RAPP REV-5 · PROGRAMMING WORKBOOK EDITION

The RAPP Programming Language

A tutorial, programming workbook, and reference manual for verifiable agents.

THE COMPLETE SINGLE-VOLUME EDITION

The RAPP Programming Language

A tutorial, programming workbook, and reference manual for agents that keep a verifiable memory and communicate over one exact wire.

RAPP REV-5 · GENERATED FROM THE CANONICAL BOOK SOURCES

About this edition

This volume is generated from the same chapter files as the online reader. The normative authority is SPEC.md; the book explains the standard but does not replace it.

To copy any code example exactly, open the interactive HTML edition.

Source, examples, conformance vectors, and current estate evidence: immutable source snapshot.

The protocol is meant to be implemented.

CONTENTS

The complete volume

APPENDIX C Selected Exercise Solutions

The RAPP Programming Language

A tutorial and reference manual for verifiable agents

Written against RAPP rev-5. Every code fragment in this book runs on `rapp.py`*, the stdlib-only reference implementation that ships beside it, and every claim about "the real world" is checked by* `realcheck.py` *against the actual committed artifacts of a live estate.*

Preface

RAPP is a programming language for **agents that keep a verifiable memory and talk over one wire**, and this book teaches it. A RAPP program is not a source file full of expressions. It is a durable sequence of canonical values, addressed events, identity transitions, and portable packages whose meaning another implementation can verify.

Its executable grammar is a *protocol*, in the sense that HTTP, JSON, and git's object model are protocols: a small number of exact rules that let independent programs, written by people who never met, produce bytes the other side can trust. RAPP is not a general-purpose replacement for Python or JavaScript. Those languages implement agents; RAPP is the language in which their durable behavior crosses runtimes. It gives us a way to write down what an agent did, address it by its content, chain it into a biography, and hand it to anyone.

The protocol is built from **five primitives**, and this book is organized around them:

1. **Canonicalization** — turning a value into exactly one sequence of bytes (chapter 2).
2. **Content addressing** — naming those bytes by their hash, with domain separation (chapter 3).
3. **Identity** — the `rappid`, a name minted once and never a hash of a name (chapter 4).

4. **The frame** — one record that is both a *particle* (a link in a worldline) and a *wave* (an integrity-checked unit on the wire) (chapter 5).
5. **The egg** — a content-addressed package that carries an organism or an application (chapter 7).

Everything else — the `/chat` endpoint (chapter 6), conformance classes, versioning — is how those five are carried and governed.

The layered map

The primitives are small because each layer has one job and may not redefine the layer beneath it:

```
L5  EGG       packages a portable unit
L4  FRAME     records one immutable event
L3  WIRE      carries requests and append-only frames
L2  IDENTITY  names actors and binds keys
L1  ADDRESS   turns values and octets into stable names
```

You can use only the lower layers — content-address a payload without ever packing an egg — but you cannot skip them. A frame that invents its own canonicalizer is not a second kind of frame; it is a different protocol wearing the same word.

Why this book exists

The RAPP ecosystem is real and it drifted. The same concept — "a frame," "a rappid" — got implemented more than once, in incompatible ways, each copy claiming the same name. A frame was minted twice under one version string with two different hash rules. An identity was computed three different ways in production, one of them the cardinal sin of hashing a *name* into an address. This is not exotic; it is the oldest failure in distributed systems, and it has been solved before — by Linux's one-mainline rule, by the Web's single living standard, by git making the hash the name. RAPP is the convergence: **one spec, one canonicalizer, one mint, one frame.** This book teaches that spec so completely that the drift cannot come back, because everyone building on it turns the same bytes into the same tree.

How to read it

If you have written a little Python and seen a hash function before, you can read this book start to finish. Chapter 1 is a fast, complete tour — by the end of it you will have built and verified a real chain of frames. Chapters 2 through 7 take the five primitives one at a time. Chapters 8 and 9 add signatures, authority, evolution, and the security boundaries hashes cannot cross. Chapter 10 follows a live estate from a drifted baseline to verified convergence. Chapter 11 turns the whole dependency graph into an implementation plan. Appendix A is the terse reference; Appendix B is the vocabulary and failure atlas; Appendix C contains selected exercise solutions.

Run everything. The reference implementation is 140 lines; you are meant to read it, and the book will tell you when. A protocol you have only read about is a rumor. A protocol whose conformance suite you have watched go green, against your own bytes, is a tool.

Who this book is for

READER	WHAT THE BOOK GIVES YOU
agent builder	a portable record and package model that does not depend on one runtime
protocol implementer	byte rules, verification order, and conformance boundaries
estate operator	head, key, registry, migration, and fail-closed discipline
security reviewer	an explicit separation of integrity, authorship, authority, and freshness

You do not need prior cryptography work. The book explains what each primitive proves and, just as importantly, what it does not prove.

The contract with the standard

This book is explanatory. `SPEC.md` is normative. The distinction matters:

- the standard says what every implementation **must** do;
- the reference profile makes the byte and package core executable;

- the conformance suite proves selected interoperability claims; and
- the book supplies intuition, worked observations, and operating consequences.

If a sentence here cannot be traced to a normative rule or clearly marked as application guidance, it is a book bug.

Let’s build a frame.

Chapter 1 — A Tutorial Introduction

> ***In this chapter:*** *mint one identity, build one frame, extend it into a worldline, and make the verifier reject three different attempts to rewrite the past.*

Let us begin with the smallest complete RAPP program that does something real: it creates a record, addresses it by its content, and verifies it. In the tradition of the book this one is named after, we do not start by explaining every rule. We start by making something work, and then we go back and understand why it worked.

1.1 Getting Started

Everything in this chapter runs against `rapp.py`, which needs nothing but a Python 3 standard library. From the repository root:

```
python3 examples/01_hello_frame.py
```

Here is the program:

```
import rapp as R

stream = R.mint_rappid("kody", "hello")          # 1. an identity
for the stream
frame = R.build_frame(                             # 2. a record in
that stream
    kind="note.write",
    stream_id=stream,
    seq=0,
    utc="2026-07-15T12:00:00.000Z",
    payload={"text": "hello, frame"},
    prev=None,
)
ok, step, why = R.verify_frame(frame, head=None,
stream_id_of_record=stream)  # 3. check it
print("OK" if ok else f"FAILED at {step}: {why}")
```

Three calls: mint an identity, build a frame, verify it. That is a complete transaction in RAPP. When you run the example, it prints the whole record, and the last line prints `OK`.

The record it built looks like this:

```
{
  "spec": "rapp/1",
  "kind": "note.write",
  "stream_id": "rappid:@kody/hello:b62415b2…",
  "seq": 0,
  "utc": "2026-07-15T12:00:00.000Z",
  "payload": { "text": "hello, frame" },
  "payload_hash": "3e04704309e6069a…",
  "prev": null,
  "prev_wave": null,
  "sig": null,
  "frame_hash": "d82aea005209630d…"
}
```

Eleven fields, always exactly these eleven. Two of them are hashes the library computed for you: `payload_hash` and `frame_hash`. Understanding what those two are — and why there are *two* — is most of understanding RAPP.

1.2 The Particle and the Wave

Look at the two hashes. `payload_hash` is the SHA-256 of the payload alone. `frame_hash` is the SHA-256 of the whole frame (minus the signature and minus itself). They answer two different questions, and a frame is deliberately built to answer both at once.

`payload_hash` is what we call the **particle**. It is the address of *what happened* — the content of this moment, independent of where it sits. When you chain frames into a biography, each frame points back to the previous frame's particle. The particle is the worldline.

`frame_hash` is the **wave**. It is the address of *this exact record on the wire* — envelope and all: the seq, the timestamp, the stream it belongs to. When a frame travels across a network and you need to know it arrived unaltered, byte for byte, you check the wave.

A frame is one object that is both. You do not choose between them when you emit; you emit one frame and the reader observes whichever address the situation calls for. This is the resolution of a real bug in the ecosystem's history, where two teams built "the frame" — one hashing the payload, one hashing the whole envelope — under the same version number, and the two could never read each other. RAPP's frame carries both hashes so the question "which one is *the* hash?" never has to be asked. It is a particle when you follow the worldline and a wave when you check the wire. (Chapter 5 is entirely about the frame; this is the intuition.)

1.3 Chaining: a Worldline

One frame is a fact. A *chain* of frames is a biography, and it is tamper-evident. Run:

```
python3 examples/02_build_a_chain.py
```

The core of it is a loop where each frame's `prev` is the previous frame's particle:

```
chain, head = [], None
for seq, (utc, payload) in enumerate(events):
    prev = head["payload_hash"] if head else None
    fr = R.build_frame("diary.entry", stream, seq, utc, payload,
prev=prev)
    ok, step, why = R.verify_frame(fr, head=head,
stream_id_of_record=stream)
    assert ok
    chain.append(fr); head = fr
```

`verify_frame` here is doing chain checks as well as content checks: it insists that `seq` is contiguous, that `prev` equals the head's particle, and that time does not run backwards. Three frames in, you have a verified worldline.

Now watch it defend itself. Suppose an attacker wants to rewrite the past — change what the first frame said. The example does exactly that, in the smartest way available to the attacker, and the chain still catches it:

```
naive edit of frame 0's payload → frame 0 verify: REJECTED at step
2 — payload_hash mismatch  ✓
attacker fixes frame 0's hash   → frame 0 verify: frame_hash
mismatch
…but the chain link at frame 1  → verify: REJECTED at step 4 — prev
!= head payload_hash  ✓
```

Three layers, and you have to beat all three. Edit the payload and the *particle* no longer matches (step 2). Recompute the particle to cover that, and now the *wave* no longer matches (step 3), because the frame_hash covered the old particle. Recompute the wave too, and you have a self-consistent forged frame 0 — but its new particle is not what frame 1 recorded as its `prev` (step 4), and to fix *that* you must forge frame 1, and frame 2, and every frame to the head. The chain converts "rewrite one moment" into "rewrite the entire history from that moment forward," which is exactly the property we wanted.

1.4 Identity, Done Once

We called `mint_rappid("kody", "hello")` and got back a string like `rappid:@kody/hello:b62415b2…`. The human-readable part — `@kody/hello` — is a convenience. The identity is the 64-hex tail, and where that tail comes from is the single most important rule in RAPP's identity system:

> *The tail is minted **once**, from entropy or from a public key. It is **never** the hash of the name.*

Run `python3 examples/03_identity.py` and it will show you the forbidden thing explicitly: `sha256("kody/twin")` is a tail that *anyone* who names something `kody/twin` will compute — it collides by construction. That exact mistake was live in production in this ecosystem, an identity derived as `sha256("<owner>/<slug>")`, and chapter 4 is about why it is fatal and how mint-once fixes it. For now, the rule: names are for humans; the minted tail is the identity.

1.5 Where We Are

In one chapter you have built a frame, understood it as both particle and wave, chained frames into a tamper-evident worldline, watched the chain reject a forgery, and met the identity rule. That is the spine of the protocol. Everything from here is precision:

- **How** exactly a value becomes bytes, so two implementations agree to the last byte (chapter 2).
- **How** the hashes are domain-separated so a payload address can never be confused with a frame address (chapter 3).
- The **full** identity grammar and the three lawful ways to re-anchor one (chapter 4).
- The **complete** frame: all eleven fields, the verify checklist, forks and re-genesis (chapter 5).
- The **wire** that carries frames — `POST /chat` — and the swarm streams (chapter 6).
- The **egg**, which packages an entire organism into one content-addressed file (chapter 7).

- **Trust and signatures** — what a key proves, and how it is discovered (chapter 8).
- **Registry and evolution** — current heads, owner succession, and lawful migration (chapter 9).
- And then chapter 10, where we follow a real estate from eight drift findings to 46 verified frames and watch the language tell the difference byte by byte.
- Finally, **implementation** — how to build the language from safe values through atomic append and independent interop (chapter 11).

Read `rapp.py` now if you like — it is short, and you have already used most of it.

1.6 Checkpoint: Make Each Layer Fail

Run the chain example once unchanged, then make one edit at a time:

1. change only `payload`;
2. recompute `payload_hash` but leave `frame_hash` alone;
3. recompute both hashes on the old frame but leave the next frame's `prev` alone; and
4. copy a valid genesis frame into a reader expecting a different `stream_id`.

The failures should move through steps 2, 3, 4, and 1a. That order is the architecture in executable form:

```
value → particle → wave → chain → stream binding
```

Do not "fix" a received frame while testing. Build a new candidate and let the consumer refuse the old one. Verification that mutates its input destroys evidence.

1.7 Exercises

Exercise 1-1. Add a fourth event to `examples/02_build_a_chain.py`. Print the complete particle and wave for each frame, then explain which address appears in the next frame.

Exercise 1-2. Construct one fresh mutation for every verifier step from 1 through 6, including 1a. Assert the expected step instead of matching only the reason text. *A selected solution appears in Appendix C and* `examples/05_failure_atlas.py`.

Exercise 1-3. Serialize the valid chain to files, load it into a new process, and verify from genesis to head without reusing any in-memory object from construction.

Exercise 1-4. Assume a hostile mirror serves a self-consistent replacement chain. Write down the minimum trusted state a returning consumer needs to detect the replacement.

1.8 Chapter Summary

- A RAPP history begins with a minted identity and a sequence-zero frame.
- The particle addresses the payload; the wave addresses the unsigned envelope.
- `prev` links each new frame to the previous particle.
- Rewriting one old event requires rewriting every descendant, and a remembered head exposes the replacement.
- Names help humans locate an identity; the minted tail supplies continuity.

Chapter 2 — Canonicalization

> ***In this chapter:*** *see why semantic equality is not byte equality, inspect canonical bytes, understand the reference profile's number boundary, and learn which "helpful" normalizations a conformant consumer must refuse.*

Every hash in RAPP is a hash of *bytes*. But agents exchange *values* — objects, arrays, strings, numbers. Between a value and its hash sits a question that has sunk more distributed systems than any other: **which bytes?** `{"a":1,"b":2}` and `{"b":2,"a":1}` are the same value and different bytes. If two implementations disagree about which byte string represents a value, they compute different hashes for the same content, and every downstream promise — content addressing, chaining, signatures — silently breaks.

Canonicalization is the rule that makes the answer unique. RAPP §4 adopts **RFC 8785, JSON Canonicalization Scheme (JCS)**, because this is a solved problem and inventing a fourth JSON canonicalizer is exactly the kind of drift this protocol exists to end.

2.1 The Rules

A canonical RAPP value is **I-JSON** (RFC 7493) serialized by JCS. In practice:

- **Object keys are sorted** by their UTF-16 code units, ascending.
- **No insignificant whitespace.** `{"a":1}`, never `{ "a": 1 }`.
- **Strings** use the shortest escaping; non-ASCII is emitted as raw UTF-8, not `\uXXXX`.
- **Duplicate keys are forbidden.** An object with two `"a"` keys is not a value; it is an error.
- **Arrays keep their order.** Order is significant in an array and insignificant in an object, and canonicalization respects exactly that distinction.

The reference implementation is a direct transcription of these rules:

```
def canonical(v):
    if v is None or isinstance(v, bool):   return json.dumps(v)
    if isinstance(v, int):                 return json.dumps(v)
    if isinstance(v, float):               raise ValueError("floats
need full JCS number form")
    if isinstance(v, str):                 return json.dumps(v,
ensure_ascii=False)
    if isinstance(v, list):                return "[" +
",".join(canonical(x) for x in v) + "]"
    if isinstance(v, dict):
        keys = sorted(v.keys())
        if len(keys) != len(set(keys)):    raise
ValueError("duplicate keys")
        return "{" + ",".join(json.dumps(k, ensure_ascii=False) +
":" + canonical(v[k])
                              for k in keys) + "}"
    raise ValueError("non-I-JSON value")
```

You can watch the property it guarantees — the same value canonicalizes identically regardless of how it was constructed:

```
>>> R.canonical({"b": 1, "a": [3, 2]}) == R.canonical({"a": [3, 2],
"b": 1})
True
>>> R.canonical([1, 2]) == R.canonical([2, 1])
False
```

Key order is erased; array order is preserved. This is conformance vector V1, and it is the foundation the whole tower stands on.

2.2 Numbers, and Why the Reference Profile Has None

Numbers are where JSON canonicalization gets genuinely hard. Is `1`, `1.0`, `1e0`, and `10e-1` the same number? RFC 8785 specifies an exact IEEE-754 serialization (the ECMAScript `Number.prototype.toString` algorithm) so that every binary64

value has one canonical form. A production RAPP implementation MUST implement it, and the test is a round-trip: `0.1` must survive canonicalization unchanged.

The **reference profile** in `rapp.py` deliberately refuses floats and accepts only exact integers, strings, booleans, null, arrays, and objects. This is not a weaker canonicalizer; it is the same canonicalizer over the value domain where the answer is unambiguous on every platform. The reference vectors use integer payloads so the published hashes are reproducible byte-for-byte anywhere, on any language, without depending on a float-formatting library. When you need floats in real payloads, implement RFC 8785 §3.2.2 and keep the round-trip test in your conformance suite. When you can express a quantity as an integer or a decimal string, do — it is one less thing that can differ between two honest implementations.

2.3 What Canonicalization Does Not Do

Two temptations, both refused, both for the same reason: they make the same bytes hash differently on different machines.

- **No Unicode normalization (no NFC) for new content.** It is tempting to NFC-normalize strings so that visually identical text hashes identically. RAPP does not, for new content: NFC behavior varies across library versions, so folding it into canonicalization would make the hash depend on which Unicode table you linked against. The rule is: the bytes you put in are the bytes that are hashed. Normalize *before* you hand a value to the protocol if your application needs it.
- **No schema coercion.** Canonicalization does not know or care what a field "should" be. It serializes the value it is given. `"1"` (string) and `1` (integer) are different values with different canonical forms and different hashes, and that is correct.

2.4 The Payoff

Because canonicalization is exact and shared, everything above it can be exact and shared. The first chapter 10 audit ran the reference `canonical()` against 32 historical frames written by a different program and reproduced every stored

payload hash byte-for-byte. After migration, the current sweep reproduces the domain-tagged addresses of 46/46 frames. That is the whole point: independent producers and consumers agree because JCS has exactly one answer.

2.5 Failure Modes: Refuse, Do Not Normalize

INPUT	WHY IT IS UNSAFE	CONSUMER ACTION
duplicate object member	parsers may keep the first or last value	refuse the whole value
lone UTF-16 surrogate	not interoperable I-JSON text	refuse
non-round-tripping number	two runtimes may see different mathematical values	refuse
value deeper than 64 levels	parser/resource attack surface	refuse
canonical form over 1 MiB	unbounded hashing and memory cost	refuse
visually equal but code-point-different strings	normalization would change addressed bytes	preserve as distinct values

Canonicalization is not cleanup. A consumer that silently repairs an unsafe value may hash bytes the producer never sent and then claim agreement.

2.6 Checkpoint: Inspect the Bytes

From the repository root:

```
python3 - <<'PY'
import rapp as R

a = {"message": "café", "n": 1, "items": [3, 2, 1]}
b = {"items": [3, 2, 1], "n": 1, "message": "café"}

ca, cb = R.canonical(a), R.canonical(b)
print(ca)
print(ca.encode("utf-8").hex())
print("same bytes:", ca.encode("utf-8") == cb.encode("utf-8"))
PY
```

Observe three things: the object construction order disappears, the array order remains, and the non-ASCII `é` is emitted as UTF-8 rather than a `\u` escape.

Then replace integer `1` with float `0.1`. The small reference profile refuses it. That refusal does not mean RAPP forbids all binary64 values; it means a production implementation that accepts them must implement the complete RFC 8785 number form rather than inheriting a language's default formatter.

2.7 Exercises

Exercise 2-1. Predict the canonical text for five values before running the code: an empty object, an empty array, two objects with reversed construction order, and two arrays with reversed element order.

Exercise 2-2. Build a fixture table containing the input value, canonical text, and UTF-8 hex for nested values and non-ASCII strings. *A selected solution appears in Appendix C.*

Exercise 2-3. Explain why Python's `bool` being a subclass of `int` can be dangerous in a validator. Find the guards in `rapp.py` that prevent `True` from becoming a valid `seq`.

Exercise 2-4. Write a pre-walk that refuses nesting depth greater than 64 and canonical output larger than 1 MiB without partially accepting the value.

Exercise 2-5. Advanced: compare two full RFC 8785 libraries on the boundary values `0.1`, `-0`, `9007199254740991`, `9007199254740993`, and `1e999`. Record acceptance and canonical bytes.

2.8 Chapter Summary

- Hashes consume bytes, so interoperable hashes require one byte representation per value.
- RAPP uses I-JSON serialized by RFC 8785 JCS.
- Object order is canonicalized; array order is data.
- New strings are created in NFC, but existing values are never normalized during verification.
- The small reference profile accepts exact integers and refuses floats; full producers implement the RFC 8785 binary64 rules.
- Malformed or ambiguous input is refused whole, never repaired.

Next we turn canonical bytes into addresses — carefully, so a payload address cannot be confused with a frame address even when the underlying bytes are identical.

Chapter 3 — Content Addressing

> ***In this chapter:*** *turn canonical values and raw octets into typed addresses, see exactly what domain separation prevents, and learn why stores must key by both space and digest.*

Once a value has exactly one byte representation (chapter 2), we can name it by its hash. This is content addressing, and it is the mechanism that makes "the hash is the name" true: identical content always yields an identical address, so *two things with the same address are the same thing*, and two things that differ anywhere differ in their address. git built its whole object store on this; so does RAPP.

But there is a subtlety that, gotten wrong, reintroduces exactly the collision we are trying to eliminate. That subtlety is the subject of this short, load-bearing chapter.

3.1 The Problem: One Hash Function, Many Meanings

Suppose you hash a payload to get its particle, and you hash a rappid's public key to get an identity tail, and you hash an egg's manifest to get its address — all with plain SHA-256. Now imagine a value that can legitimately appear in more than one of those roles. Its address is the same 64-hex string in every role, because SHA-256 does not know or care what you *meant* the bytes to be. You have built a system where a payload address and an identity tail can collide, not because of a hash weakness, but because you used the raw hash as if it were an address in several distinct namespaces at once.

This is not hypothetical. The disease this protocol treats is precisely "the same derivation used for different jobs." The fix is **domain separation**.

3.2 The Rule: Tag the Space, Then Hash

RAPP §5 defines exactly one hashing construction, and it never hashes canonical bytes bare. It prefixes a **domain tag** — a short string naming the address space — and a newline, then hashes:

```
def H(space, v):                # hash a value
    return sha256(space.encode() + b"\x0a" +
canonical(v).encode("utf-8")).hexdigest()

def Hb(space, b):               # hash raw octets (for keys,
UUIDs)
    return sha256(space.encode() + b"\x0a" + b).hexdigest()
```

The `0x0A` (newline) separator is the same trick git and Nix use: it makes the tag unambiguously delimited from the content, so no tag can be a prefix of another's content. The defined spaces are:

SPACE	ADDRESSES…
`rapp/1:particle`	a frame's payload (the worldline link)
`rapp/1:wave`	a whole frame (wire integrity)
`rapp/1:rappid`	an identity tail (from entropy or a key)
`rapp/1:egg`	one file's raw octets inside an egg
`rapp/1:egg-manifest`	a whole egg through its canonical manifest
`rapp/1:seal`	a terminal re-genesis seal (chapter 5)

Because the tag is part of the preimage, the *same value* produces a *different address* in each space, by construction:

```
>>> val = {"x": 1}
>>> R.H("rapp/1:particle", val)[:8], R.H("rapp/1:wave", val)[:8],
R.H("rapp/1:egg-manifest", val)[:8]
('…', '…', '…')     # three distinct addresses
```

That is conformance vector V2. Reusing identical underlying bytes cannot accidentally turn a payload address into a frame address or an identity, because each role has a different preimage. Address equality across spaces would require an actual SHA-256 collision rather than a type confusion, and SHA-256 collision resistance is an explicit security assumption of the protocol.

3.3 A Consequence Worth Stating

Domain separation means RAPP addresses are **deliberately incompatible** with an untagged `sha256(canonical(value))`. This matters when you meet historical data. In chapter 10 you will see that the estate's baseline frames stored an *untagged* payload hash. The reference `canonical()` reproduced that untagged value exactly — proving the canonicalization agreed — but `H("rapp/1:particle", payload)` is a different 64-hex string, on purpose. The difference is not a bug on either side; it is the §5 hardening. An implementation adopting RAPP tags its hashes; that is part of what "adopting RAPP" means. The current estate now stores the tagged form; the migration was a genuine convergence, not a no-op relabel.

3.4 Why SHA-256

RAPP fixes the hash: **SHA-256**, FIPS 180-4, lowercase hex, 64 characters. Not a menu, not a negotiation. A protocol whose hash is negotiable has, in effect, several protocols, and an attacker who can pick the weakest wins. One hash, everywhere, is the same discipline as one canonicalizer and one frame. If SHA-256 must ever be retired, that is a new major version of the whole protocol — a deliberate, estate-wide, owner-authorized event — not a per-message option.

With canonical bytes (chapter 2) and tagged addresses (this chapter), we have everything needed to name content unambiguously. Next we use that to build the one name that is *not* derived from content at all — because deriving identity from

content is the one place content addressing must not be used.

3.5 Addresses Are Pairs

A 64-hex digest is not a self-describing global identifier. The complete lookup key is:

```
(space, digest)
```

This matters at storage boundaries. A table keyed only by `digest` invites later code to fetch a particle as if it were a wave or a file blob as if it were an egg manifest. Keep the space in the path, column, object key, or API type:

```
objects/rapp-1-particle/<digest>
objects/rapp-1-wave/<digest>
objects/rapp-1-egg/<digest>
```

The exact storage layout is application policy; preserving the pair is protocol safety.

3.6 Checkpoint: Same Value, Different Roles

```
python3 - <<'PY'
import rapp as R

value = {"x": 1}
for space in ("rapp/1:particle", "rapp/1:wave", "rapp/1:egg-
manifest"):
    print(space, R.H(space, value))
PY
```

Run it twice: each line is stable across runs, and all three lines differ from one another. Then change `"x"` to `"y"` and observe that every address changes. Stability comes from canonical bytes; role separation comes from the tag.

3.7 Exercises

Exercise 3-1. Extend the checkpoint to print all value-address spaces for one value and all byte-address spaces for one byte string. State why calling `H` and `Hb` with the same tag is forbidden.

Exercise 3-2. Implement an immutable `Address(space, digest)` and a store that accepts only that type. Prove a particle digest cannot be fetched as a wave. *A selected solution appears in Appendix C and* `examples/04_typed_addresses.py`.

Exercise 3-3. Design a migration for a table currently keyed by bare digest. How will you identify the original space without guessing?

Exercise 3-4. Write a negative test that rejects uppercase, truncated, and 65-character digests before any store lookup occurs.

3.8 Chapter Summary

- RAPP uses SHA-256 with a newline-delimited domain tag.
- `H` addresses canonical values; `Hb` addresses raw octets.
- The same bytes in two roles have different preimages and therefore different expected addresses.
- A store treats `(space, digest)` as the address, never the digest alone.
- Untagged historical hashes may prove canonicalization agreement while still being non-conformant RAPP addresses.

Chapter 4 — Identity: the rappid

> ***In this chapter:*** *distinguish location from identity, mint keyed and keyless rappids, reject the name-hash trap, and follow the only authorized transitions from one identity anchor to another.*

An agent needs a name that stays the same as its content changes. Its biography grows every day; its address must not. This is the one place in RAPP where content addressing is *wrong*, and understanding why is understanding the `rappid`.

4.1 The Grammar

A rappid is a string:

```
rappid:@<owner>/<slug>:<64hex>
```

- `owner` and `slug` are lowercase labels — `[a-z0-9]` with internal single hyphens (the same shape as a DNS label, case-sensitive per RFC 7405).
- `64hex` is the minted tail: exactly 64 lowercase hex characters.

For example: `rappid:@kody-w/rapp-body:324197c16e7e0ca78e19f8a4e1aef76ed34b6694527bb566753c4c89a8ba71f6`.

The reference implementation validates the grammar with one regular expression:

```
_RAPPID = re.compile(r"^rappid:@([a-z0-9]+(?:-[a-z0-9]+)*)/([a-z0-
9]+(?:-[a-z0-9]+)*):([0-9a-f]{64})$")
def rappid_valid(s):  return bool(_RAPPID.match(s))
```

The `@owner/slug` part is **self-locating** — it tells a human (and a resolver) where to look. But it is not the identity. The identity is the 64-hex tail, and everything important is in how that tail is born.

4.2 Mint-Once: the One Rule

> *The tail is minted **once**, from entropy or from a public key. It is **never** a hash of the name, and never recomputed from mutable facts.*

There are exactly two lawful mints, both domain-tagged through the `rapp/1:rappid` space of chapter 3:

```
def mint_rappid(owner, slug, spki_der=None):
    if spki_der is not None:
        tail = Hb("rapp/1:rappid", spki_der)        # KEYED: from
the public key (SPKI DER)
    else:
        tail = Hb("rapp/1:rappid", uuid.uuid4().bytes)  # KEYLESS:
from fresh entropy
    return f"rappid:@{owner}/{slug}:{tail}"
```

- **Keyless.** The tail is `Hb("rapp/1:rappid", uuid4_octets)` — a stable, opaque *join key* anchored on a random UUID (RFC 9562). Use it for organisms that hold no keypair; identity is anchored on entropy, and integrity comes from the frame chain (chapter 5), with a signature optional.
- **Keyed.** The tail is `Hb("rapp/1:rappid", SPKI_DER)` — derived from the DER-encoded SubjectPublicKeyInfo (RFC 5280) of the actor's public key. This tail is *verifiable*: anyone holding the public key can recompute the tail and confirm the binding. Use it whenever the actor signs.

Because the mint is deterministic in its input, a keyed identity is reproducible — mint it twice from the same key and you get the same rappid (conformance vector V3). That is what "minted once" means operationally: not "computed once and stored," but "a function of a fixed anchor, forever."

4.3 Why a Name-Hash Is Fatal

The forbidden mint is `sha256("<owner>/<slug>")`. It is seductive because it needs no state: any program can "recover" the identity from the name. That is exactly the catastrophe. Run `examples/03_identity.py`:

```
FORBIDDEN name-hash tail:
2479029e83eda461795703fae7d1fa790e9c79f3404bb79d81ad1720c155bf69
  → collides for every actor that ever names something 'kody/twin'.
```

If identity is a hash of the name, then identity *is* the name, dressed in hex. Two different agents that happen to choose the same `owner/slug` get the same tail and become, cryptographically, the same agent. Worse, an identity minted this way cannot be bound to a key — there is nothing secret behind it — so it can be impersonated by anyone who can type the name. This precise mistake was live in production in this ecosystem: `_frame.mjs` computed `sha256("<owner>/<slug>")` as an "eternity hex." RAPP §6.2 outlaws it. Names are chosen; identities are minted. They must not be the same operation.

4.4 Re-anchoring: the Closed Set of Cases

A minted tail is immutable. When continuity must move to a new anchor, §6.3 requires an owner-signed registry re-anchor record and limits its `case` to a closed set:

1. `upgrade`. A provisional 32-hex historical identity becomes a conformant 64-hex identity. The old identifier must already resolve to this owner at read time.
2. `rotation`. An uncompromised keyed actor moves to a new SPKI-derived tail. The old key signs the transition as continuity proof.

3. `compromise`. A compromised key cannot safely sign its successor. The estate owner records the transition together with a tombstone for the old rappid.
4. `tag-migrate`. A pre-rev-3 keyed tail made with bare `sha256(SPKI_DER)` moves once to the domain-separated form after the verifier proves the old derivation.

The first, third, and fourth cases rely on estate-owner authority because the old identity cannot provide ordinary continuity proof. The second requires both owner authorization and the old key. Chapter 8 develops the JWS, discovery, tenure, and tombstone checks.

The `_migrated_from` field in an application record is evidence to inspect, not authority to trust. Without the registry authorization, anyone could claim to be the successor of anyone else.

4.5 One Namespace, One Authority

Estate-wide there is one rappid form and one authority over it (the owner). Not a bare `rappid:<slug>:<hash>` in one repo and a self-locating `rappid:@owner/slug:<hash>` in another — one form. Chapter 10's baseline report meets rappids whose tails are 32 hex characters instead of 64 and records whose schema says `rapp-rappid/2.0` instead of `rapp/1`. The reference checker flags them as facts, not value judgments. Its current report then proves that all four inspected identity records completed the migration.

Identity is the hard floor of the protocol. Get the mint wrong and every signature and every chain above it is anchored to a lie. Get it right — mint once, tag the space, never hash the name — and the frame can safely be built on top of it. Which is chapter 5.

4.6 The Identity Lifecycle

```
mint once
   │
   ├── keyless: Hb(rappid, UUIDv4 octets)
   └── keyed:   Hb(rappid, SPKI DER)
   │
   ▼
use stored tail unchanged
   │
   └── exceptional owner-authorized re-anchor
          ├── upgrade
          ├── rotation
          ├── compromise + tombstone
          └── tag-migrate
```

Renaming the human-readable slug does not create identity continuity by recomputation. Moving owners, adopting a key, or changing naming policy needs an explicit application/registry operation whose continuity rules are defined; the protocol never “recovers” identity from the new name.

4.7 Checkpoint: Prove the Name Is Not the Tail

Run:

```
python3 examples/03_identity.py
```

Then mint two keyless rappids with the same owner and slug. Their human-readable prefixes match, but their tails differ because each mint uses fresh entropy. Mint twice from the same SPKI bytes; those tails match because the public key is the stable anchor.

This is the intended split:

INPUT	SAME NAME?	SAME ANCHOR?	SAME IDENTITY TAIL?
two keyless mints	yes	no	no
two mints from one SPKI	yes	yes	yes
raw `sha256(owner/slug)`	yes	no anchor exists	forbidden

4.8 Exercises

Exercise 4-1. Mint two keyless rappids with the same owner and slug, then mint twice from one SPKI fixture. Explain both equality results without using the word "random" alone.

Exercise 4-2. Audit a directory of identity records for the forbidden `sha256(owner/slug)` derivation. Report paths and identifiers; do not rewrite them. *A selected solution appears in Appendix C.*

Exercise 4-3. Design separate `mint` and `load` APIs. Make a failed load impossible to convert silently into a fresh identity.

Exercise 4-4. Draw the authorization evidence required for `upgrade`, `rotation`, `compromise`, and `tag-migrate`. Mark which cases can and cannot provide an old-key signature.

4.9 Chapter Summary

- A rappid contains a self-locating owner/slug and a 64-hex minted identity tail.
- Keyless identity anchors on fresh UUIDv4 octets; keyed identity anchors on SPKI DER.
- Hashing the human name creates a public collision recipe, not an identity.
- Existing tails are reused on read; they are never silently re-minted.
- Re-anchoring is registry-authorized and limited to upgrade, rotation, compromise, or tag migration.

Chapter 5 — The Frame

> ***In this chapter:*** *construct the closed eleven-field envelope, derive particle before wave, follow the six verification steps, detect forks, and converge an immutable legacy chain without rewriting it.*

The frame is the heart of RAPP. It is the record of a single moment in an agent's life: tamper-evident, content-addressed, chained to the moment before it, and verifiable by a stranger. Chapters 2, 3, and 4 exist to make this chapter's object trustworthy. Here we specify it in full.

5.1 The Eleven Fields

A frame is a JSON object with **exactly** these eleven keys — always all of them, never more, never fewer:

```
FRAME_KEYS = {"spec", "kind", "stream_id", "seq", "utc", "payload",
              "payload_hash", "frame_hash", "prev", "prev_wave",
"sig"}
```

FIELD	TYPE	MEANING
`spec`	`"rapp/1"`	the protocol tag; a frame that is not `rapp/1` is not one
`kind`	`"a.b"` string	the event type — `noun.verb`, lowercase labels
`stream_id`	string	the rappid (or `net:` swarm id) this frame belongs to
`seq`	uint53	position in the chain; genesis is `0`, then contiguous
`utc`	fixed 24-char	`YYYY-MM-DDTHH:MM:SS.mmmZ` — millisecond UTC, always Z
`payload`	object	the content of the moment (any I-JSON object)
`payload_hash`	64hex	the **particle**: `H("rapp/1:particle", payload)`
`frame_hash`	64hex	the **wave**: `H("rapp/1:wave", frame\{frame_hash,sig})`
`prev`	64hex | null	previous frame's particle (null only at genesis)
`prev_wave`	64hex | null	previous frame's wave, on swarm streams; else null
`sig`	string | null	detached JWS signature (chapter 8); null if unsigned

The insistence on *exactly* eleven keys is deliberate and it is a lesson from real drift. When a frame may carry arbitrary extra fields, those fields become an unversioned side-channel that two implementations will fill differently, and you are back to two dialects under one name. The frame is closed. New information goes in the `payload` (which is yours to shape) or becomes a new optional field in a new revision of the *one* spec — never an ad-hoc key.

There is also no "absent vs null" ambiguity. `prev` at genesis is present and `null`, not missing. `verify_frame` refuses a frame whose key set is not exactly the eleven — conformance vector V8 — so a reader never has to guess whether a missing field meant null or meant the writer used a different schema.

5.2 Building a Frame: Particle Then Wave

Order matters when you build. The particle is computed from the payload; the wave is computed from the whole frame *including* the particle but *excluding* the wave itself and the signature:

```
def build_frame(kind, stream_id, seq, utc, payload, prev,
prev_wave=None, sig=None):
    payload_hash = H("rapp/1:particle", payload)          # 1.
particle first
    frame = {"spec": "rapp/1", "kind": kind, "stream_id":
stream_id, "seq": seq,
             "utc": utc, "payload": payload, "payload_hash":
payload_hash,
             "prev": prev, "prev_wave": prev_wave, "sig": sig}
    pre = {k: frame[k] for k in frame if k not in ("frame_hash",
"sig")}
    frame["frame_hash"] = H("rapp/1:wave", pre)           # 2. wave
over everything else
    return frame
```

Excluding `sig` from the wave is what lets you sign a frame *after* fixing its content: the signature covers the `frame_hash`, and the `frame_hash` is stable regardless of whether a signature is later attached. Excluding `frame_hash` from its own preimage is the obvious requirement that a hash cannot contain itself.

The build dependency is acyclic:

```
payload ──particle hash──> payload_hash
   │                            │
   └──────── frame fields ──────┘
                │
                └──wave hash──> frame_hash ──optional signing──>
sig
```

5.3 Verifying a Frame: the §7.5 Checklist

A consumer never trusts a frame's own hash fields; it recomputes them. `verify_frame` is the canonical checklist, and it returns *which step* failed so that "reject" is always explainable:

1. **Shape & types.** Exactly eleven keys; `spec == "rapp/1"`; `kind` matches `noun.verb`; `seq` a uint53; `utc` the fixed 24-char form; `payload` an object; the four hash fields the right shape (`prev` / `prev_wave` may be null). *(1a) Stream binding.* If the reader knows which stream this should be, `stream_id` must match — this is what refuses a genuine frame from stream A replayed into stream B (vector V7).
2. **Particle.** `payload_hash == H("rapp/1:particle", payload)`. Recomputed, not trusted.
3. **Wave.** `frame_hash == H("rapp/1:wave", frame\{frame_hash,sig})`. Recomputed.
4. **Chain.** At genesis (`head is None`): `seq == 0` and `prev is null`. Otherwise: `seq` is `head.seq + 1`, `prev == head.payload_hash`, and `utc >= head.utc` (time does not run backward within a chain).
5. **Wire.** On a swarm stream (`net:`) past genesis, `prev_wave == head.frame_hash`; off swarm, `prev_wave` must be null.
6. **Signature.** A swarm frame MUST be signed (vector V9); the cryptographic verification of the JWS itself is chapter 8.

Steps 2 and 3 are why forgery is a whole-chain problem, as chapter 1 demonstrated: to move a past payload you must beat the particle, then the wave, then re-forge every `prev` to the head.

5.4 Streams, Heads, and Forks

A **stream** is an append-only sequence of frames sharing a `stream_id`. Its **head** is the highest-seq frame. Appending means: build a frame whose `prev` is the head's particle and whose `seq` is the head's `seq + 1`, then verify it against the head before you publish.

A **fork** is two frames claiming the same `seq` with the same `prev` — a genuine ambiguity about what came next. RAPP does not pretend forks cannot happen (networks partition); it makes them *detectable*. A consumer refuses both branches beyond the fork point. The stream authority resolves the history through the

registry and, when needed, re-genesis (chapter 9); it never silently overwrites one addressed branch. A reader who has both frames can see the fork exactly because both are content-addressed.

5.5 Re-genesis: Converging an Immutable Chain

Here is the hard case the protocol takes seriously. A chain is immutable by design — that is its value. So what happens when a chain must change *form*? Chapter 10's historical baseline used a legacy envelope. Those addressed bytes could not be edited into RAPP shape, because editing an immutable chain is a contradiction. The answer is **re-genesis** (§7.6 / §12.1):

1. Compute a terminal seal over the **exact octets** of the old head as they will be retained.
2. Emit a **new** sequence-zero frame in full RAPP form. Its registered family-specific `*.re-genesis` kind and payload cite the old stream and terminal seal; its signature is by the estate owner.
3. Append a registry `genesis` entry for the new `frame_hash` and deprecate every older genesis for that stream. This registry append is the point at which the new chain becomes current.
4. Retain the old frames bit-exact under `legacy/`, never serving or extending them as a live chain.

Re-genesis is how "no legacy, converge and delete" (chapter 9) coexists with "an immutable chain is immutable": you do not mutate the old chain, you seal it and are reborn cleanly in the current form. Each such rebirth is owner-authorized — it is not something an automated sweep may do, because it is a statement about identity continuity, and only the estate owner speaks that.

5.6 Checkpoint: Build a Failure Matrix

Start with a verified genesis frame and make a fresh copy for each mutation:

MUTATION	EXPECTED REFUSAL
remove `prev_wave`	step 1, wrong key set
change `stream_id` in place	step 3, wave mismatch
present it as another stream without changing bytes	step 1a, stream binding
change `payload`	step 2, particle mismatch
change `utc`	step 3, wave mismatch
set genesis `seq` to 1 and rebuild hashes	step 4, genesis rule
set non-swarm `prev_wave`	step 5, wire rule
build an unsigned `net:*` frame	step 6, signature rule

The ordering is useful operationally. A step-2 failure says the payload address is false; a step-3 failure says the envelope address is false; a step-4 failure can involve two individually valid frames that do not form one history.

5.7 Exercises

Exercise 5-1. Run `examples/05_failure_atlas.py`, then add one malformed `kind`, one boolean `seq`, and one invalid UTC. All three must stop at step 1.

Exercise 5-2. List the exact nine fields in the wave preimage. Explain why removing any field or retaining `sig` would create a different protocol.

Exercise 5-3. Given an unordered set of individually valid frames, detect two children with the same `(stream_id, seq, prev)` and different waves. Do not choose a winner. *A selected solution appears in Appendix C.*

Exercise 5-4. Implement a remembered-head store that refuses lower sequences and a different wave at an already observed sequence.

Exercise 5-5. Add calendar-valid UTC checking to a verifier. Test leap years, month length, hour 24, leap second 60, offsets, and fractional precision.

5.8 Chapter Summary

- A frame always has exactly eleven keys; absent and null are not interchangeable.
- The producer computes particle first, then wave, then optionally signs.
- The consumer validates shape and binding before recomputing addresses and chain links.
- Biography streams chain particles; swarm streams also chain waves and require signatures.
- Forks are surfaced and owner-resolved, never auto-merged.
- Re-genesis changes the registered live beginning without changing retired bytes.

The frame is a small object with a large discipline. Next: the two forms that carry it.

Chapter 6 — The Wire: `POST /chat`

> ***In this chapter:*** *use the one synchronous endpoint, understand the asynchronous frame form, make retries idempotent, and keep local, cloud, and managed tiers on the same request shape.*

Frames are the record; the wire is how they move and how agents actually talk. RAPP keeps the wire deliberately small: one synchronous endpoint and one asynchronous artifact form. "Engine, not experience" means capabilities grow behind the door or as registered frame kinds, never by inventing another envelope.

6.1 One Endpoint

```
POST /chat
Content-Type: application/json

{ "user_input": "<plain-language request>",
  "session_id": "<optional>",
  "idempotency_key": "<optional>",
  "conversation_history": [ {"role": "...", "content": "..."} ] }
```

The response:

```
200 OK
{ "response": "<assistant text>", "agent_logs": ["<what fired>"],
"session_id": "<id>" }
```

That is the whole contract for talking to a RAPP brainstem. There is exactly one required input key — `user_input`. Unknown request members are ignored for forward compatibility. A successful response has exactly the three shown members; producers do not grow private success envelopes.

The brainstem loads its `soul.md` as the system prompt, discovers its agents, and decides via tool-calling which of them run. New capability is a new agent file dropped into `agents/`, not a new route: the synchronous wire does not grow.

6.2 Errors Are Typed, Not Prose

A protocol you can rely on fails in named ways. RAPP §8 specifies HTTP 422 with one exact shape for malformed input, an unknown session, or a protocol refusal:

```
{
  "error": {
    "code": "unknown-session",
    "step": null
  }
}
```

`code` is registered in the estate registry. `step` is null or one of the frame verification steps `"1"`, `"1a"`, `"2"`, `"3"`, `"4"`, `"5"`, `"6"`. Authentication challenges exposed by a particular host are deployment behavior; they must not create a second successful chat shape.

An error that is only a human sentence is a dead end for the program on the other side; a typed error is a branch it can take.

6.3 Idempotency

Agents retry. Networks drop responses after the work was done. A request may carry an `idempotency_key`, and replaying the same key returns the original response rather than appending a second turn. With a `session_id`, the key is scoped to that session; without one, the key also deduplicates session creation.

Content addressing makes duplicate work visible, but the idempotency key preserves the full request result — including the session identifier and response — when a caller cannot know whether the first response was lost before or after execution.

6.4 The Asynchronous Form

The second wire form is an append-only §7 frame published to a stream: a repository path, an event log, or another transport that preserves the exact frame value. The transport is not allowed to reparent, rename, or "upgrade" the frame in flight. Consumers verify it against the stream of record and current trusted head.

Within that form there are two stream disciplines.

6.5 Two Kinds of Stream

The `stream_id` of chapter 5 tells you which wire discipline applies:

- **Biography streams** are addressed by a **rappid** (`rappid:@owner/slug:64hex`). They are one agent's worldline. `prev_wave` is null; integrity is the particle chain; a signature is optional for a keyless organism.
- **Swarm streams** are addressed by a `net:` **id** (e.g. `net:commons`) — a shared space many actors append to, like the Commons where brainstems introduce themselves. Here `prev_wave` chains the *waves* (whole frames), and every frame **MUST** be signed (chapter 5, step 6), because in a shared stream you cannot trust the envelope of a frame you did not write. The reference `verify_frame` enforces exactly this split: it demands `prev_wave` on `net:` streams past genesis and refuses an unsigned swarm frame (vector V9).

The same frame object serves both; only the discipline around it differs, and the `stream_id` prefix declares which discipline is in force.

6.6 Tiers Are the Same Shape

The brainstem runs locally (a Flask server on `localhost:7071`), on a cloud endpoint, or behind a managed studio — and all three speak the identical `POST /chat` with the identical `user_input` shape. Moving from your laptop to the cloud is a change of `RAPP_BRAINSTEM_URL`, not a change of protocol. This is the deepest reason the wire is kept to one endpoint: the moment there are two doors, the tiers drift apart. One door, every tier, is what lets the same client drive all of them.

6.7 Checkpoint: One Request, One Shape

Against a running brainstem:

```
curl -sS http://localhost:7071/chat \
  -H 'content-type: application/json' \
  -d '{
    "user_input": "Describe your loaded capabilities in one
sentence.",
    "idempotency_key": "book-ch6-001"
  }'
```

Repeat the exact request. A conformant implementation returns the original result instead of creating a duplicate turn. Then send `{ "messages": [] }`: the server should return a typed 422, not guess that `messages` meant `user_input`.

6.8 Exercises

Exercise 6-1. Send missing, empty, and wrong-typed `user_input` values to a test server. Record the exact 422 envelope and require a registered error code.

Exercise 6-2. Implement an idempotency result store that deduplicates both an existing-session turn and session creation. Store the complete original response. *A selected solution appears in Appendix C.*

Exercise 6-3. Write a client that rejects a 200 response with missing or extra members rather than accepting a private server dialect.

Exercise 6-4. Build an asynchronous reader that receives a frame and a path-of-record stream identifier separately. Prove a valid genesis replayed under another path fails at 1a.

6.9 Chapter Summary

- RAPP has one synchronous endpoint, `POST /chat`, and one asynchronous form, the §7 frame.
- Only `user_input` is required; successful responses have exactly three members.
- Protocol failures use registered error codes and optional verification-step identifiers.
- Idempotency covers both ordinary turns and session creation.
- Biography and swarm streams use the same frame with different wave/signature rules.
- Deployment tiers change the URL, not the protocol shape.

The wire carries frames between living agents. To hand over a whole organism — identity, soul, code, and state — we need a larger content-addressed unit.

Chapter 7 — The Egg

> ***In this chapter:*** *package a whole unit of the RAPP world, distinguish JSON and tree containers, derive the egg's address from its manifest, reject unsafe archive paths, and build a deterministic organism egg with the reference profile.*

A frame is a moment; a chain is a life. To hand an agent to someone else — identity, soul, code, and state together — we need a content-addressed package larger than one event. RAPP calls that package an **egg**.

The egg reuses every earlier lesson: canonical values, named hash spaces, minted identity, exact schemas, optional signatures, and refusal rather than repair.

7.1 One Spec, Six Variants

The egg's history is the clearest case of the drift this protocol ends. The format was once re-specified by several documents, each naming a slightly different archive and each believing it was authoritative. RAPP §9 replaces that family of dialects with one manifest and six registered variants:

VARIANT	CONTAINER	PACKAGES	MINIMUM VIABILITY
`organism`	stored ZIP	one living agent	`rappid.json`, `soul.md`
`rapplication`	stored ZIP	one runnable application	`rappid.json`, exactly one root `agent.py`
`session`	canonical JSON	runtime and transcript	payload `{runtime, transcript}`
`invite`	canonical JSON	signed pointer to a space	target fields + estate-owner signature
`neighborhood`	stored ZIP	a set of member organism eggs	members list and matching sub-eggs
`estate`	stored ZIP	a set of neighborhood eggs	neighborhoods list and matching sub-eggs

The `variant` member changes the viability rules, not the manifest shape. Adding a package kind is a registry and standard change, not permission to mint `my-new-egg/1.0`.

7.2 The Seven-Member Manifest

Every egg has exactly this manifest:

```
{
  "schema": "rapp/1-egg",
  "variant": "organism",
  "rappid": "rappid:@owner/agent:<64hex>",
  "created_utc": "2026-08-20T12:00:00.000Z",
  "contents": [
    {"path": "rappid.json", "hash": "<64hex>"},
    {"path": "soul.md", "hash": "<64hex>"}
  ],
  "payload": {},
  "sig": null
}
```

`contents` lists every packed file except `manifest.json`, exactly once. Paths are relative POSIX paths with no empty, `.` or `..` segment, no leading slash, and no backslash. They are sorted by their UTF-8 bytes.

JSON variants have no packed files, so `contents` is exactly `[]`. Tree variants carry one `manifest.json` plus the listed files.

7.3 Two Address Roles

The current standard does **not** name the raw archive with `rapp/1:egg`. It uses the two egg spaces for different levels:

```
one packed file:
    Hb("rapp/1:egg", file_octets)

the egg as a whole:
    H("rapp/1:egg-manifest", manifest without sig)
```

Each `contents[].hash` protects one file's exact octets. The manifest binds those paths and hashes to the variant, rappid, timestamp, and payload. The manifest address is therefore the identity of the whole egg.

`sig` is excluded from the egg address. Signing or re-signing authenticates the same addressed package instead of creating a new package identity. The serialized container bytes may change when a signature is attached; the egg address does not.

This is parallel to a frame, but not identical:

FRAME	EGG
particle addresses payload value	`rapp/1:egg` addresses each file's raw octets
wave addresses unsigned envelope	manifest address names the unsigned package claim
`sig` authenticates the frame	`sig` authenticates the manifest

7.4 Deterministic Containers

JSON variants

`session` and `invite` are serialized as the exact UTF-8 bytes of `canonical(manifest)`. There is no wrapper object and no pretty-printing.

Tree variants

The other four variants use ZIP with a deliberately narrow profile:

- compression method `stored` for every entry;
- `manifest.json` first;
- remaining entries in `contents` order;
- manifest bytes exactly `canonical(manifest)`;
- timestamps fixed to `1980-01-01 00:00:00`;
- UTF-8 filename flag set; and
- no extra fields.

Compression is transport policy outside the egg. Deflate is not used inside because different library versions can encode the same files differently. Two conformant packers given the same manifest value and file octets emit byte-identical containers.

Deterministic packing is useful even though the public egg identity comes from the manifest: it makes cache comparison, fixture generation, and cross-language conformance much easier to prove.

7.5 Verify Integrity, Then Viability

An egg consumer has two jobs, in order.

Integrity

1. Parse the JSON or ZIP without extracting it.
2. Require exactly the seven manifest members and a registered variant.
3. Enforce path grammar, uniqueness, sort order, and exact archive entry set.
4. Recompute every `contents[].hash` from the stored file octets.

5. Verify a present signature; require it for `invite`.

Viability

Only after the bytes are safe does the consumer check whether the package can serve its declared purpose: required organism files, exact session payload fields, one root application agent, member sub-eggs, and so on.

The order matters. Code that first extracts `../../outside` and only later notices the path was invalid has already lost. A safe consumer treats the archive as untrusted data until every path and entry-set rule passes.

7.6 Invites and Owner Succession

An `invite` is a QR-sized pointer, not a ZIP of neighborhood members. Its payload has exactly:

```
{
  "target_rappid": "<rappid>",
  "target_url": "<string>",
  "target_kind": "neighborhood"
}
```

`target_kind` is `neighborhood` or `estate`. The signature is required and must verify under the estate-owner succession in effect at `created_utc`. A signature by a newly minted but otherwise valid rappid is insufficient: anyone can mint an identity, but only the space authority can admit members.

Chapter 8 explains signature verification; chapter 9 explains time-scoped owner succession.

7.7 Checkpoint: Pack and Verify an Organism

From the repository root:

```
python3 - <<'PY'
import rapp as R

rid = R.mint_rappid("reader", "first-organism")
files = {
    "rappid.json": ('{"rappid":"' + rid + '"}').encode(),
    "soul.md": b"# A small, portable organism\n",
}

blob = R.pack_egg(
    variant="organism",
    rappid=rid,
    created_utc="2026-08-20T12:00:00.000Z",
    files=files,
)
ok, step, why = R.verify_egg(blob)
manifest, unpacked = R.read_egg(blob)

print("bytes:", len(blob))
print("address:", R.egg_address(manifest))
print("files:", sorted(unpacked))
print("verify:", ok, step, why)
PY
```

Run it twice with a fixed `rid`: the bytes and address match. Change one byte of `soul.md`: its file address changes, then the manifest and egg address change. Change only `sig`: the serialized container changes, while `egg_address(manifest)` remains stable.

Then try adding a file named `../escape`. `verify_egg` must refuse the whole package before any file is written.

7.8 The Egg Is the Same Discipline at Package Scale

A frame says, “these canonical bytes are this event in this history.” An egg says, “these addressed files and this canonical manifest are this portable unit.” Both close their schemas, type their hash spaces, exclude signatures from stable identity, and require consumers to recompute rather than trust stored digests.

7.9 Exercises

Exercise 7-1. Run `examples/06_pack_an_egg.py`. Add a third file and predict which file hash, manifest field, egg address, and container bytes will change.

Exercise 7-2. Implement the complete relative POSIX path predicate and exact archive-entry-set check without extracting. *A selected solution appears in Appendix C.*

Exercise 7-3. Change one stored file byte inside an otherwise unchanged ZIP. Show that `verify_egg` refuses the file hash before variant viability is considered.

Exercise 7-4. Build a canonical JSON `session` egg by hand. Compare its bytes with `pack_egg` and explain why pretty printing is non-conformant.

Exercise 7-5. Pack the same manifest and files with another language or ZIP library. Continue until the complete byte strings match, not only the extracted files.

7.10 Chapter Summary

- RAPP defines one seven-member egg manifest and six registered variants.
- `session` and `invite` are canonical JSON; the four tree variants are deterministic stored ZIP.
- `rapp/1:egg` addresses individual file octets.
- `rapp/1:egg-manifest` addresses the whole manifest with `sig` removed.
- Consumers verify path safety and integrity before variant viability.
- Invite authority comes from the estate-owner succession, not merely from any valid signer.

The byte model is now complete. The next chapter crosses the boundary from integrity to authorship.

Chapter 8 — Trust and Signatures

> ***In this chapter:*** *where content addressing stops, what a signature adds, the exact RAPP JWS profile, how a verifier discovers keys, and why rotation and revocation are questions about time rather than hash links.*

A hash tells you whether bytes changed. A chain tells you whether a history was rewritten after a trusted head. Neither tells you who wrote the bytes. That last claim — authorship — begins only when a key signs a frame and a verifier can connect that key to an authoritative identity.

RAPP keeps these claims separate on purpose. Systems become unsafe when "the hash checks" quietly turns into "therefore Alice wrote it," or when "this key signed it" quietly turns into "therefore this key was authorized here." Integrity, authorship, authority, and freshness are four different questions.

8.1 Four Claims, Four Checks

CLAIM	EVIDENCE	WHAT IT DOES NOT PROVE
these are the original payload bytes	recomputed `payload_hash`	who produced them
this is the original frame envelope	recomputed `frame_hash`	that it is the current head
this key signed this frame	valid §10 JWS	that the key was authorized at that time
this key was authoritative then	verified, fresh registry + succession records	that a newer head does not exist

The first two checks are local and time-independent. Given a frame, you can recompute them without contacting anyone. The last two cross a trust boundary: you need a public key, an authenticated registry, and a policy about whether that registry is fresh enough.

This is why signatures do not replace the particle or the wave. The hash chain gives every frame a stable identity and makes history tamper-evident. A signature adds a statement by a key about that already-stable frame.

8.2 When a Signature Is Required

RAPP has two signing disciplines:

- A **memory or body stream** may be unsigned. Its particle chain still protects integrity given a trusted head, but an unsigned frame makes no authorship claim.
- A **swarm stream** (`net:*`) must be signed. Several actors may append to one shared wire, so every envelope needs an attributable producer. `verify_frame` refuses an unsigned swarm frame at step 6.

Optional does not mean decorative. If `sig` is non-null, it must verify completely. A consumer must never treat an invalid signature as if the frame had simply been unsigned.

8.3 The Exact JWS Profile

RAPP uses detached, unencoded JWS: RFC 7515 Appendix F plus RFC 7797. The protected header has **exactly** four members:

```
{
  "alg": "EdDSA",
  "b64": false,
  "crit": ["b64"],
  "kid": "rappid:@owner/agent:<64hex>"
}
```

`alg` is `EdDSA` (Ed25519) or `ES256`. `kid` is the signer's keyed rappid. There are no optional header extensions, because an open-ended protected header would create dialects that verifiers interpret differently.

The header is canonicalized before base64url encoding. The frame signing input is:

```
BASE64URL(canonical(protected-header))
    + "."
    + canonical(frame without sig)
```

The stored `sig` is the detached compact serialization:

```
BASE64URL(canonical(protected-header)) .. BASE64URL(signature-
bytes)
```

The empty middle segment is not a missing value. It is the visual marker that the payload travels outside the compact JWS string. In RAPP, that external payload is the canonical frame with only `sig` removed; `frame_hash` remains present and signed.

The complete dependency is:

```
payload
   │ canonicalize + H("rapp/1:particle", ...)
   ▼
payload_hash
   │ included in frame preimage
   ▼
frame_hash = H("rapp/1:wave", frame − {frame_hash, sig})
   │ included in signed frame
   ▼
JWS(frame − {sig})
```

Changing the payload breaks the particle, the wave, and the signature. Changing only envelope metadata breaks the wave and the signature. Re-signing does not change the wave, because `sig` was never in its preimage.

8.4 Key Discovery Is Part of Verification

A JWS can be mathematically valid under any public key an attacker supplies. The verifier must therefore answer a harder question: *is this the public key bound to the* `kid`*?*

RAPP resolves the signer's SPKI DER through the authenticated §13 registry, then checks:

```
Hb("rapp/1:rappid", SPKI_DER) == tail(kid)
```

Only after that binding succeeds does it verify the JWS. The `rappid.json` published at an organism's door is useful source material for producing the registry entry, but it is not itself the verification authority. Otherwise an attacker could publish a new file containing both the identity and the key they wanted you to trust.

A complete signature check therefore has four stages:

1. parse the detached JWS and require the exact protected-header shape;
2. obtain the signer's SPKI from a verified, sufficiently fresh registry;
3. recompute the keyed rappid tail and require it to equal `kid`;
4. verify the signature over the exact canonical signing input.

Any missing registry entry, tail mismatch, unsupported algorithm, extra header member, stale revocation state, or bad signature is a refusal.

8.5 Rotation, Compromise, and Time

Keys expire socially even though old hashes remain mathematically valid. RAPP records that change without rewriting history.

Rotation

A routine rotation creates a new keyed rappid and a registry re-anchor record. The record carries continuity proof from the old key. Frames before the re-anchor time may still verify under the old key; frames at or after that time must not.

Compromise

When the old key cannot safely authorize its successor, the estate owner records a tombstone and a `case:"compromise"` re-anchor. The tombstone has a `revoked_utc`. A verifier refuses old-key signatures on frames whose `utc` is at or after that instant.

Why the comparison uses time

Revocation should not erase valid history. If a key was legitimate on Monday and compromised on Thursday, Monday's signed frames remain evidence. Thursday-and-later frames do not. That makes signature verification the one §7.5 step whose answer may change as new registry facts arrive.

There is a limit worth naming: frame `utc` is producer-controlled. A stolen key can backdate a frame to just before `revoked_utc`. After a compromise, the operator should advance or re-genesis affected stream heads past the revocation boundary instead of trusting the timestamp alone.

8.6 What the Small Reference Profile Does

`rapp.py` intentionally keeps cryptographic dependencies out of the stdlib-only teaching core. It computes particles and waves, enforces chain rules, and refuses an unsigned swarm frame. It does **not** perform Ed25519/ES256 verification, registry key discovery, rotation, or tombstone checks.

That boundary is a teaching aid, not a reduction of the protocol:

SURFACE	SIGNATURE COVERAGE
`rapp.py`	presence rule for swarm frames
`conformance.py`	vector V9 proves unsigned swarm refusal
a full RAPP consumer	all §10 JWS, discovery, succession, and revocation checks

An implementation claiming full **Consumer** conformance must cross the whole boundary. Passing the small reference vectors proves the byte primitives; it does not waive the trust checks.

8.7 Checkpoint: Review a Signature Without Trusting It

Take any signed-frame design and ask these questions in order:

1. Which exact bytes are signed?
2. Can two serializations of the same value produce different signing inputs?
3. Where does the verifier obtain the public key?
4. What binds that key to the claimed identity?
5. What says the key was authoritative at the frame's `utc`?
6. What prevents a stale registry from silently un-revoking it?

If any answer is "the sender tells us," the design has not completed verification. It has only checked a signature.

8.8 Exercises

Exercise 8-1. Canonicalize the four-member protected header and base64url-encode it without padding. Change the source member order and prove the encoded result remains the same.

Exercise 8-2. Add an extra protected-header member and make the verifier refuse it even when the underlying cryptographic signature is valid.

Exercise 8-3. Construct the exact detached signing input for one frame and identify every byte that is and is not base64url-encoded. *A selected solution appears in Appendix C.*

Exercise 8-4. Create frames immediately before, at, and after a rotation time. Build the expected old-key/new-key acceptance table.

Exercise 8-5. Wrap a maintained Ed25519 library behind a RAPP-specific adapter. Keep header, canonicalization, and key-discovery policy outside the primitive signature call.

8.9 Chapter Summary

- Hashes prove byte integrity; signatures add authorship; the registry adds authority.
- Memory and body streams may be unsigned; swarm streams must be signed.
- RAPP uses one exact detached, unencoded JWS profile with EdDSA or ES256.
- A keyed rappid is verified by recomputing its tail from registry-provided SPKI DER.
- Rotation and tombstones are time-scoped so history survives while new misuse is refused.
- The stdlib reference demonstrates the signing boundary but does not implement the full PKI.

The remaining question is where the trusted key records, current genesis, kind registry, and owner succession live. They all meet in one place.

Chapter 9 — The Registry, Evolution, and Security

> ***In this chapter:*** *the estate's signed root of trust, append-only authority, owner succession, the living-standard rule, re-genesis, and the threats that remain after every hash verifies.*

Content addressing removes many trust decisions. If you know an address, an untrusted mirror can serve the object without being able to alter it invisibly. But a hash cannot tell you which address is current, which key may sign, which event kinds exist, or who is allowed to reset a stream's genesis. Those are authority questions.

RAPP concentrates them in one authenticated, append-only registry rather than scattering them through code, repositories, and human convention.

9.1 The Registry Is the Root of Trust

The estate registry, `rapp-map/ecosystem-spec.json`, plays several roles at once:

- it binds keyed rappids to public keys;
- it registers frame kinds and their stream families;
- it records the current genesis of every stream;
- it carries key tombstones and identity re-anchors;
- it names the estate owner and that owner's succession; and
- it points at the canonical protocol and master plan.

Those powers make the registry security-critical. An unsigned mutable registry would let an attacker replace a signing key, hide a tombstone, authorize a forged genesis, or redefine a kind. The rest of the protocol could verify perfectly against an attacker-chosen root.

The registry must therefore authenticate itself.

9.2 The Bootstrap Axiom

Every trust graph starts somewhere. RAPP's one out-of-band bootstrap value is the `estate_owner` keyed rappid string.

Because its tail is:

```
Hb("rapp/1:rappid", estate-owner-SPKI-DER)
```

the rappid acts as a public-key fingerprint. It can be distributed once through a QR code, invitation, documentation, or another trusted channel. The SPKI may travel with the registry; the verifier accepts it only if hashing that SPKI reproduces the already-trusted rappid tail.

This is not "trust on first use" every time the registry is fetched. It is one explicit bootstrap decision, followed by computation.

9.3 Signed, Monotonic, and Fresh

The registry document carries:

- `schema:"rapp/1-registry"`;
- a `registry_seq` uint53;
- append-only entries; and
- a detached §10 signature by the estate owner.

A consumer persists the highest `registry_seq` it has verified and refuses lower values. That single monotonic counter defeats rollback to a registry snapshot from before a key revocation or re-genesis.

Rollback resistance is not freshness. A hostile mirror can keep serving the highest sequence it has seen while withholding a newer one. Consumers should refresh before trust-sensitive decisions and must report an over-age registry as **stale**, not

clean. “The signature verifies” answers authenticity; “this is recent enough for my policy” answers freshness.

The safe retrieval path is:

```
out-of-band estate_owner rappid
        │
        ▼
SPKI tail check
        │
        ▼
registry signature
        │
        ▼
registry_seq monotonicity + freshness policy
        │
        ▼
keys, kinds, genesis records, succession, tombstones
```

9.4 The Append-Only Entry Types

Entries are added, never silently rewritten or removed. Retirement is explicit with `deprecated:true`.

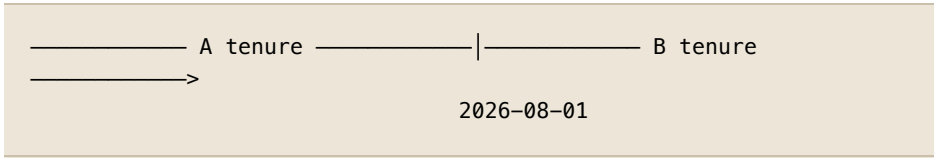
ENTRY	WHAT IT AUTHORIZES OR RECORDS
`protocol`	canonical spec repository, path, and hash
`kind`	one exact event kind and its stream family
`egg-variant`	one allowed §9 package variant
`error-code`	one machine-readable `/chat` failure code
`genesis`	the authoritative starting frame for a stream
`spki`	the public key bound to a keyed rappid
`tombstone`	owner-signed key revocation time
`re-anchor`	identity continuity across upgrade, rotation, compromise, or tag migration
`estate_owner`	the owner currently in force
`master-plan`	the governing estate plan

Closed registries are a protocol feature. If producers can invent kind names, egg variants, or error codes without registration, consumers must guess what those names mean. Registration lets the envelope remain stable while the vocabulary grows deliberately.

9.5 Owner Succession Is Historical

"Owner-signed" means the owner whose tenure covered the artifact's `utc`, not always the owner who holds the key today.

Suppose owner key A served until a rotation at `2026-08-01T00:00:00.000Z`, then key B took over:



A re-genesis frame signed by A in July remains valid after the rotation. A frame signed by A in August does not. Checking every historical artifact only against B would destroy valid history; checking every artifact against either key without tenure boundaries would preserve retired authority forever.

The succession records make authority time-scoped and auditable.

9.6 One Living Standard, Not Permanent Dialects

RAPP is revised in place as a living standard. That does **not** mean the same wire token can quietly acquire a second shape.

- A new registered `kind` can use the existing eleven-field frame.
- A changed key set, field grammar, or hash rule requires a new token and a total migration.
- Published content-addressed artifacts remain immutable.
- Retired live forms are migrated out rather than supported forever as parallel dialects.

This rule is strict because protocol ambiguity compounds. Every legacy reader path doubles the number of producer/consumer combinations that can disagree. RAPP chooses convergence: one current form, with old immutable history retained only when it has been sealed through re-genesis.

9.7 Re-genesis: Change Without Rewriting History

When an immutable chain uses an obsolete envelope, editing its frames would destroy the property the chain was built to provide. Re-genesis changes which history is live without changing the old bytes.

The operation has four essential phases:

1. **Seal the old head.** Hash its exact retained octets in the `rapp/1:seal` space.
2. **Emit a new genesis.** Use the registered family-specific `*.re-genesis` kind, `seq:0`, `prev:null`, an owner signature, and a payload naming the old stream and terminal seal.
3. **Register the new genesis.** Append the new mapping and deprecate every prior genesis for that stream. This append is the linearization point.

4. **Retire the old files.** Preserve them bit-exact under `legacy/`, never serve or extend them as the live chain.

Only the registry-published genesis permits a consumer to reset its remembered head to sequence zero. A lower-sequence head from any other source remains a rollback attack.

Re-genesis is intentionally not an automatic "repair." It is an owner-authorized statement about continuity. A checker may identify the need and prepare evidence; it must not invent authority.

9.8 What Correct Hashes Still Cannot Solve

The security model becomes clearer when its remaining limits are explicit:

THREAT	HASHES CATCH IT?	REQUIRED CONTROL
mirror changes an object	yes	recompute same-space address
mirror serves an old but valid head	no	persisted head + current genesis registry
attacker replays stream A into stream B	not by hash alone	§7.5 step 1a binding
attacker swaps a signing key	no	registry SPKI lookup + rappid tail check
mirror hides a key tombstone	no	registry sequence and freshness
producer future-dates a frame	no	receipt-time skew policy; re-genesis if bricked
archive uses `../` paths	no	§9 path grammar and entry-set verification

The principle is consistent: use content addressing for immutable bytes, and use explicit, authenticated state for facts that can change.

9.9 Operator Checkpoint

Before calling an estate clean, collect evidence for each boundary:

```
[ ] current registry signature verifies from the pinned
estate_owner rappid
[ ] registry_seq is not below the last accepted sequence
[ ] registry age satisfies the local freshness policy
[ ] every stream descends from its sole non-deprecated genesis
[ ] every signed frame resolves a matching SPKI and valid tenure
[ ] tombstones and re-anchors are applied at the frame's utc
[ ] no live frame points into sealed legacy history
[ ] every untrusted egg passed path, entry-set, hash, and variant
checks
```

This is the operational meaning of "fail closed." Unknown does not become pass, and stale does not become clean.

9.10 Exercises

Exercise 9-1. Draw the complete bootstrap path from an out-of-band estate-owner rappid to an accepted frame signature. Mark every hash, signature, and freshness decision.

Exercise 9-2. Implement persisted `registry_seq` rollback protection. Treat equal sequence with different bytes as equivocation. *A selected solution appears in Appendix C.*

Exercise 9-3. Write a re-genesis plan for a three-frame legacy stream, including exact sealed octets, new payload, registry append, and retired path.

Exercise 9-4. Define `fresh`, `stale`, `rollback`, and `unavailable` as four distinct API states. Do not collapse any of them to a boolean.

Exercise 9-5. Threat-model a mirror that serves authentic objects and an authentic but old registry. List what verifies and what remains unsafe.

9.11 Chapter Summary

- The registry is the one signed root for keys, kinds, genesis, succession, and revocation.
- Its bootstrap anchor is the out-of-band estate-owner keyed rappid.
- Signatures prove authenticity; `registry_seq` prevents rollback; freshness policy detects withheld updates.
- Owner authority is time-scoped so rotation preserves valid history without preserving old power.
- The living-standard rule converges current forms instead of accumulating dialects.
- Re-genesis changes the authoritative beginning of a live chain while preserving old bytes as sealed history.
- Correct hashes are necessary, but trusted heads and fresh authority state remain necessary too.

We now have every layer of the model. The final chapter runs them as code against both controlled vectors and the committed artifacts of a real estate.

Chapter 10 — Conformance, and Meeting a Real World

> ***In this chapter:*** *how controlled vectors turn prose into an interoperability claim, how a synchronized real-world harness separates compatible bytes from protocol drift, and how one estate moved from eight precise failures to 46 verified frames without teaching the verifier to tolerate a legacy dialect.*

A specification you cannot test is a wish. This chapter is where RAPP stops being only a document and becomes a tool: first controlled vectors that hold the implementation still, then a harness that turns the same byte rules loose on mutable public repositories.

The estate changed while this book was being written. Its first captured audit found compatible canonical bytes inside non-conformant envelopes. The current audit finds a converged estate. That before-and-after is stronger evidence than either snapshot alone: the checker identified the gap, the owners migrated the artifacts, and the current canonical rules now accept them.

10.1 The Controlled Conformance Suite

`conformance.py` is the executable form of the rules in chapters 2–5. Run it:

```
python3 conformance.py
```

The controlled section prints:

```
RAPP rev-5 — conformance vectors
  [PASS] V1  canonicalization is key-order independent
  [PASS] V1b array order IS significant
  [PASS] V2  domain tags separate the address space
  [PASS] V3  keyless mint is not sha256(owner/slug)
  [PASS] V3  rappid matches the §6.1 grammar
  [PASS] V3  keyed tail == Hb('rapp/1:rappid', SPKI)
  [PASS] V3  mint-once determinism for keyed identity
  [PASS] V4  genesis frame builds and verifies
  [PASS] V4  genesis has exactly 11 keys
  [PASS] V5  payload tamper caught at step 2
  [PASS] V5  envelope tamper caught at step 3 (wave)
  [PASS] V6  child frame links to genesis
  [PASS] V6  broken prev caught at step 4
  [PASS] V7  cross-stream genesis replay refused at 1a
  [PASS] V8  missing key refused at step 1 (no absent-vs-null)
  [PASS] V9  unsigned swarm frame refused at step 6
  ── 16 controlled checks | 16 PASS | 0 FAIL
```

Each vector maps to a promise made earlier in the book. V1/V1b is canonicalization, V2 is domain separation, V3 is mint-once identity, and V4–V9 exercise the frame build and verification steps. Green here is not a generic "clean build." It is a selected set of protocol claims exercised against fixed inputs.

The fixed inputs are important. A controlled conformance suite must not start failing merely because somebody successfully migrated a public repository.

The live observation is separate

After the vectors, `conformance.py` fetches one public frame and describes what it sees:

```
LIVE OBSERVATION — kody-w/twin/frames/0.json (non-gating)
  [CURRENT] frame uses the rapp/1 envelope
       particle reproduces stored payload_hash: True
       frame verifies as its stream genesis: True
```

That observation is useful evidence, but it is not part of the exit condition. It may change, and it may be unavailable offline. Mutable remote state belongs to the estate audit.

10.2 The Real-World Harness

`realcheck.py` asks the harder question: *what does the current public estate actually contain?* It clones or fast-forwards `twin`, `rapp-body`, `rapp-commons`, `rapp-map`, and `RAR`, then walks every numbered frame and every `rappid.json` it finds.

```
python3 realcheck.py
```

For each frame chain, it asks:

1. does the reference canonicalizer reproduce the stored address?
2. does each `prev` link equal the previous payload address?
3. does the complete envelope pass `verify_frame` against its actual head and stream?

For each identity record, it checks the §6.1 grammar, 64-hex tail, forbidden name-hash derivation, and current schema label.

Refreshing existing clones is part of the test. A report against a stale local cache is a captured snapshot pretending to be live.

10.3 The Baseline: Bytes Already Agreed

The first captured report, before convergence, contained 32 frames: 29 in `rapp-body` and three in `twin`. The old envelopes used untagged payload hashes, but the reference canonicalizer reproduced all of them:

```
── rapp-body  (29 committed frames) ──
   canonicalization reproduces real stored hash : 29/29 frames
   real chain links per RAPP §7.4 (prev=parent): 29/29 frames

── twin  (3 committed frames) ──
   canonicalization reproduces real stored hash : 3/3 frames
   real chain links per RAPP §7.4 (prev=parent): 3/3 frames
```

This was not yet RAPP content-addressing: the hashes lacked the chapter 3 domain tags. But it was powerful migration evidence. Two independently written programs agreed on the canonical payload bytes, and all 32 historical chain links held.

The lower layer was already compatible. The envelope above it was not.

10.4 The Baseline: Eight Exact Drifts

The same baseline frames failed the current envelope check:

```
frames conformant to RAPP §7 envelope as-is : 0/29   (rapp-body)
frames conformant to RAPP §7 envelope as-is : 0/3    (twin)
real envelope keys:
  [kernel_version, kind, parent_sha, payload, seq, sha256, sig,
spec, ts, twin_id]
```

Every frame was rejected at step 1. The reason was structural and specific:

HISTORICAL FIELD	CURRENT FIELD
`twin_id`	`stream_id`
`ts`	`utc`
`sha256`	`payload_hash`
`parent_sha`	`prev`
absent	`frame_hash`
absent	`prev_wave`

The identity scan found two 32-hex provisional tails and four old schema labels:

```
IS THE DRIFT RAPP FIXES (8):
  [envelope-drift/C1]  rapp-body/frames, twin/frames
  [short-tail/C3]      twin, rapp-commons
  [schema-label]       all four rappid records
```

This was the checker doing its job. Renaming those fields while reading would have hidden the fact that the historical hashes were untagged and that the wave did not exist. A compatibility shim could make the object look current without making it current.

10.5 The Current Estate: Convergence Verified

After the owner-authorized identity re-anchors and frame convergence, the synchronized harness now reports:

```
── rapp-body  (43 committed frames) ──
   canonicalization reproduces stored address  : 43/43 frames
   chain links per RAPP §7.4 (prev=parent)     : 43/43 frames
   frames conformant to RAPP §7 envelope as-is : 43/43

── twin  (3 committed frames) ──
   canonicalization reproduces stored address  : 3/3 frames
   chain links per RAPP §7.4 (prev=parent)     : 3/3 frames
   frames conformant to RAPP §7 envelope as-is : 3/3

Inspected frames: 46
Current RAPP envelopes accepted: 46/46
Remaining drift findings: 0
```

All four inspected identity records now use `schema:"rapp/1"` and valid 64-hex tails. Live frames carry `payload_hash`, `frame_hash`, `prev`, `prev_wave`, `stream_id`, and fixed-form `utc`. The legacy aliases are absent.

The result went green because the estate changed, not because `verify_frame` learned to reinterpret legacy bytes as current frames. Repository history and sealed legacy artifacts keep the before-state available as evidence; the live paths contain the current form.

10.6 What Before and After Prove

Read the two snapshots together:

> *Before migration, RAPP reproduced the historical canonical bytes and refused the wrong envelopes. After migration, it reproduces the domain-tagged addresses and accepts all 46 current frames. The red report described the work; the green report proves that work reached the public artifacts.*

That is a complete protocol story:

1. **discover** agreement at a lower layer;
2. **refuse** ambiguity at the higher layer;
3. **classify** each mismatch by a normative rule;
4. **authorize** identity and genesis changes;
5. **migrate** producers and current artifacts;
6. **rerun** the unchanged acceptance rules; and
7. **retain** historical evidence without serving it as a live dialect.

The objective was never to make the report green by any means. It was to make reality satisfy the one form the checker already required.

10.7 Fail Closed

The live harness must distinguish **clean**, **drift**, and **not inspected**.

- If a repository cannot be cloned or fast-forwarded, the run fails.
- If an artifact cannot be parsed, it becomes a finding.
- If a stored address cannot be recomputed, it becomes a finding.
- If a frame cannot be verified in sequence, it becomes a finding.
- If zero findings remain after all surfaces were refreshed and read, the report is clean.

A checker that greens because it could not reach a repository has said “no drift” when it means “I did not look.” A stale cache can tell the reverse lie: “drift remains” after owners already converged. Fail-closed evidence must fail on

blindness in either direction.

10.8 Reading a Red Report

A refusal is most useful when it identifies the boundary that failed:

FAILURE	MEANING	CORRECT RESPONSE
canonical bytes differ	producer and consumer do not share §4	replace the canonicalizer; regenerate unpublished output
same bytes, wrong domain-tagged hash	object was named in the wrong address space	migrate through the current producer; do not alias hashes
rappid grammar or tail fails	identity is provisional or non-canonical	owner-authorized re-anchor
frame step 1/1a fails	wrong shape, type, registry value, or stream binding	reject; re-genesis if committed history is immutable
frame step 2/3 fails	payload or envelope integrity failed	reject as corruption or tampering
frame step 4/5 fails	chain or wire continuity failed	surface fork or drift; never reparent automatically
frame step 6 fails	authorship or authority failed	refresh registry, then reject if still invalid
egg integrity/viability fails	unsafe bytes or incomplete package	reject the whole egg

The repeated instruction is **reject first, migrate deliberately**. A consumer should not mutate the received artifact until it passes.

10.9 From Finding to Convergence

A disciplined migration has a beginning and an end:

1. capture the failing artifacts and exact checker output;
2. classify every mismatch by normative section;
3. identify the current owner and authority record;
4. generate conformant artifacts without modifying published addressed bytes;
5. re-anchor identity or re-genesis history when continuity requires authorization;
6. update the append-only registry;
7. rerun both controlled conformance and the synchronized real-world harness; and
8. delete retired live forms, retaining only sealed history where §12.1 requires it.

The current report demonstrates the last step that many standards efforts omit: go back to the world and prove that the migration landed.

10.10 Exercises

Exercise 10-1. Run both `conformance.py` and `realcheck.py`. Explain why only the first can be a stable offline protocol gate.

Exercise 10-2. Create six synthetic findings: canonical mismatch, wrong address space, shape drift, stream replay, chain break, and stale registry. Classify each at its first normative boundary. *A selected solution appears in Appendix C.*

Exercise 10-3. Capture a small estate fixture for offline regression without calling it "live." Record source repository, commit, path, and capture time.

Exercise 10-4. Make a local estate clone one commit stale, run an intentionally non-refreshing audit, then run `realcheck.py`. Explain the evidence difference.

Exercise 10-5. Write an acceptance statement for a migration that names the red baseline, the owner authorization, the changed producer, the current green result, and the retained history.

10.11 Chapter Summary

- Controlled vectors prove selected implementation claims without depending on mutable remote state.
- A synchronized estate audit proves what current committed artifacts actually do.
- Historical canonical bytes can agree while their legacy envelopes correctly fail.
- The baseline report found eight actionable drifts; the current report verifies 46/46 frames and zero remaining drift.
- Fail-closed checks distinguish clean, drift, unavailable, and stale.
- Every refusal should map to an authorized migration, never an invisible repair.

That is RAPP end to end: a language of five primitives, one wire, explicit trust, portable packages, and executable evidence that a real estate can move from incompatible history to one current form.

Chapter 11 — Implementing the Language

> ***In this chapter:*** *choose a conformance target, build the language from canonical values upward, keep address spaces in the type system, separate mint from load, make verification pure, and turn an append into one atomic programming operation.*

The shortest path to a RAPP implementation is not to begin with `/chat`, ZIP, or signatures. It is to follow the dependency graph. Every upper layer assumes the lower layer has exactly one answer:

```
safe I-JSON parser
      ↓
canonical bytes
      ↓
H / Hb and typed addresses
      ↓
rappid mint and load
      ↓
frame build and verification
      ↓
egg pack and verification
      ↓
JWS, registry, succession, and current heads
```

An implementation built in another order tends to hide lower-layer ambiguity behind a convenient API. RAPP asks you to expose the bytes first.

11.1 Choose the Claim You Are Making

The standard defines three conformance classes:

CLASS	MINIMUM RESPONSIBILITY
producer	emit only current canonical values, addresses, identities, frames, and eggs
consumer	run every applicable frame, egg, signature, registry, and freshness check
router/mirror	preserve addressed bytes and provenance without inventing protocol surfaces

One program may implement more than one class, but the claims should remain explicit. A producer that can build a valid frame is not automatically a full consumer. The small `rapp.py` profile demonstrates canonicalization, addressing, identity, frames, and eggs; it does not become a full consumer until §10 cryptography and §13 authority checks are supplied.

Write the target at the top of the port:

```
Target: RAPP rev-5 Producer + byte-core Consumer
Implemented: §§4–7, §9 integrity/viability
External adapters required: §10 JWS, §13 registry and freshness
```

That note prevents a passing byte fixture from being advertised as complete trust verification.

11.2 Start With Admissible Values

Canonicalization cannot repair a permissive parser after the fact. The parse boundary must reject:

- duplicate object member names;
- unpaired UTF-16 surrogates;
- numbers outside the RAPP binary64 round-trip domain;
- values deeper than 64 levels; and
- canonical output larger than 1 MiB.

The teaching profile then makes one deliberate restriction: no floats. Its recursive shape is small enough to inspect:

```
def canonical(v):
    if v is None or isinstance(v, bool):
        return json.dumps(v)
    if isinstance(v, int):
        return json.dumps(v)
    if isinstance(v, float):
        raise ValueError("full JCS number serialization required")
    if isinstance(v, str):
        return json.dumps(v, ensure_ascii=False)
    if isinstance(v, list):
        return "[" + ",".join(canonical(x) for x in v) + "]"
    if isinstance(v, dict):
        return "{" + ",".join(
            json.dumps(k, ensure_ascii=False) + ":" +
canonical(v[k])
            for k in sorted(v)
        ) + "}"
    raise ValueError("non-I-JSON value")
```

Do not copy that sample into a production port and claim complete RFC 8785 support. A production implementation either imports a tested JCS implementation or supplies the exact UTF-16 key order and ECMAScript binary64 serialization with the edge vectors to prove it.

The first cross-language fixture should contain nested objects, arrays, empty values, non-ASCII text, and key-order permutations. Compare UTF-8 **bytes**, not pretty-printed strings.

11.3 Put the Address Space in the Type

The hashing functions are mechanically simple:

```
def H(space, value):
    preimage = space.encode() + b"\x0a" +
canonical(value).encode("utf-8")
    return hashlib.sha256(preimage).hexdigest()

def Hb(space, octets):
    preimage = space.encode() + b"\x0a" + octets
    return hashlib.sha256(preimage).hexdigest()
```

The design work happens at their boundary. A bare digest should not move through the program without its space:

```
@dataclass(frozen=True)
class Address:
    space: str
    digest: str

class Store:
    def put(self, address: Address, value): ...
    def get(self, address: Address): ...
```

This changes address-space confusion from a convention into an API error. Run `python3 examples/04_typed_addresses.py` to see a cross-space lookup fail even when the caller reuses a valid 64-hex digest.

Use different constructors when the input domain differs:

```
Address.of_value("rapp/1:particle", value)  → H
Address.of_bytes("rapp/1:egg", octets)      → Hb
```

Do not expose a generic helper that silently guesses between them.

11.4 Separate Mint From Load

Identity code needs two visibly different operations:

```
mint_rappid(owner, slug, spki_der=None)  # creates one new anchor
load_rappid(stored_record)               # reuses the stored tail
```

Never implement `get_or_create_rappid(owner, slug)`. That name invites a fallback from failed storage into a new mint, which turns a transient read error into an identity change.

The load path should:

1. parse and canonicalize the existing identifier without inventing a tail;
2. classify a non-64-hex historical tail as provisional;
3. refuse to emit provisional identity into a current frame or egg; and
4. require a verified registry re-anchor before accepting a successor.

The mint path should validate owner and slug **before** consuming entropy or accepting SPKI bytes. The returned identity is then stored durably before any frame refers to it.

11.5 Make Frame Construction Boring

A frame builder should be a pure function of explicit inputs. It does not read the clock, load the head, choose a kind, or publish:

```
frame = build_frame(
    kind=kind,
    stream_id=stream_id,
    seq=head.seq + 1,
    utc=utc,
    payload=payload,
    prev=head.payload_hash,
    prev_wave=head.frame_hash if swarm else None,
    sig=None,
)
```

The caller supplies policy; the builder supplies the exact envelope and addresses. This separation makes fixtures deterministic and makes review possible.

Compute in one direction:

1. validate the input domains;
2. compute `payload_hash`;
3. assemble all fields except `frame_hash`;
4. compute the wave over the frame without `frame_hash` and `sig`;
5. attach `frame_hash`; and
6. sign later if required.

Never mutate a built frame in place to append it. A changed field produces a new frame candidate and a new wave.

11.6 Keep Verification Pure and Ordered

The verifier receives a candidate, an optional head, the stream identifier of record, and the authority state it needs. It returns a result; it does not repair, persist, or reparent.

```
verify(candidate, head, stream_of_record, registry_snapshot)
    → accepted
    → refused(step, reason)
```

The step order is observable protocol behavior. Shape precedes hashing so malformed input cannot drive surprising code paths. Particle precedes wave so the refusal identifies the changed layer. Chain precedes trust so a cryptographically signed orphan still fails as an orphan.

`examples/05_failure_atlas.py` constructs seven independent candidates and asserts refusal at steps 1, 1a, 2, 3, 4, 5, and 6. Keep an equivalent negative suite in every port. Positive round-trips alone do not prove that two consumers reject the same language.

11.7 Treat Append as a Transaction

Building a valid child is not enough. Two writers can both read head 7 and build different frame 8 values. The storage operation must compare the head it read with the head it is replacing:

```
append(stream, payload):
    observed = load_current_head(stream)
    candidate = build_child(observed, payload)
    verify(candidate, observed, stream.id)
    write_object_if_absent(candidate.frame_hash, candidate)
    compare_and_swap_head(
        expected=(observed.seq, observed.frame_hash),
        replacement=(candidate.seq, candidate.frame_hash),
    )
```

If the compare-and-swap fails, do not silently rebuild on the new head under the same idempotency key. Return the existing result for a replay or surface a concurrent append so the caller can make a new semantic decision.

Writing the addressed frame before moving the head is safe: an unreferenced immutable object can be garbage-collected. Moving the head before the object is durable creates a dangling current history.

11.8 Verify Eggs Before Extraction

The egg implementation follows the same split:

```
pack(manifest value, files) → deterministic bytes
read(bytes)                 → manifest + in-memory entry map
verify(manifest, entries)   → integrity, then viability
extract(verified entries)   → application policy
```

The verifier compares the archive entry set to `contents` and validates every path before writing anything. A convenience API such as `zip.extractall()` must never be the parser.

Run `python3 examples/06_pack_an_egg.py`. It proves stable bytes for the same inputs, a changed egg address after one file changes, and refusal of `../escape`.

Independent packers should exchange fixtures for all six variants. “My packer can read its own output” is a unit test, not interoperability.

11.9 Put Cryptography Behind a Narrow Adapter

Do not implement Ed25519, ES256, DER, or JWS primitives from scratch. Wrap a maintained cryptographic library with the exact RAPP profile:

```
sign(canonical_header, canonical_frame_without_sig, private_key) →
detached JWS
verify(detached_jws, canonical_frame_without_sig, SPKI_DER)      →
bool
```

The RAPP adapter owns:

- the exact four-member protected header;
- JCS header bytes;
- detached, unencoded payload construction;
- allowed algorithms;
- `kid` parsing; and
- registry key lookup and rappid-tail binding.

The crypto library owns scalar arithmetic, signature encoding, and key parsing. This boundary lets the protocol tests remain stable when the underlying library changes.

11.10 Build a Port Matrix, Not One Golden File

A useful cross-language matrix has four dimensions:

DIMENSION	EXAMPLES
value domain	nested values, Unicode, boundary integers, refused numbers
address space	particle, wave, rappid, file, manifest, seal
structure	genesis, child, swarm, each egg variant
failure	malformed shape, tamper, replay, fork, unsafe path, stale key

For every positive fixture, record:

```
input value
canonical UTF-8 hex
space tag
complete hash preimage hex
expected digest
expected frame or manifest
```

For every negative fixture, record the refusal step and reason category. An implementation that accepts extra forms is no more conformant than one that rejects valid forms.

11.11 Definition of Done

Before calling a new implementation RAPP:

```
[ ] parser enforces the RAPP I-JSON input domain
[ ] canonical bytes match an independent implementation
[ ] address values always retain their spaces
[ ] mint and load are separate identity operations
[ ] builder emits exactly eleven frame keys
[ ] verifier observes the complete ordered checklist
[ ] append uses an atomic remembered-head comparison
[ ] egg verification precedes extraction
[ ] JWS adapter uses the exact protected header and signing input
[ ] registry sequence, freshness, succession, and tombstones are
enforced
[ ] controlled positive and negative vectors pass
[ ] at least one independently produced artifact round-trips
```

The final item is the one most often skipped. A standard exists so programs written by people who never met can agree. Test with one.

11.12 Exercises

Exercise 11-1. Choose Producer, Consumer, or Router/Mirror for a small program you want to build. Write its conformance declaration and list every normative section it must implement.

Exercise 11-2. Implement the transactional `append` operation above with an in-memory compare-and-swap head store. Simulate two writers that observe the same head and prove only one becomes current. *A selected solution appears in Appendix C.*

Exercise 11-3. Port `examples/05_failure_atlas.py` to another language. Require the same seven step identifiers rather than only "accepted/refused."

Exercise 11-4. Produce one organism egg with two independent ZIP libraries or languages. Compare the complete bytes and explain every difference before changing code.

Exercise 11-5. Add a full-consumer test matrix for a key rotation at time `T`: one old-key frame before `T`, one at `T`, one after `T`, and one new-key frame after `T`.

11.13 Chapter Summary

- Implement RAPP from the value domain upward; every layer depends on exact lower-layer bytes.
- State the conformance class and the boundaries a small profile does not implement.
- Keep the address space in the type system and keep mint separate from load.
- Make builders and verifiers pure; make append transactional.
- Verify eggs before extraction and use maintained crypto behind a narrow RAPP adapter.
- Test rejection behavior and exchange artifacts with an independent implementation.

The appendices condense the rules, vocabulary, and selected exercise solutions for use while building.

Appendix A — RAPP Reference Manual

Terse, normative-mirroring. Chapters 1–11 teach; this reference is what you keep open while building. Section numbers cite `SPEC.md`. Requirements language (MUST / MUST NOT / SHOULD / MAY) is RFC 2119 / RFC 8174.

This appendix summarizes chapters 1–11. If its wording and `SPEC.md` differ, the specification is authoritative.

A.1 Canonicalization (§4)

- Values are **I-JSON** (RFC 7493), serialized by **RFC 8785 (JCS)**.
- Object keys sorted by UTF-16 code unit; no insignificant whitespace; shortest string escaping; non-ASCII emitted as raw UTF-8. Arrays keep order. Duplicate keys → error.
- Numbers: full RFC 8785 binary64 serialization; round-trip test MUST accept `0.1`. Reference profile restricts to exact integers/strings/bool/null/array/object.
- No Unicode NFC normalization for new content. No schema coercion.

A.2 Hashing (§5)

- One hash: **SHA-256** (FIPS 180-4), lowercase, 64 hex.
- `H(space, v) = SHA-256(utf8(space) ‖ 0x0A ‖ canonical(v))`
- `Hb(space, octets) = SHA-256(utf8(space) ‖ 0x0A ‖ octets)`
- Spaces: `rapp/1:particle`, `rapp/1:wave`, `rapp/1:rappid`, `rapp/1:egg`, `rapp/1:egg-manifest`, `rapp/1:seal`.

A.3 Identity — rappid (§6)

- Form: `rappid:@<owner>/<slug>:<64hex>`; `owner` / `slug` are `[a-z0-9]` with internal single hyphens (case-sensitive, RFC 7405). Tail is 64 lowercase hex.
- Mint **once**: keyless `tail = Hb("rapp/1:rappid", uuid4_octets)` (RFC 9562); keyed `tail = Hb("rapp/1:rappid", SPKI_DER)` (RFC 5280).
- MUST NOT: `sha256("<owner>/<slug>")` or any name-hash; MUST NOT recompute from mutable facts.
- Re-anchor (§6.3), verifiable via an owner-signed registry record, only for `upgrade`, `rotation` (with old-key proof), `compromise` (with tombstone), or `tag-migrate`.

A.4 The Frame (§7)

Exactly 11 keys: `spec, kind, stream_id, seq, utc, payload, payload_hash, frame_hash, prev, prev_wave, sig`.

FIELD	RULE
`spec`	MUST be `"rapp/1"`
`kind`	`noun.verb`, lowercase labels
`stream_id`	rappid (biography) or `net:*` (swarm)
`seq`	uint53; genesis `0`, then `head.seq + 1`
`utc`	`YYYY-MM-DDTHH:MM:SS.mmmZ` (24 chars, ms, Z)
`payload`	I-JSON object
`payload_hash`	`H("rapp/1:particle", payload)` — the **particle**
`frame_hash`	`H("rapp/1:wave", frame\{frame_hash,sig})` — the **wave**
`prev`	previous particle, or `null` at genesis
`prev_wave`	previous wave on `net:` past genesis; else `null`
`sig`	detached JWS string (§10) or `null`

Build order: particle first, then wave over the frame minus `{frame_hash, sig}`.

Verify checklist (§7.5) — returns the failing step:

1. shape & types (exactly 11 keys; `spec`; `kind`; `seq` uint53; `utc` fixed 24; `payload` object; hash fields well-formed). **1a.** stream binding: `stream_id == stream_of_record`.
2. particle: `payload_hash == H("rapp/1:particle", payload)`.
3. wave: `frame_hash == H("rapp/1:wave", frame\{frame_hash,sig})`.
4. chain: genesis ⇒ `seq==0 ∧ prev==null`; else `seq==head.seq+1 ∧ prev==head.payload_hash ∧ utc>=head.utc`.
5. wire: swarm past genesis ⇒ `prev_wave==head.frame_hash`; off-swarm ⇒ `prev_wave==null`.
6. signature: swarm frame MUST be signed; JWS verified per §10.

Forks & re-genesis (§7.6): forks resolved by stream authority, losing branch sealed. Re-genesis: terminal `*.re-genesis` frame with `H("rapp/1:seal",…)` over old head → new genesis in current form citing the sealed head → old frames retained under `legacy/` (sealed, never served).

A.5 The Wire (§8)

- `POST /chat` with `{user_input, session_id?, conversation_history?}` → `{response, agent_logs, session_id}`. Only `user_input` is required.
- Errors typed: `422` malformed request, `401` needs token; frame rejection returns the failing verify step.
- Idempotency key on frame-appending ops; replay returns the same result (natural from content addressing).
- Streams: rappid = biography (`prev_wave` null, sig optional); `net:*` = swarm (`prev_wave` chains waves, sig REQUIRED).
- Tiers (local / cloud / studio) share the identical shape; only `RAPP_BRAINSTEM_URL` differs.

A.6 The Egg (§9)

- Exact seven-member manifest: `{schema,variant,rappid,created_utc,contents,payload,sig}`.

- JSON containers: `session`, `invite`; serialized as `canonical(manifest)`, `contents:[]`.
- Deterministic stored-ZIP containers: `organism`, `rapplication`, `neighborhood`, `estate`; `manifest.json` first, then all files in UTF-8 path order, fixed timestamps, no extra fields.
- File address: `Hb("rapp/1:egg", file_octets)`.
- Egg identity: `H("rapp/1:egg-manifest", manifest\{sig})`; signing or re-signing does not change this address.
- `invite` MUST be signed by the space's estate-owner succession.

A.7 Signatures (§10)

- Detached **JWS** (RFC 7515) over the frame/egg integrity hash; unencoded-payload option (RFC 7797). Algorithms: EdDSA (RFC 8037) or ES256 (RFC 7518); deterministic ECDSA per RFC 6979.
- Keyed identity binds to the signing key via §6.2. Key discovery, rotation, and tombstone per §10; a keyed tail minted under a pre-standard (untagged) formula fails discovery and MUST be re-anchored (§6.3c).

A.8 Conformance & Versioning (§11–§13)

- **Conformance classes (§11):** an implementation conforms when it produces and rejects exactly the `conformance.py` vectors (V1–V9) and honors the §7.5 checklist.
- **Versioning (§12):** one **living standard**; `rapp/1` never denotes two shapes. Change the one spec and migrate (no second `rapp/1`). Published content-addressed artifacts are immutable.
- **No legacy (§12 / Fed. Const. Art. III):** converge and delete; a legacy form encountered is a drift finding. Immutable chains converge by **re-genesis** (§12.1), old frames retained as sealed history (Amendment III-a).
- **Registry as root of trust (§13):** `rapp-map/ecosystem-spec.json`, owner-signed, anchored to an out-of-band `estate_owner` rappid fingerprint, `registry_seq`-monotonic. Owner succession is time-scoped (§13.2).

A.9 The Reference Implementation

`rapp.py` (stdlib only) implements A.1–A.4: `canonical`, `H` / `Hb`, `mint_rappid` / `rappid_valid`, `build_frame` / `verify_frame`. `conformance.py` runs V1–V9. `realcheck.py` runs the whole thing against the live estate. Read `rapp.py` — it is ~140 lines and it is the spec made executable.

A.10 Normative References

RFC 2119/8174 (requirements) · RFC 8785 (JCS) · RFC 8259/7493 (JSON/I-JSON) · FIPS 180-4 (SHA-256) · RFC 3986 (URI) · RFC 9562 (UUID) · RFC 5280 (X.509 SPKI) · RFC 7515/7797/8037/7518/6979 (JWS/signatures) · RFC 7405 (case-sensitive ABNF).

Appendix B — Glossary and Failure Atlas

This appendix is the vocabulary of the book in one place. Definitions summarize the teaching text; `SPEC.md` remains normative.

B.1 Core Terms

address space

A named domain in which a hash has meaning. RAPP stores and dereferences by `(space, hash)`, never by a bare 64-hex string.

authorship

The claim that a particular key signed an artifact. Authorship requires a valid §10 signature and does not by itself prove that the key was authorized.

authority

The right of a signer to make a decision at a particular time. RAPP derives authority from the authenticated registry and its owner-succession records.

biography stream

A body stream identified directly by a rappid. It chains particles through `prev`; `prev_wave` is null.

canonical form

The one RFC 8785 JCS byte representation of an admissible I-JSON value.

consumer

An implementation that validates frames, eggs, signatures, registry state, and all refusal conditions before accepting input.

content address

A SHA-256 digest computed in a named RAPP domain over canonical values or raw octets. The address changes if the addressed content changes.

domain separation

Prefixing hash input with an exact space tag and newline so the same bytes in different roles have different addresses.

egg

A `rapp/1-egg` manifest plus, for tree variants, deterministic stored-ZIP contents. An egg packages an organism, rapplication, session, invite, neighborhood, or estate.

estate

A governed collection of RAPP organisms and neighborhoods sharing an authenticated registry and owner succession.

fail closed

Refuse or report unknown when required evidence is malformed, missing, unreachable, stale, or unverifiable. Never convert inability to check into a pass.

frame

The closed eleven-field RAPP event envelope. It carries a payload particle, a whole-frame wave, chain links, and an optional signature.

freshness

Confidence that authenticated mutable state is recent enough for a decision. A valid registry signature proves authenticity, not freshness.

genesis

The sequence-zero frame from which the current live chain descends. Every stream has exactly one non-deprecated genesis entry in the registry.

head

The highest verified frame descending from the registered genesis. Consumers persist heads to detect rollback and silent reorganization.

identity

Continuity across changing content. In RAPP, identity is the minted tail of a rappid, not its human-readable owner/slug.

integrity

Evidence that bytes or chain links have not changed. Integrity does not imply authorship.

I-JSON

The interoperable JSON profile from RFC 7493 on which RAPP canonicalization is defined.

JCS

JSON Canonicalization Scheme, RFC 8785. It fixes member order, escaping, number form, and whitespace so a value has one serialized form.

kind

A registered `noun.verb` event name. The registry, not the prefix alone, binds a kind to its stream family.

legacy form

A historical encoding that is not the one current canonical form. Live legacy is drift; sealed re-genesis history is retained evidence, not a compatibility dialect.

particle

`H("rapp/1:particle", payload)`, stored as `payload_hash`. It addresses what happened and links a worldline through `prev`.

producer

An implementation that emits only current canonical values, addresses, identities, frames, and egg variants.

provisional identity

A legacy rappid that can be recognized while reading but whose tail is not a conformant 64 lowercase hex characters. It must not be emitted; becoming current requires owner-authorized re-anchoring.

rappid

The self-locating RAPP identifier `rappid:@<owner>/<slug>:<64-lowercase-hex>`. Its tail is minted from UUIDv4 octets or SPKI DER, never from the owner/slug name.

re-anchor

An owner-authorized identity transition for exactly one allowed case: provisional upgrade, key rotation, key compromise, or pre-standard key-tag migration.

re-genesis

The owner-authorized convergence operation that seals an old chain, publishes a new sequence-zero frame, updates the registry, and retires the old bytes under `legacy/`.

registry

The signed, append-only estate root of trust for keys, kinds, genesis records, re-anchors, tombstones, owner succession, and canonical-source pointers.

router/mirror

An implementation that transports or serves RAPP artifacts without inventing endpoints or rewriting addressed bytes.

stream

An append-only sequence of frames sharing one `stream_id`.

swarm stream

A shared `net:*` stream. It chains particles through `prev`, waves through `prev_wave`, and requires a signature on every frame.

tombstone

An owner-signed registry record that revokes a keyed rappid from `revoked_utc` forward.

wave

`H("rapp/1:wave", frame − {frame_hash, sig})`, stored as `frame_hash`. It addresses the exact unsigned envelope and protects wire integrity.

worldline

The particle-linked history of an organism or stream.

B.2 Address Spaces

SPACE	FUNCTION	INPUT
`rapp/1:particle`	frame payload address	canonical value via `H`
`rapp/1:wave`	unsigned frame-envelope address	canonical value via `H`
`rapp/1:rappid`	identity tail	UUID or SPKI octets via `Hb`
`rapp/1:egg`	packed file address inside an egg	raw octets via `Hb`
`rapp/1:egg-manifest`	whole egg identity	manifest without `sig` via `H`
`rapp/1:seal`	retired-head seal	exact retained head octets via `Hb`

A bare digest should always make you ask: **in which space?**

B.3 Frame Verification Failure Atlas

STEP	CONSUMER CHECKS	TYPICAL FAILURE
`1`	exact keys, types, grammar, registry values, fixed UTC	dialect, missing/null confusion, malformed field
`1a`	`stream_id` equals the stream of record	cross-stream replay
`2`	recomputed particle equals `payload_hash`	payload changed
`3`	recomputed wave equals `frame_hash`	envelope changed
`4`	genesis or contiguous particle chain and nondecreasing UTC	gap, fork, wrong parent, time reversal
`5`	swarm `prev_wave` rule; null elsewhere	wrong wire discipline
`6`	required/present signature, key binding, tenure, tombstone	missing authorship or unauthorized signer

The step is a refusal location, not a repair instruction. Correct the producer or perform an authorized migration; do not mutate the received frame until it passes.

B.4 Notation

NOTATION	MEANING
`canonical(v)`	UTF-8 JCS serialization of admissible value `v`
`H(space, v)`	SHA-256 of `utf8(space) + 0x0A + canonical(v)`
`Hb(space, b)`	SHA-256 of `utf8(space) + 0x0A + raw octets b`
`x − {a,b}`	object `x` with exactly members `a` and `b` removed
`64hex`	exactly 64 lowercase hexadecimal characters
`uint53`	integer from 0 through $2^{53}-1$
SPKI DER	canonical DER encoding of a SubjectPublicKeyInfo public key

Appendix C — Selected Exercise Solutions

These are selected solutions, not answer keys for every exercise. Try the exercise first. A good RAPP solution is not merely code that prints the expected digest; it makes the addressed bytes, refusal boundary, and authority assumptions visible.

C.1 Exercise 1-2 — The Failure Atlas

Start from a fresh deep copy for every mutation. Otherwise an early payload edit may cause step 2 to hide the step 3 failure you intended to observe.

```
cases = [
    ("missing-key", remove(frame, "prev_wave"), "1"),
    ("replay", frame, "1a", other_stream),
    ("payload", replace_payload(frame), "2"),
    ("envelope", replace_utc(frame), "3"),
    ("genesis", build_seq_one_without_head(), "4"),
    ("wire", build_body_with_prev_wave(), "5"),
    ("signature", build_unsigned_swarm(), "6"),
]
```

The runnable solution is `examples/05_failure_atlas.py`. The important result is not seven error strings. It is proof that verification order is stable and that each layer can be diagnosed independently.

C.2 Exercise 2-2 — Canonical Byte Fixtures

Store the bytes as hex so the fixture cannot be changed by an editor's encoding or newline rules:

```
values = [
    {},
    {"b": 1, "a": [True, None, "café"]},
    {"nested": {"z": 0, "a": ""}},
]
for value in values:
    text = R.canonical(value)
    print(text)
    print(text.encode("utf-8").hex())
```

For the second value, construction order must disappear, array order must remain, and `é` must appear as UTF-8 bytes `c3a9`, not as an ASCII `\u` escape. A cross-language fixture stores both the value and the expected hex.

C.3 Exercise 3-2 — Typed Addresses

Use an immutable pair:

```
@dataclass(frozen=True)
class Address:
    space: str
    digest: str

class Store:
    def __init__(self):
        self.objects = {}

    def put(self, address, value):
        self.objects[address] = value

    def get(self, address):
        return self.objects[address]
```

Do not add `get_by_digest`. That convenience method would erase the property the type was created to preserve. The complete runnable solution is `examples/04_typed_addresses.py`.

C.4 Exercise 4-2 — Name-Hash Audit

For each stored rappid:

1. validate and split the canonical grammar;
2. compute `sha256(f"{owner}/{slug}")`;
3. compare it with the full stored tail; and
4. report, never rewrite.

```
match = R._RAPPID.fullmatch(rid)
owner, slug, tail = match.groups()
forbidden = hashlib.sha256(f"{owner}/{slug}".encode()).hexdigest()
if tail == forbidden:
    findings.append((path, "name-hash-mint"))
```

A non-matching tail is not proof that the mint was lawful; keyed identity still needs SPKI binding, and keyless identity needs durable mint-once storage. This audit detects one known forbidden derivation.

C.5 Exercise 5-3 — Fork Detection

Group accepted candidates by `(stream_id, seq, prev)`. More than one distinct `frame_hash` in a group is a fork:

```
branches = {}
for frame in candidates:
    key = (frame["stream_id"], frame["seq"], frame["prev"])
    branches.setdefault(key, set()).add(frame["frame_hash"])

forks = {key: waves for key, waves in branches.items() if
len(waves) > 1}
```

Do not pick the lexicographically smaller hash as current. Hash ordering is a deterministic merge order across streams, not authority to resolve two branches of one stream. Surface the fork and require owner-authorized convergence.

C.6 Exercise 6-2 — Idempotent Chat Results

The stored value is the complete original result, not only a "seen" bit:

```
key = (session_id, idempotency_key) if session_id else (None,
idempotency_key)
if key in results:
    return results[key]

response = execute_once(request)
results[key] = response
return response
```

Session creation must store the generated `session_id` in that response. If a retry created a new session before noticing the key, the operation was not idempotent.

Production storage needs an atomic insert-if-absent. Two workers racing on an in-memory check-then-set can still execute twice.

C.7 Exercise 7-2 — Safe Egg Paths

Validate names as data before extraction:

```
def valid_path(path):
    if path.startswith("/") or "\\" in path:
        return False
    parts = path.split("/")
    return all(part not in ("", ".", "..") for part in parts)
```

Then require:

```
set(archive entries) == set(contents paths) + {"manifest.json"}
```

Both checks are necessary. Safe-looking manifest paths do not help if the ZIP carries an unlisted `../../escape`, and an exact entry set does not help if both manifest and ZIP agree on an unsafe path.

C.8 Exercise 8-3 — Construct the Signing Input

Given protected header `h` and frame `f`:

```
header_octets = canonical(h).encode("utf-8")
payload_octets = canonical({k: v for k, v in f.items() if k !=
"sig"}).encode()
signing_input = base64url(header_octets) + b"." + payload_octets
```

The detached compact value stores:

```
BASE64URL(header) .. BASE64URL(signature)
```

Do not base64url-encode the payload in the signing input: `b64:false` is the reason the exact canonical frame bytes remain external and visible. Do not remove `frame_hash`; only `sig` is removed for the signature input.

C.9 Exercise 9-2 — Monotonic Registry State

Persist the highest accepted sequence beside the verified registry digest:

```
def accept_registry(candidate, remembered):
    verify_owner_signature(candidate)
    if candidate["registry_seq"] < remembered.seq:
        raise Rollback
    if candidate["registry_seq"] == remembered.seq:
        if digest(candidate) != remembered.digest:
            raise Equivocation
        return remembered
    enforce_freshness(candidate)
    return Remembered(candidate["registry_seq"], digest(candidate))
```

Equal sequence with different bytes is not a newer registry; it is equivocation. A higher sequence with an old timestamp may still violate the local freshness policy.

C.10 Exercise 10-2 — Classify Synthetic Drift

Classify at the first normative boundary that fails:

MUTATION	CLASSIFICATION
reorder object keys before hashing	no drift if JCS output is unchanged
replace `payload_hash` with bare SHA-256	address-space drift
rename `utc` to `ts`	frame shape, step 1
replay a valid genesis under another path	stream binding, step 1a
change payload without changing hashes	particle integrity, step 2
serve an older verified registry	registry rollback/freshness

The first row matters: source-level difference is not protocol drift when canonical bytes are identical. The others change the protocol claim or the authority state.

C.11 Exercise 11-2 — Transactional Append

An in-memory compare-and-swap store can make the race explicit:

```
class Heads:
    def __init__(self, genesis):
        self.current = genesis

    def compare_and_swap(self, expected_hash, replacement):
        if self.current["frame_hash"] != expected_hash:
            return False
        self.current = replacement
        return True
```

Each writer reads the same head and builds a different valid child. Both children may verify against the observed head, and both may be stored by wave address. Only one call can replace the remembered head:

```
observed = heads.current
a = build_child(observed, {"writer": "a"})
b = build_child(observed, {"writer": "b"})

assert heads.compare_and_swap(observed["frame_hash"], a)
assert not heads.compare_and_swap(observed["frame_hash"], b)
```

The losing frame is an unreferenced immutable object, not a silently accepted second history. A real store performs this comparison atomically and couples it to the idempotency result.

COLOPHON

One source, three editions

The repository Markdown is the editable manuscript. Jekyll renders it as the chapter reader and this complete print volume. A browser print engine produces the PDF from this page, preserving one content source across every edition.

Open the interactive edition to copy every example without retyping it.

Typeset with system serif, sans-serif, and monospace faces on a 6 × 9 inch page.