not subtract from a capsule, you replace it. Subtraction would leave ambiguity about what the removed piece still knows or can do.
- **Multi-membership**: A piece can belong to many capsules. A phone might be in a "daily carry" capsule with a watch and earbuds, a "home studio" capsule with a laptop and audio interface, and a "lab" capsule with a set of sensors.

11.3.1 Capsules as Pre-Flow Infrastructure

Capsules are not flows. They are infrastructure that flows depend on. A flow needs a capsule to know who the participants are, what keys to use for encrypted communication, and what the ensemble topology looks like. Making capsules a flow would create a bootstrapping dependency: you would need a working flow evaluation to load the capsule that tells you who to sync flows with.

A capsule's piece set is add-only: pieces are authorized in but never individually removed (the capsule is retired whole instead). This means the merge function is set union—the simplest possible convergent data structure. Two pieces connect, exchange their piece sets, take the union, done. No operation logs, no causal tracking, no conflict resolution beyond deduplication by device identity. This simplicity is deliberate: it means the same data structure works on a full device and on a microcontroller with kilobytes of RAM.

11.4 Ensembles

> **Naming: Ensemble vs. Outfit**
>
> The active-subset concept needs a name with the right connotations. Two candidates:
>
> **Ensemble** is neutral, with a dual meaning as a collection of things considered as a whole (particularly musicians playing together).
>
> **Outfit** is construable as a group of people undertaking a particular activity together, with a secondary fashion meaning.
>
> This document uses *ensemble* as a placeholder. The final choice is pending.

An **ensemble** is the dynamic subset of a capsule's pieces that are currently online and actively coordinating with each other. Where a capsule is a persistent authorization list, an ensemble is a live, moment-to-moment reality: who is actually here right now, responding, and participating in traffic routing.

Ensembles have the following properties:

- **Real-time membership**: Pieces join and leave ensembles as they come online, move into range, or go to sleep. Every participating piece tracks the current ensemble membership with both additions and subtractions. A piece is a member of the ensemble if it is *reachable*—directly or through intermediaries—not only if it has a direct transport-layer connection (see §11.8).
- **Shared topology picture**: All pieces in an ensemble maintain a shared understanding of the current network topology—who is responding, what the connectivity graph looks like, and what routes are available. This picture is maintained via encrypted advertisements and broadcasts over BLE (or whatever the transport layer provides).

- **Associated data structures**: An ensemble may have directed multigraph data structures and synchronization states associated with it, representing the live routing topology, pending data transfers, and coordination state. These structures are maintained collaboratively by all participating pieces.
- **Scoped to a capsule**: An ensemble is always a subset of exactly one capsule. A piece that belongs to multiple capsules may participate in multiple ensembles simultaneously, but each ensemble draws its membership from a single capsule's authorization list.

11.5 Parures

A **parure** is a set of pieces that are integrated at the hardware or firmware level, intended to be together from the beginning. The term comes from jewelry: a parure is a matched set of pieces (necklace, earrings, bracelet, brooch) designed and sold together.

In the Rim protocol, a parure is a set of devices that:

- Were sold together, or are part of the same hardware ecosystem
- Communicate via specialized, possibly proprietary protocols between themselves
- Can be assumed to have solved the internal routing problem—they know how to talk to each other efficiently without the general-purpose Rim discovery and topology mechanisms
- Effectively form a *clique* whenever they appear together in an ensemble: every piece in the parure can reach every other piece in the parure directly, with known latency and bandwidth characteristics

A parure simplifies the ensemble's routing problem. Whenever two or more members of a parure appear in an ensemble, the topology tracker can treat them as a pre-solved clique—no discovery or route-finding needed among them—and focus routing decisions on the links between the parure cluster and the rest of the ensemble.

The near-term hardware development path—starting with individual ESP32-S3 microcontrollers as accessories—is heading toward parures: matched sets of custom hardware designed and built together, communicating over shared protocols, forming a pre-solved clique in any ensemble they join.

11.6 Accessories

An **accessory** is a device that can store data or serve some simple, singular role but is not expected to manage connections in a sophisticated way. Where a full piece runs the Soradyne/Rim core libraries and participates in ensemble topology tracking and routing decisions, an accessory implements a minimal interface.

Examples of accessories:

- A microcontroller with storage (e.g., an ESP32 with an SD card) that holds dissolution shards
- An actuator that receives commands from a flow
- A simple bridge to a different device or protocol (e.g., a BLE-to-serial adapter)

Most accessories participate in the transport fabric like any other piece—they route traffic through themselves when they sit between two other pieces in the topology (see §11.8). Routing requires only forwarding opaque envelopes based on destination headers, which is cheap enough for most microcontrollers. A few genuinely constrained devices (e.g., an actuator with a unidirectional radio, or a sensor that can only transmit) may lack the capability to route; this is the exception, not the rule. The key design constraint is that implementing an accessory should be straightforward: a small interface, minimal dependencies, no requirement for the full Soradyne library stack. This makes it feasible to build accessories on resource-constrained hardware.

11.7 Transport Philosophy

Why BLE First

Most networking protocols treat IP-based transports (TCP sockets, HTTP, WebSockets) as the baseline and wireless or hardware-specific transports as extensions. Rim inverts this deliberately.

BLE is the prototypical transport because it embodies the physical reality the protocol is designed for: devices on or near a person's body, communicating over short distances, with constrained bandwidth and power budgets, using broadcast-capable radios. The protocol is locality-centered, body-centered, fashion-centered, and hardware-centered.

A stream's baseline implementation targets BLE or simulated-BLE semantics. Extending or replacing this with TCP, UDP, or other IP-based transports is possible but represents *additional* development effort, not the default path. This ensures that the simplest, most natural deployment—a person's devices talking to each other in proximity—works without any network infrastructure.

Ensemble topology maintenance is expected to rely heavily on BLE's broadcast and advertising capabilities:

- **Encrypted advertisements**: Pieces advertise their presence and capsule membership via encrypted BLE advertisement packets. Only pieces holding the capsule's shared key material can decode these advertisements.
- **Topology synchronization**: When pieces discover each other via advertisements, they establish connections and synchronize their view of the ensemble's directed multigraph—who can reach whom, through what paths, at what quality.
- **Graceful extension**: An implementation may extend a BLE-based ensemble with IP-based links (e.g., when two pieces are on the same WiFi network, or when a cloud relay is available). These extensions layer on top of the BLE-first topology, they do not replace it.

11.8 Routing as Transport Fabric

Routing data through intermediate pieces is a **transport-layer concern**, not a flow role. Every piece in an ensemble participates in routing: if piece A can reach piece B, and piece B can reach piece C, then B routes traffic between A and C transparently. This is analogous to how routers in IP networks forward packets without knowledge of the application-layer content they carry. A piece functioning as a relay has no responsibilities regarding the data it shuttles—it need not parse, store, or understand the flow-level content passing through it.

This is a deliberate architectural separation. Contrast with *memorization*, which is a flow role (§5): a memorization role stores data and makes it retrievable later, which requires understanding the flow's data model (at least enough to serialize and deserialize). A memorization role's implementation might involve multiple hops across routing nodes—the point-to-point character of "store this, recall it later" is abstract from the physical path the data takes through the network. The relay nodes on that path are part of the transport fabric, not participants in the flow.

11.8.1 Logical Addressing and Message Envelopes

The mechanism that makes routing invisible to flows is **logical addressing**. Messages between pieces are wrapped in envelopes that carry a source identifier, a destination identifier, and a time-to-live (TTL) counter. An intermediate piece receiving an envelope checks the destination: if it matches the local piece, the message is delivered locally; otherwise, the envelope is forwarded to the next hop with a decremented TTL. The intermediate piece never inspects the payload—it operates entirely on the envelope headers.

This means a flow sending data to a remote piece uses the same interface regardless of whether the destination is one hop away or three: *send to this UUID*. The messaging layer consults the ensemble's directed multigraph to determine the next hop, handles forwarding through intermediaries, and uses the TTL to break routing loops. Broadcast messages (destination = all) are forwarded to all neighbors, propagating through the mesh until TTL is exhausted.

11.8.2 Reachability vs. Direct Connection

An important consequence of multi-hop routing is that a piece's presence in the ensemble is determined by **reachability**, not by having a direct transport-layer connection. A piece is part of the ensemble if messages can reach it—whether directly (one hop) or through intermediaries (multiple hops). The ensemble's topology tracks both cases: for directly connected pieces, the next hop is the piece itself; for indirectly reachable pieces, the next hop is the neighbor that sits on the path toward them.

This distinction matters because ensembles are often not fully connected. In a three-piece capsule where the phone and the accessory can each reach the laptop but not each other, both the phone and the accessory are full ensemble members—the laptop routes traffic between them. Neither the phone nor the accessory needs to know that the other is only indirectly reachable; from their perspective, messages simply arrive.

Routing decisions are made by the ensemble's topology manager based on the directed multigraph of connectivity. The topology manager finds paths, manages retries, and handles link failures. From the flow's perspective, data written to a stream simply arrives at its destination; the hops in between are invisible.

12 Terminology Reference

Term	Definition
Self-Data Flow (or SDFlow)	A persistent, typed, UUID-identified bundle of streams with policies for data movement, transformation, delegation, and exceptions. The core abstraction of the Rim protocol.
Stream	The basic I/O abstraction within a flow. Named, typed, with read/write/subscribe interface. All data enters and leaves a flow through streams.
Drip	A stream that provides eventually consistent, authoritative data. Slow, deliberate, consensus. Named after the slow, steady nature of a water drip.
Jet	A stream that provides fast, possibly lossy, real-time data. Prompt, per-observer, non-consensus. Named after a fast water jet.
Fitting	A transformation connecting streams to each other within a flow. A fitting's functionality is real and implemented in real code, but a fitting is not a first-class named entity at the same level as a stream—it is implicit in its implementation.
Flow Role	A named set of responsibilities within a flow. Filled by parties (devices, processes) at runtime. A device can fill many roles; many devices can fill the same role. Examples: data source, data sink, memorization, curator, aligner. Often shortened to "role" in discussion.
Policy	A rule encoded in a flow's type definition or configuration governing stream behavior under various conditions: errors, delegation, discovery, memorization, retirement.
Flow Type	A versioned design document specifying a flow's streams, roles, policies, and application interface. Identified by the content hash of its canonical serialization. Defined at the application level.
Flow Configuration	Instance-specific parameters for a flow: participating parties, network addresses, storage quotas, etc. Stored on-device, keyed by flow UUID.
Data Geometry	The description of a data type's spatial, temporal, and structural character: query coordinates, data coordinates, their discreteness/continuity/orderedness, and supported transforms.
Query Coordinates	The dimensions along which data is requested from a stream. Define the address space.
Data Coordinates	The dimensions of values returned at each query point. Define the value space.
Memorization	A flow role responsible for reliably storing and recalling stream data. Subject to security policies (plaintext, encrypted, dissolved) and robustness policies (ephemeral, replicated, reallocating).

Term	Definition
Dissolution	Splitting data across multiple devices using erasure coding so that no single device holds enough to reconstruct it.
Crystallization	Recombining dissolved data when enough devices come together.
ConvergentDocument	A CRDT engine in soradyne_core used to implement drip backing stores. Generic over a schema that defines item types and fields.
DataChannel	A concrete stream implementation for in-memory pub/sub. Formerly called `SelfDataFlow<T>`. One way to implement a stream, not a flow.
Piece	A device considered as a member of one or more capsules. Any device participating in the Rim protocol's device topology.
Capsule	A persistent, curated set of pieces authorized to work together, built incrementally via trustworthy linking (QR codes, NFC, etc.). Retired whole rather than shrunk, in the style of proactive security. A piece can belong to many capsules.
Ensemble	The dynamic, real-time subset of a capsule's pieces that are currently online and coordinating. Membership tracked with additions and subtractions by all participating pieces. May have associated directed multigraph data structures. (Name pending; see §11.4 for alternatives.)
Parure	A set of pieces integrated at the hardware or firmware level, intended to be together from the start (e.g., sold together, same ecosystem). Form a clique when present in an ensemble, with pre-solved internal routing.
Accessory	A device serving a simple, singular role (storage, actuation, bridging) without full topology management. Implements a minimal interface; does not require the full Soradyne library stack.
Routing	Transport-layer forwarding of data through intermediate pieces. Not a flow role—all pieces route transparently, like IP routers, without knowledge of flow-level content. Messages are wrapped in envelopes with source, destination, and TTL; intermediate pieces forward envelopes without inspecting payloads. Distinct from memorization, which is a flow role requiring data-model awareness.

13 Appendix: Comparison with Nestbox Twigs

The Nestbox project's Twig system provides a useful point of comparison, as it implements a subset of what self-data flows generalize.

Concept	Twig	Self-Data Flow
Identity	`stream_id` + `coord_sys_id`	Flow UUID
Schema	Protobuf message definition	Flow type (design document, content-hashed)
Routing	`SampleRouter` with JSON config	Fittings defined by flow policies
Data format	Fixed protobuf (`MeasurementSet`)	Data geometry described in type
Dimensions	Enum (`X, Y, Z, T, VX, ...`)	Query/data coordinate specification
Uncertainty	`CovarianceMatrix`	Precision matrices (recommended)
Transforms	`TransformationMatrix` per measurement set	Part of query interface
Discovery	Bespoke per deployment	Flow discovery policy
Memorization	None (live only)	Memorization roles with policies
Multi-device	TCP + protobuf, per-connection	Flow bundles all streams, multi-transport

The Twig system does its job well for the specific Nestbox use case. Self-data flows generalize the pattern so that each new device type or application does not require new bespoke plumbing.

14 Appendix: Status of Current Implementation

Component	Status	Notes
`Flow` trait	Implemented	Bootstrap-from-UUID model. Directionally correct.
`FlowRegistry`	Implemented	HashMap-based. Placeholder for on-device storage.
`FlowConfigStorage`	Implemented	In-memory. Needs persistent backend.
`Stream` trait	Implemented	read/write/subscribe interface.
`StreamSpec`	Implemented	drip/jet/singleton/per-party.
`BasicFlow`	Implemented	Schema-validated stream registration.
`DataChannel<T>`	Implemented	Former `SelfDataFlow<T>`. In-memory pub/sub.
`ConvergentDocument`	Implemented	CRDT engine. Giantt and Inventory schemas.
Flow type definitions	Not started	Design documents per this spec. Detailed implementation plan exists (DripHostedFlow, image/composite/album flow types).
Real I/O through flows	Not started	Streams are in-memory only. Plan specifies BLE GATT-based sync.
Dissolution/crystallization	Partial	Reed-Solomon in storage module, not wired to flows.
Nestbox integration	Not started	Twig replacement via spatial measurement flow type.
Cryptographic identity	Not started	Plan specifies Ed25519 + X25519 per-device identity, capsule shared keys. Positioned as Phase 0 (prerequisite for all else).
Device topology (capsules)	Not started	Plan specifies add-only set with gossip merge. Pre-flow infrastructure (not a CRDT, not a flow).
Ensemble tracking	Not started	Plan specifies directed multigraph, encrypted BLE advertisements, topology sync protocol, and a logical messaging layer for multi-hop routing (flows address peers by UUID; routing is invisible).
BLE transport	Not started	Plan specifies btleplug (Rust-native) on all hosted platforms, simulated BLE for testing. BLE-first, transport-agnostic to frontends.