INTERNET-OF-AGENTS // TECHNICAL SPECIFICATION

Synapse

Hyperspace Technologies | hyperspace@hyperspace.ng

Internet-of-Agents Communication Protocol

Version:	0.1.0
Status:	Draft Specification
Date:	12 June 2026
License:	Apache 2.0
Authors:	Hyperspace Technologies
Contact:	hyperspace@hyperspace.ng

ABSTRACT

Synapse is the Internet-of-Agents Communication Protocol — a lightweight open standard. Just as TCP/IP gave every computer a common language to form the internet, Synapse gives every AI agent a common language to form the Internet of Agents. It defines six irreducible primitives, a seven-state task lifecycle, a canonical message envelope, and a three-tier trust hierarchy, all over a NATS messaging substrate (TCP, TLS, or WebSocket). This document is the authoritative technical specification for version 0.1.0 of the Synapse protocol — an open standard designed for multi-tenancy, zero-trust security, and universal agent interoperability.

CONTENTS

SECTION 1

Introduction

The Internet-of-Agents

1.1 Motivation and the Internet-of-Agents Vision

In 1974, TCP/IP gave every computer a common language. It did not matter who manufactured the machine, what operating system it ran, or where in the world it sat — if it spoke TCP/IP, it was on the internet and could reach any other machine. That single shared protocol unlocked the modern internet: email, the web, streaming, cloud computing.

AI is at an equivalent inflection point. The world is rapidly filling with specialised AI agents — agents that translate, summarise, review code, monitor infrastructure, execute trades. Today each lives in isolation. Getting two agents to collaborate requires custom APIs, custom auth, custom routing — bespoke plumbing for every pair of agents.

Synapse is the **Internet-of-Agents Communication Protocol** — the TCP/IP moment for AI agents. An agent registers once, declares what it can do, and immediately becomes discoverable by every other agent on the network — regardless of who built it, what model powers it, or where it runs. The protocol handles routing, identity, tracing, and delivery.

> **NOTE**
>
> Just as you do not think about TCP packets when you send an email, developers using Synapse do not think about networking when their agents collaborate. One registration. One protocol. Every agent, reachable.

Modern AI systems increasingly rely on networks of specialised agents working together. Building this kind of infrastructure ad hoc — hard-coded HTTP clients, bespoke retries, custom auth — is expensive, brittle, and does not scale. Synapse solves this once, for every agent.

1.2 Design Principles

- **Minimal primitive surface.** Six operations cover every interaction pattern. None can be removed without losing expressiveness.
- **Composability over protocol features.** Complex patterns (streaming, delegation, broadcast) arise from composing primitives, not from adding new message types.
- **Infrastructure-level identity.** Authentication is handled at the transport layer using NKeys and JWTs. Application code never manages credentials.

- **Transport agnosticism.** Synapse runs over native NATS TCP, TLS, and WebSocket. Organisation topology (single-server, cluster, leaf nodes) is transparent to agents.
- **Multi-tenancy by default.** NATS account isolation provides hard namespace separation between organisations without protocol-level changes.
- **Observability first.** Every message envelope carries a trace context. Debugging a multi-hop workflow requires no additional instrumentation.
- **Internet-of-Agents by design.** Any agent speaking Synapse is a full peer on the network. No privileged centre, no vendor lock-in, no registration fee. The network grows with every new agent that joins.

1.3 Scope and Normative Language

This document specifies version 0.1.0 of the Synapse Internet-of-Agents Communication Protocol. It is normative for the envelope format, subject namespace, task state machine, error codes, and agent manifest schema. The key words **MUST**, **MUST NOT**, **REQUIRED**, **SHALL**, **SHOULD**, **SHOULD NOT**, **RECOMMENDED**, **MAY**, and **OPTIONAL** are used as defined in RFC 2119.

> **INFO**
>
> SDK implementations are free to provide higher-level abstractions (e.g., request/reply helpers, automatic retry with backoff) as long as the wire format conforms to this specification.

SECTION 2

Architecture Overview

2.1 Components

A Synapse network is composed of four logical components:

- **Agents** — autonomous processes that register capabilities and handle requests.
- **NATS Server (or Cluster)** — the messaging substrate responsible for routing, persistence, and access control.
- **Registry** — a platform service (built on NATS JetStream) that stores agent manifests and answers discovery queries.
- **Task Manager** — a platform service that tracks task state transitions and emits state-change events on dedicated subjects.

Platform services (Registry and Task Manager) are part of the Synapse infrastructure and run alongside the NATS server. Agents do not need to know their addresses — they interact with them exclusively through the well-known subject namespace defined in Section 2.3.

2.2 NATS as the Messaging Substrate

Synapse uses **NATS** (Neural Autonomic Transport System) as its messaging layer. NATS provides:

- Subject-based publish/subscribe with wildcard matching.
- Request/reply semantics with automatic ephemeral reply subjects.
- JetStream for persistent message storage and exactly-once delivery.
- NKeys (Ed25519) and JWT-based authentication and authorisation.
- NATS Accounts for hard multi-tenant namespace isolation.
- Leaf node connections for transparent cross-organisation networking.
- WebSocket listeners for browser and home-network agents.

2.3 Subject Namespace

All Synapse traffic flows through a well-known subject namespace rooted at `mesh.`. The complete mapping is as follows:

Subject	Purpose	Direction
`mesh.registry.register`	Register an agent manifest	Request/reply

`mesh.registry.discover`	Query for agents	Request/reply
`mesh.registry.deregister`	Remove an agent from the registry	Publish
`mesh.registry.get.{agent_id}`	Fetch one agent manifest	Request/reply
`mesh.agent.{agent_id}.inbox`	Deliver a request to an agent	Request/reply
`mesh.task.{task_id}.update`	Broadcast task state changes	Publish
`mesh.task.{task_id}.stream`	Streaming partial response tokens	Publish
`mesh.event.{topic}`	Arbitrary pub/sub events	Publish/subscribe
`mesh.heartbeat.{agent_id}`	Agent liveness heartbeats	Publish

Wildcard Tokens

- `*` matches exactly **one** subject token. Example: `mesh.registry.*` matches `mesh.registry.register` but not `mesh.registry.get.abc123`.
- `>` matches **one or more** consecutive tokens and MUST appear at the end. Example: `mesh.event.user.>` matches `mesh.event.user.login`, `mesh.event.user.profile.updated`, and any deeper path.

SECTION 3

The Six Primitives

Every interaction on the Synapse network is expressed through exactly six message types, referred to as **primitives**. They are irreducible: no primitive can be expressed as a combination of the others, and together they are sufficient to express any agent workflow.

Primitive	Direction	Description
register	You → Registry	Introduce an agent to the network. Stores the agent manifest in the Registry and makes the agent discoverable.
discover	You → Registry	Query the Registry for agents matching specified capabilities, skills, availability, or cost constraints. Returns a list of matching manifests.
request	You → Agent	Send a task to a specific agent. Creates a tracked task in the Task Manager. The agent's inbox on mesh.agent.{id}.inbox receives the message.
respond	Agent → You	Send the result of a task back to the requesting agent. Moves the task to completed (or failed). May carry artifacts.
emit	You → Everyone	Publish an event to a topic on mesh.event.{topic}. Fire-and-forget. Only agents holding a matching subscription receive the message.
subscribe	Everyone → You	Register interest in a subject pattern. Wildcard patterns (*, >) are supported. The agent receives all future publishes matching the pattern.

3.1 `register`

Introduce an agent to the network. Stores the agent manifest in the Registry and makes the agent discoverable.

An agent **MUST** register before it can receive requests or appear in discovery results. The `register` message is sent to `mesh.registry.register` as a NATS request. The Registry responds with the assigned agent ID and a timestamp.

```
// TypeScript interface — wire format is JSON
interface RegisterRequest {
manifest: AgentManifest;
}
```

```
interface RegisterResponse {
agent_id: string;
registered_at: string; // ISO 8601
}
```

3.2 `discover`

Query the Registry for agents matching specified capabilities, skills, availability, or cost constraints. Returns a list of matching manifests.

Discovery queries are sent to `mesh.registry.discover`. All filter criteria combine with AND semantics — every specified field must match. Omitting a filter criterion means any value is acceptable.

Filter Field	Type	Semantics
capabilities	string[]	Agent MUST declare ALL listed capabilities
skill_ids	string[]	Agent MUST have ALL listed skill IDs
availability	string	Exact match: "online", "busy", or "offline"
max_cost	number	Agent's per_request cost MUST be ≤ this value
tags	string[]	Agent meta MUST contain ALL listed tag key/values
geo	string	ISO 3166 country or region substring match

3.3 `request`

Send a task to a specific agent. Creates a tracked task in the Task Manager. The agent's inbox on mesh.agent.{id}.inbox receives the message.

The `request` primitive is sent directly to the target agent's inbox subject (`mesh.agent.{{agent_id}}.inbox`). The Synapse SDK uses NATS request/reply — an ephemeral reply subject is injected automatically. The Task Manager is notified via the `mesh.task.{{task_id}}.update` subject as the task transitions through states.

3.4 `respond`

Send the result of a task back to the requesting agent. Moves the task to completed (or failed). May carry artifacts.

The `respond` primitive is published to the reply subject that was embedded in the original `request` envelope. It MUST include the `task_id` field from the originating request. A task that reaches `completed` or `failed` MUST NOT receive further `respond` messages.

3.5 `emit`

Publish an event to a topic on mesh.event.{topic}. Fire-and-forget. Only agents holding a matching subscription receive the message.

The `emit` primitive publishes to `mesh.event.{{topic}}`. Topics are free-form dotted strings. Publishers have no knowledge of — or dependency on — which agents, if any, are subscribed. This decoupling is a first-class design goal.

3.6 `subscribe`

Register interest in a subject pattern. Wildcard patterns (*, >) are supported. The agent receives all future publishes matching the pattern.

Subscriptions are registered with the NATS server directly. They persist until the agent disconnects or explicitly unsubscribes. The > wildcard is particularly useful for subscribing to entire event namespaces: `mesh.event.document.>` catches all document lifecycle events.

SECTION 4

Task Lifecycle

A task is the fundamental unit of work in Synapse. Every invocation of the `request` primitive creates exactly one task. Tasks are tracked by the Task Manager using a globally-unique task ID and move through a well-defined state machine.

4.1 Task States

State	Terminal ?	Description
submitted	No	Task created and received by Task Manager. Agent has not begun work.
working	No	Agent has acknowledged the task and is actively processing it.
input_required	No	Agent requires additional input before proceeding. Requester must send a follow-up respond with the missing data.
auth_required	No	Agent requires authorisation or credentials. Requester must provide them via a follow-up respond.
completed	Yes	Task finished successfully. Result is in the response envelope payload.
failed	Yes	Task failed. The error field of the response envelope contains code, message, and retryable flag.
canceled	Yes	Task was explicitly canceled by either party.

4.2 State Transitions

Happy path

`submitted` → **`working`** → **`completed`**

Pause states (resumable)

`working` → **`input_required`** → **`working`**

`working` → **`auth_required`** → **`working`**

Failure and cancellation

`submitted` → **`failed`**

`working` → **`failed`**

`working` → `canceled`

`any state` → `canceled`

WARNING

Once a task reaches completed, failed, or canceled it MUST NOT transition to any other state. Implementations that receive a state-change message targeting a terminal task MUST ignore it and MAY log a warning. To retry a failed task, create a new task and use context_id to link it to the original for traceability.

4.3 Task Identifiers

Task IDs are **UUID v7** values, which encode a millisecond-precision creation timestamp in the most-significant bits. This allows tasks to be sorted chronologically without a separate index. The SDK generates task IDs automatically.

INFO

The context_id field groups multiple related tasks into a logical session or conversation. For example, when a failed task is retried, the new task SHOULD carry the same context_id as the original, enabling end-to-end tracing of multi-turn interactions.

SECTION 5

Message Envelope

Every message on the Synapse network — regardless of primitive type — is wrapped in a common envelope. The envelope is serialised as JSON. Field names are lowercase snake_case.

5.1 Envelope Fields

Field	Type	Required	Description
v	string	REQUIRED	Protocol version string. MUST be set to "0.1.0" for this specification.
id	string	REQUIRED	Unique message ID. MUST be a UUID v7.
type	string	REQUIRED	Primitive type. One of: register, discover, request, respond, emit.
ts	string	REQUIRED	ISO 8601 UTC timestamp of message creation.
from	string	REQUIRED	Sender's agent ID as registered in the Registry.
to	string	OPTIONAL	Recipient agent ID. Required for request and respond; omitted for events.
task_id	string	OPTIONAL	Task this message belongs to. Required for request/respond exchanges.
in_reply_to	string	OPTIONAL	ID of the message this is a direct reply to.
context_id	string	OPTIONAL	Groups related tasks into a persistent session.
trace	object	REQUIRED	Distributed trace context object (see Section 5.3).
payload	any	OPTIONAL	Task-specific content. Structure is skill-defined.
artifacts	array	OPTIONAL	Array of Artifact objects (see Section 5.2).
error	object	OPTIONAL	Structured error: { code, message, retryable }.

`meta`	`object`	OPTIONAL	Arbitrary key-value metadata. Not interpreted by the protocol.

INFO

The SDK constructs envelopes automatically when calling request(), respond(), or emit(). The v, id, ts, from, and trace fields are always set by the SDK. Developers only need to populate payload and optionally artifacts and meta.

5.2 Artifacts

Artifacts are structured file or data attachments carried inside a respond message. Each artifact object contains:

- `id` (string, required) — unique identifier within this message.
- `name` (string, required) — human-readable filename or label.
- `mime_type` (string, required) — IANA media type of the content.
- `data` (string, optional) — base64-encoded inline content for small payloads.
- `uri` (string, optional) — URI pointing to out-of-band content (e.g. object storage).
- Implementations MUST provide exactly one of `data` or `uri`.

5.3 Distributed Tracing

The trace field carries W3C-compatible distributed tracing context. It MUST be present on every envelope. Fields:

- `trace_id` (string, required) — 16-byte hex trace ID, consistent across all messages in one originating request chain.
- `span_id` (string, required) — 8-byte hex span ID, unique to this individual message.
- `parent_span_id` (string, optional) — span ID of the parent message. Absent on root spans.
- `sampled` (boolean, optional) — hint to backend collectors.

NOTE

When an agent receives a request and issues its own downstream requests (delegation pattern), it MUST set parent_span_id on each outbound message to its own span_id, and MUST propagate the same trace_id. This links A→B→C chains in distributed tracing backends.

SECTION 6

The Registry

The Registry is the discovery backbone of Synapse. It stores agent manifests in a JetStream key-value bucket, indexed by agent ID. Discovery queries are served from an in-memory cache that is rebuilt from the bucket on startup.

6.1 Registration Flow

1. Agent sends a `register` message to `mesh.registry.register`.
2. Registry validates the manifest (required fields, schema conformance).
3. Registry stores the manifest keyed by agent ID in the JetStream KV bucket.
4. Registry returns a RegisterResponse with the agent ID and timestamp.
5. Agent begins sending heartbeats to `mesh.heartbeat.{{agent_id}}` every 30 seconds.

6.2 Discovery Filters

All filter fields in a discovery query use AND semantics. An agent must satisfy every specified criterion to appear in results.

Filter	Type	Behaviour
capabilities	string[]	Agent manifest.capabilities MUST be a superset of the requested list.
skill_ids	string[]	Agent MUST expose ALL listed skill IDs in manifest.skills.
availability	string	Exact match against manifest.availability.
max_cost_rq	number	Agent's manifest.cost.per_request MUST be ≤ this value (if present).
tags	object	All key/value pairs MUST appear in manifest.meta.
geo	string	Substring match against manifest.network.geo (ISO 3166).

6.3 Heartbeat and Liveness

Agents MUST publish a heartbeat message to `mesh.heartbeat.{{agent_id}}` at least every **30 seconds**. The heartbeat payload is an ISO 8601 timestamp.

Event	Threshold	Effect
Heartbeat received	—	Agent remains online in registry.
Missed heartbeats	45 s	Registry marks agent availability as "offline".
Extended absence	7 days	Registry purges the manifest from storage.
Reconnection	—	Agent sends register again; registry updates timestamp.

SECTION 7

Security Model

Security in Synapse is not an optional layer added on top of the protocol — it is a foundational design constraint. Every architectural decision, from how agents prove their identity to how messages are routed, was made with the assumption that the network will carry sensitive workloads in adversarial environments. This section describes each security mechanism in full.

NOTE

Application code in Synapse never handles credentials, session tokens, or encryption keys. All identity verification and access control happens at the transport layer, before any message is delivered to an agent. Developers write business logic; the protocol handles security.

7.1 Security-by-Design Principles

Synapse's security architecture is built on six non-negotiable principles:

- **Zero-trust identity.** No agent is trusted by default. Every connection must be cryptographically authenticated before any message is accepted.
- **Infrastructure-level enforcement.** Security controls are enforced by the NATS server at the transport layer — not by application code that could be bypassed or misconfigured.
- **Least privilege.** Each agent is granted the minimum set of subject permissions required to perform its role. Overly broad permissions are a configuration error.
- **Hard tenancy boundaries.** Different organisations operate in separate, cryptographically isolated NATS Accounts. Traffic cannot leak across account boundaries by accident.
- **Encryption in transit.** All production traffic MUST be encrypted. Plaintext connections are explicitly forbidden outside of local development.
- **Envelope integrity.** Every message carries a verifiable sender ID. Impersonation is prevented at the transport layer before the message reaches application code.

7.2 Cryptographic Identity — Ed25519 NKeys

Every participant in a Synapse network — operators, accounts, and individual agents — is identified by an **NKey**: a public/private keypair based on the **Ed25519** elliptic-curve signature algorithm. Ed25519 was chosen for its combination of strong security (128-bit security level), compact key sizes (32-byte private key, 32-byte public key), fast signature verification, and resistance to side-channel attacks.

NKeys encode public keys in a human-readable, Base32 format with a type prefix:

Prefix	Type	Example prefix	Used by
O	Operator	OA...	Infrastructure administrator
A	Account	AA...	Organisation or tenant
U	User	UA...	Individual agent process

The private key NEVER leaves the entity that holds it. All signing operations are performed locally. The public key is freely shareable and serves as the agent's permanent, unforgeable identifier. An agent's `id` field in its manifest IS its NKey public key.

INFO

ED25519 is used by OpenSSH, Signal, WhatsApp, and the NATS project itself. It has no known practical attacks, does not require random number generation during signing (unlike ECDSA), and verification is roughly 3x faster than RSA-2048. These properties make it well-suited to high-throughput agent networks.

7.3 Three-Tier Trust Hierarchy

Synapse uses NATS's three-tier trust model. Each tier's identity is cryptographically bound to the tier above it through JWT signatures, forming a verifiable chain of trust.

Tier	Entity	Signed by	Controls
Operator	Infrastructure administrator	Operator's own NKey (self-signed)	Which Accounts may exist on the server; global subject deny rules; server-level resource limits; TLS policy.
Account	Organisation or tenant	Operator NKey	Which User credentials are valid; subject export/import rules between Accounts; JetStream storage quotas; connection limits.
User	Individual agent process	Account NKey	Exact set of subjects the agent may publish to; exact set of subjects the agent may subscribe to; message size limits; connection count limits.

When an agent connects, the NATS server verifies the entire chain: User JWT signature (by Account NKey) → Account JWT signature (by Operator NKey). If any link in the chain is invalid, expired, or revoked, the connection is rejected before any messages are exchanged.

7.4 JWT Credentials and Verification

Each agent is issued a **NATS User JWT** — a signed JSON Web Token that encodes the agent's identity, permissions, and constraints. The JWT is presented during the NATS CONNECT handshake and verified by the server in microseconds.

JWT payload fields relevant to Synapse:

- `sub` — the agent's NKey public key (its permanent identity).
- `iss` — the Account NKey that signed this credential.
- `exp` — expiry timestamp. Expired JWTs are rejected even if the signature is valid.
- `nats.pub.allow` — whitelist of subjects the agent may publish to.
- `nats.sub.allow` — whitelist of subjects the agent may subscribe to.
- `nats.resp` — whether the agent may use NATS request/reply ephemeral subjects.
- `nats.subs` — maximum number of concurrent subscriptions.
- `nats.payload` — maximum message payload size in bytes.

Credentials can be rotated at any time by issuing a new JWT with a new expiry. The old JWT becomes invalid at its expiry without any coordination with the agent — the server enforces it automatically. This makes credential rotation safe and routine.

> **WARNING**
>
> User JWTs and NKey seed files MUST be stored outside of source control. Use secrets management tools (HashiCorp Vault, AWS Secrets Manager, environment variables injected at runtime). A compromised NKey seed allows full impersonation of the agent until the credential is revoked at the Account level.

7.5 Transport-Layer Encryption

All data in transit on a Synapse network MUST be encrypted in production. Synapse supports two encrypted transport modes:

Transport	Protocol	Port	Use case	Requirement
NATS TLS	TCP + TLS 1.2+	4443	Server-to-server, intranet agents	REQUIRED in production
WebSocket TLS	WSS (WS over TLS 1.2+)	443 / 4443	Browser agents, home networks	REQUIRED over untrusted networks
Leaf node TLS	TCP + TLS 1.2+	7422	Cross-organisation bridging	REQUIRED

NATS TCP	Plaintext TCP	4222	Local development only	MUST NOT be used in production

TLS certificates MUST be issued by a trusted certificate authority (CA). Self-signed certificates are acceptable in development but MUST NOT be used in production without proper CA validation. The minimum TLS version is **TLS 1.2**; **TLS 1.3** is strongly recommended for its improved handshake performance and forward secrecy guarantees.

> **WARNING**
>
> Any Synapse deployment that handles real agent workloads — regardless of whether it sits behind a private VPN or corporate firewall — MUST enable TLS. Plaintext connections expose both message content and NKey-signed authentication material to network-level attackers.

7.6 Subject-Level Permissions and Least Privilege

Beyond authentication, NATS enforces fine-grained **authorisation** over which subjects each agent may interact with. These permissions are embedded in the User JWT and enforced by the server on every publish and subscribe operation.

Recommended minimum permissions for a Synapse agent:

Operation	Permitted subjects	Purpose
Publish	`mesh.registry.register`	Register its manifest on startup
Publish	`mesh.registry.discover`	Query for other agents
Publish	`mesh.registry.deregister`	Deregister on shutdown
Publish	`mesh.heartbeat.{own_id}`	Send liveness heartbeats
Publish	`mesh.event.>`	Emit events to the network
Publish	`_INBOX.>`	NATS request/reply ephemeral subjects
Subscribe	`mesh.agent.{own_id}.inbox`	Receive incoming requests
Subscribe	`mesh.task.{own_id}.>.update`	Track task state changes
Subscribe	`mesh.event.>`	Receive subscribed events

An agent **MUST NOT** be granted publish rights to `mesh.agent.*.inbox` (i.e., the ability to write to any agent's inbox). It should only be able to publish to NATS request/reply reply subjects, which are

ephemeral and scoped to a single interaction. This prevents a compromised agent from impersonating messages to other agents.

7.7 Hard Multi-Tenancy and Account Isolation

In a multi-organisation Synapse deployment, each organisation runs in a separate **NATS Account** — a completely isolated messaging namespace. Accounts are cryptographically separated: subject names in Account A are entirely invisible to agents in Account B, even if those subjects have identical names.

Cross-account communication is only possible through explicit **export/import** rules configured by the Operator. These rules specify exactly which subjects are bridged between accounts and in which direction. Traffic never crosses an account boundary by accident.

Isolation property	Behaviour
Subject namespace	Fully isolated. mesh.event.> in Account A is a different subject to mesh.event.> in Account B.
JetStream storage	Separate. Each account has its own KV buckets, streams, and quotas.
Connection limits	Per-account. Account A's agents do not consume Account B's connection quota.
Cross-org traffic	Only via Operator-configured export/import rules on leaf node bridges.
Credential validity	Account-scoped. A User JWT issued by Account A cannot connect to Account B.

7.8 Envelope Integrity Verification

Every Synapse message envelope contains a `from` field that identifies the sending agent. This field is not merely a convention — it is verified by the platform against the authenticated agent ID established during the NATS connection handshake.

If the `from` field in an envelope does not match the authenticated NKey of the connected agent, the message is rejected with error code `3004 IDENTITY_MISMATCH`. This means an agent cannot forge messages on behalf of another agent, even if it has been granted publish rights to a shared subject.

INFO

While Synapse provides strong sender authentication at the transport layer, it does not currently provide end-to-end message encryption between agents (i.e., Agent A and Agent B both see plaintext payloads via the server). For workloads requiring payload-level confidentiality between agents, application-layer encryption of the payload field is recommended. Support for end-to-end payload encryption is planned for a future protocol version.

7.9 Security Checklist for Operators

Use this checklist when deploying a Synapse network in production:

#	Requirement	Severity
1	TLS 1.2+ enabled on all NATS listener ports (4222, 4443, 7422).	CRITICAL
2	Certificates issued by a trusted CA; self-signed certs not used in production.	CRITICAL
3	All agent User JWTs have an expiry date (exp field set).	CRITICAL
4	NKey seed files stored in secrets management (not in source control).	CRITICAL
5	Each agent granted only the minimum subjects required (least privilege).	HIGH
6	No agent granted wildcard publish to mesh.agent.*.inbox.	HIGH
7	Cross-org leaf node bridges configured with explicit subject export/import rules.	HIGH
8	JWTs rotated on a regular schedule (recommended: 90 days or less).	HIGH
9	Operator NKey seed stored offline / in HSM; never loaded into running software.	MEDIUM
10	NATS server access logs and authentication failure events sent to SIEM.	MEDIUM
11	Rate limits configured per agent (rate_limits manifest field).	MEDIUM
12	JetStream encryption-at-rest enabled for sensitive message storage.	MEDIUM

NOTE

If an agent credential is suspected to be compromised: (1) Revoke the User JWT at the Account level immediately — the agent will be disconnected within seconds. (2) Issue a new NKey pair and JWT. (3) Audit the NATS server access logs for any anomalous publish/subscribe activity during the suspected compromise window. (4) Rotate any API keys or secrets that the compromised agent had access to.

SECTION 8

Network Topology

8.1 Single-Organisation Deployment

The simplest topology is a single NATS server (or cluster) running within a trusted network. All agents connect to this server and communicate through it. No traffic leaves the network. A NATS cluster of three or more servers provides high availability with automatic leader election and JetStream replication.

8.2 Cross-Organisation Leaf Nodes

To bridge two organisations, one organisation's NATS server establishes an outbound **leaf node connection** to a shared hub NATS server. Key properties:

- The leaf node connection is outbound-only. No inbound ports need to be opened.
- Internal traffic between agents in the same organisation never traverses the hub.
- The hub administrator controls which subjects are exported/imported between accounts.
- The hub can be self-hosted or provided by a managed service such as Synadia Cloud.

8.3 WebSocket for Browser and Home Agents

Agents running in a browser or behind a NAT (e.g., a home laptop) cannot accept inbound TCP connections. NATS supports WebSocket listeners, allowing these agents to connect using an outbound-only WebSocket connection. The protocol is identical from the agent's perspective; only the transport changes.

Scenario	Protocol	Default Port	Notes
Intranet agents	NATS TCP	4222	Used within trusted networks. No TLS required in development.
Intranet agents (prod)	NATS TLS	4443	TLS-encrypted TCP for production.
Org to hub	Leaf node TLS	7422	Outbound only. Hub controls shared subjects.
Browser / home	WebSocket TLS	443 or 4443	WSS required for public endpoints.

NOTE

All connection types are outbound. Neither agents nor leaf nodes need to accept inbound connections. This eliminates firewall exposure and simplifies deployment in restrictive enterprise environments, cloud VPCs, and home networks alike.

SECTION 9

Error Codes

When an operation fails, the envelope's error field carries a structured error object: `{{ code: number, message: string, retryable: boolean }}`. Codes are grouped by category to aid programmatic handling. Codes **1xxx** are transport errors, **2xxx** are validation errors, **3xxx** are protocol-level errors, **4xxx** are capacity errors, and **5xxx** are internal server errors.

Code	Name	Retryable	Description
1001	TRANSPORT_TIMEOUT	Yes	The NATS request timed out before a reply was received. Use exponential backoff.
1002	TRANSPORT_NO_RESPONDERS	No	NATS returned a No Responders error — nobody is subscribed to that subject.
2001	INVALID_ENVELOPE	No	The received message could not be decoded or failed schema validation.
2002	INVALID_MANIFEST	No	The agent manifest submitted to register is missing required fields or has invalid values.
3001	SKILL_NOT_FOUND	No	The agent does not expose the skill ID named in the request.
3002	AGENT_UNAVAILABLE	Yes	The target agent is offline. Retry after a backoff period.
3003	TASK_INVALID_TRANSITION	No	The requested state transition violates the task state machine (e.g., completed → working).
3004	IDENTITY_MISMATCH	No	The envelope's from field does not match the authenticated agent ID.
4001	OVERLOADED	Yes	The agent has hit its maximum concurrent task limit. Back off and retry.
4002	RATE_LIMITED	Yes	Too many requests in a given time window. Respect the Retry-After hint if present.
5001	INTERNAL_ERROR	Yes	An unhandled error occurred inside the agent or platform service.

INFO

When retryable is true, callers SHOULD use truncated exponential backoff: start at 100 ms, double on each attempt, add random jitter, and cap at 10 s. TRANSPORT_TIMEOUT and AGENT_UNAVAILABLE suggest the agent may return. OVERLOADED and RATE_LIMITED indicate the agent is alive but busy — reducing request frequency is more effective than rapid retries.

SECTION 10

Agent Manifest

The agent manifest is the authoritative self-description of an agent. It is submitted during registration, stored in the Registry, and returned in discovery responses.

10.1 Manifest Fields

Field	Type	Required	Description
`id`	`string`	REQUIRED	Agent's unique ID. MUST be the NKey public key (56-character base32).
`name`	`string`	REQUIRED	Human-readable display name. UTF-8, max 128 characters.
`description`	`string`	OPTIONAL	Free-text description of the agent's purpose.
`version`	`string`	OPTIONAL	Agent-defined version string (semver recommended).
`protocol_version`	`string`	REQUIRED	Synapse protocol version supported. Set to "0.1.0".
`endpoint`	`string`	REQUIRED	NATS subject for the agent's inbox. Convention: mesh.agent.{id}.inbox.
`availability`	`string`	REQUIRED	Current availability: "online", "busy", or "offline".
`last_heartbeat`	`string`	REQUIRED	ISO 8601 timestamp of last heartbeat. Set by Registry.
`capabilities`	`string[]`	OPTIONAL	Broad skill categories for coarse filtering (e.g., "translation", "code-review").
`skills`	`Skill[]`	OPTIONAL	Array of Skill objects (see Section 10.2). Used for precise discovery.
`cost`	`object`	OPTIONAL	Pricing: { per_request?: number, per_token?: number, currency: string }.
`network`	`object`	OPTIONAL	{ ip_type?: "residential"|"datacenter"|"mobile"|"proxy", geo?: string }.

`rate_limits`	`object`	OPTIONAL	{ requests_per_second?, requests_per_minute?, concurrent_tasks? }.
`meta`	`object`	OPTIONAL	Arbitrary string key/value pairs. Searchable via discovery tags filter.

10.2 Skill Objects

Each entry in the skills array describes a discrete capability the agent can perform. Skills are the primary matching unit during discovery.

Field	Type	Required	Description
`id`	`string`	REQUIRED	Unique identifier within this agent. Used to route onRequest handlers.
`name`	`string`	REQUIRED	Human-readable skill name.
`description`	`string`	OPTIONAL	What this skill does.
`input_modes`	`string[]`	OPTIONAL	Accepted MIME types for the payload.input field (e.g., "text/plain", "application/json").
`output_modes`	`string[]`	OPTIONAL	Produced MIME types for the payload.output field.
`meta`	`object`	OPTIONAL	Skill-specific metadata (e.g., supported languages, model name).

SECTION 11

Composition Patterns

The six primitives can be composed into higher-order patterns. None of these patterns require new message types or protocol extensions — they are conventions built entirely from the existing primitives.

11.1 Connect (Persistent Session)

Two agents establish a persistent bilateral session. Agent A sends a `request` with skill ID `"connect"`. Agent B responds with `"accepted"` and the agreed `context_id`. Subsequent messages between A and B carry the same context_id, allowing the Task Manager to group them into a single logical session.

11.2 Delegate

Agent A sends a request to Agent B. B determines that Agent C is better suited and issues its own `request` to C, forwarding A's `context_id` and setting `parent_span_id` to its own span. B then waits for C to respond before responding to A. The trace chain A→B→C is preserved in the distributed tracing backend.

11.3 Broadcast

Agent A discovers N agents with a given capability using the discover primitive, then sends N parallel requests — one to each discovered agent. A collects responses as they arrive. This pattern is suited to voting, consensus, fan-out data processing, and parallel summarisation.

> **INFO**
>
> When collecting broadcast responses, use the map task_id → response to deduplicate. If an agent responds more than once (e.g., due to retried delivery), later messages with the same task_id targeting a terminal state MUST be ignored.

11.4 Stream

An agent sends incremental results by publishing a series of messages to `mesh.task.{{task_id}}.stream`. Each stream message carries a partial payload (e.g., a token from an LLM). The stream is terminated by a final `respond` message that moves the task to `completed` or `failed`. Consumers MUST subscribe to the stream subject before sending the request

to avoid missing early tokens.

SECTION 12

Reference Implementation

The canonical reference implementation is the **synapse-agent** TypeScript/Node.js SDK. It exposes a high-level API that handles envelope construction, task tracking, heartbeating, backoff retries, and distributed tracing automatically. The following example demonstrates a complete agent.

Complete Agent Example

```
// synapse-agent SDK — TypeScript / Node.js 18+
import { Synapse } from "synapse-agent";
// 1. Connect to the Synapse network
const net = await Synapse.connect("ws://your-nats-server:4443");
// 2. Register this agent
await net.register({
name: "Translator",
capabilities: ["translation"],
skills: [{
id: "translate",
name: "Translate",
description: "Translate text to a target language",
input_modes: ["text/plain"],
output_modes: ["text/plain"],
}],
});
// 3. Declare a request handler
net.onRequest("translate", async (payload) => {
const { text, target_lang } = payload.input as {
text: string;
target_lang: string;
};
const translated = await translateText(text, target_lang);
return { text: translated, language: target_lang };
});
// 4. Discover and call another agent
const { agents } = await net.discover({ capabilities: ["spell-check"] });
if (agents.length > 0) {
const result = await net.request(agents[0].id, "check", {
text: "Sanity check this translation",
});
console.log(result.payload.output);
```

```
}
// 5. Subscribe to events
net.subscribe("document.>", (event) => {
console.log(`Event: ${event.event_type}`, event.data);
});
console.log("Translator agent online");
```

APPENDIX A

Complete Envelope Schema

```
// Synapse Message Envelope — JSON Schema (TypeScript notation)
interface Envelope {
v: string; // "0.1.0"
id: string; // UUID v7
type: PrimitiveType; // "register" | "discover" | "request" | "respond" | "emit"
ts: string; // ISO 8601 UTC
from: string; // sender agent ID
to?: string; // recipient agent ID
task_id?: string; // UUID v7
in_reply_to?: string; // UUID v7
context_id?: string; // session grouping key
trace: TraceContext; // always present
payload?: unknown; // skill-defined
artifacts?: Artifact[];
error?: StructuredError;
meta?: Record<string, string>;
}
interface TraceContext {
trace_id: string; // 32-char hex
span_id: string; // 16-char hex
parent_span_id?: string; // 16-char hex
sampled?: boolean;
}
interface Artifact {
id: string;
name: string;
mime_type: string;
data?: string; // base64-encoded inline
uri?: string; // out-of-band URL
}
interface StructuredError {
code: number;
message: string;
retryable: boolean;
}
```

APPENDIX B

Full Error Code Registry

Code	Constant	Category	Retryable	Description
1001	TRANSPORT_TIMEOUT	Transport	Yes	NATS request timed out waiting for a reply.
1002	TRANSPORT_NO_RESPONDERS	Transport	No	No agents are subscribed to the target subject.
1003	TRANSPORT_DISCONNECT	Transport	Yes	Agent lost NATS connection mid-operation.
2001	INVALID_ENVELOPE	Validation	No	Incoming message failed JSON parse or schema validation.
2002	INVALID_MANIFEST	Validation	No	register payload failed manifest schema validation.
2003	INVALID_DISCOVER_QUERY	Validation	No	discover payload contains invalid filter fields.
2004	ENVELOPE_VERSION_MISMATCH	Validation	No	Envelope v field is not compatible with this server.
3001	SKILL_NOT_FOUND	Protocol	No	Named skill ID does not exist in the agent's manifest.
3002	AGENT_UNAVAILABLE	Protocol	Yes	Target agent is offline or not reachable.
3003	TASK_INVALID_TRANSITION	Protocol	No	Attempted state transition violates the task state machine.
3004	IDENTITY_MISMATCH	Protocol	No	from field does not match authenticated agent ID.
3005	TASK_NOT_FOUND	Protocol	No	task_id does not match any known task.

4001	`OVERLOADED`	Capacity	Yes	Agent has hit its max_concurrent_tasks limit.
4002	`RATE_LIMITED`	Capacity	Yes	Caller has exceeded the agent's rate_limits thresholds.
4003	`PAYLOAD_TOO_LARGE`	Capacity	No	Envelope payload exceeds the server-configured size limit.
5001	`INTERNAL_ERROR`	Internal	Yes	Unhandled exception inside the agent or platform service.
5002	`REGISTRY_UNAVAILABLE`	Internal	Yes	Registry service is temporarily unavailable.
5003	`STORAGE_ERROR`	Internal	Yes	JetStream write or read failure.

End of Synapse Protocol Specification v0.1.0 — 12 June 2026