CMB Architecture Version 3

Author: Otto L. Lecuona

Date: 12/31/2025

Table of Contents

Introduction

The **Cognitive Message Bus (CMB)** is the communication backbone of an AGI system, enabling modular components to exchange information seamlessly. It functions as a **message bus**, which is a combination of a common data model, a common command set, and a messaging infrastructure allowing different systems (modules) to communicate through shared interfaces. The CMB decouples modules by providing an asynchronous, channel-based messaging system.

In **Version 2** of the CMB architecture, improvements have been incorporated to enhance completeness, clarity, and robustness based on prior feedback. These include clearer channel definitions, a standard message format with priorities and expiration, code examples for usage, and provisions for future enhancements (like security and direct addressing).

Architectural Principles

The bus is an integration contract, not a convenience API. Specific channels establish a cognitive connection between 2 or more modules within the system. The channel connection is selects the context of the contract.

Integration Contract

In the context of computer communication, an **integration contract** is a formal specification that defines how different modules or systems interact and exchange information. It is not just a convenience API, but a set of rules and expectations that participating components must follow to communicate effectively. In the CMB Architecture Version 3, the bus itself acts as the integration contract, establishing the rules for how modules connect and communicate.

- **Channels as Contracts:** Specific channels within the bus act as cognitive connections between two or more modules. The channel connection itself is the contract—modules agree to send and receive messages according to the channel's protocol and message format.
- **Standardization:** The contract covers aspects like message structure (standardized format, metadata, priorities, expiration), channel definitions (each channel has a specific role and topic domain), routing and addressing (how messages are delivered to intended recipients), extensibility (how new modules or channels can be added without breaking existing communication), and diagnostics/logging (how communication is monitored and recorded for transparency and debugging).
- **Decoupling:** This approach ensures that modules are decoupled—they don't need to know each other's internal details, only how to communicate via the contract. This makes the system more robust, scalable, and easier to maintain.

Architectural Intent

The Cognitive Message Bus is designed to behave as a **policy-enforcing, identity-aware communication fabric**, not a collection of point-to-point connections. This enables:

- Centralized observability
- Deterministic routing
- Scalable module composition
- Future hardware mapping
- Robust fault isolation

All future extensions to the CMB must preserve these invariants.

Design Goals and Key Features

- **Decoupled Communication:** Modules do not call each other directly. Instead, they send messages over the bus, ensuring loose coupling and modularity. This allows modules to be added or removed with minimal impact on others.
- **Asynchronous Messaging:** Communication is asynchronous via **publish/subscribe** and **push/pull** patterns, so senders and receivers operate independently. No module is blocked waiting for another, improving concurrency and system responsiveness.
- **Multiple Channels (Topic Domains):** Messages are organized into distinct **channels** by topic/domain (e.g. Control, Perception, Memory). This segmentation prevents irrelevant traffic from reaching modules and helps manage complexity. Each channel has its own message router and topic namespace, supporting parallel message flows.
- **Standard Message Structure:** All messages adhere to a common format (the `CognitiveMessage`), which includes metadata like source, targets, priority, etc. This ensures a **canonical data model** for communication. The consistent format simplifies processing and logging of messages system-wide.
- **Extensibility and Scalability:** New modules or channels can be introduced without altering the core bus logic. The underlying ZeroMQ-based infrastructure is network-capable (TCP sockets), allowing distribution of modules across processes or machines. As long as modules know the channel and message schema, they can participate.
- **Diagnostics and Logging:** A dedicated diagnostic channel (and logging module) exists to monitor and record bus traffic. Every message can be logged or inspected for debugging or analysis, supporting system transparency and introspection.
- **Prioritization and TTL:** Each message carries a priority level and a time-to-live (TTL). While not all v2 components fully utilize these yet, they lay the groundwork for future enhancements like prioritizing urgent messages and discarding stale messages automatically.

Separation of Concerns

The CMB enforces strict separation between:

- **Semantic channels** (what the message means)
- **Transport mechanics** (how the message is delivered)
- **Routing policy** (where the message goes)

Semantic meaning belongs in the **message schema**, not in: - Port numbers - Socket types - Hardcoded topology

Improvements in Version 3

CMB Architecture Version 3 incorporates several enhancements over the initial design to address completeness and expand functionality:

- **Comprehensive Channel Set:** All original channels (CC, SMC, VB, BFC, DAC, EIG, PC, MC, IC, TC) are included with clear definitions, ensuring full coverage of the agent's cognitive spaces. New channels can be added in a similar fashion if new domains of communication are identified. The central port mapping config makes this straightforward.
- **Standardized Message Schema:** The `CognitiveMessage` dataclass formalizes the message content, adding fields like `priority`, `ttl`, and `signature` that were absent or implicit before. This makes messages self-descriptive and ready for future use cases (e.g., expiring unhandled messages, authenticating senders).
- **Code Snippets and Examples:** Version 2 documentation (as seen above) now includes concrete code examples showing how to send and receive messages. This addresses previous ambiguity by demonstrating usage patterns of the CMB API (ModuleEndpoint, message creation, etc.) in a real context.
- **Diagnostic Logging Mechanism:** The introduction of the Diagnostic channel (DAC) and a conceptual logging module (`cmb_logger`) provides a built-in way to capture and inspect the internal message flow. This was a recommended addition to improve observability – now every significant action can be emitted as a message and recorded, facilitating debugging and even training data for meta-cognitive analyses.
- **Threaded Router and Concurrency:** The router component is designed to run on a separate thread per channel, which improves the system's ability to handle multiple channels concurrently without bottlenecking the entire bus. This multi-threaded (or multi-process) architecture was an improvement to ensure scalability as more channels and higher message volumes are used.
- **Foundation for Identity-Based Routing:** By using ZeroMQ's ROUTER socket for input, the system is ready to leverage advanced routing patterns. Future versions could assign permanent identities to module endpoints, enabling direct addressability or request-reply semantics through the bus (where a message can specify a reply-to identity). Version 2 lays this groundwork, even though the current logic treats messages in a simple publish-subscribe manner.
- **Error Handling and Stability:** Recommendations to handle errors more gracefully have been applied. The router's loop catches exceptions during routing to prevent crashes and logs errors. The `ModuleEndpoint` will block or time out on receive, and send operations can be designed to use non-blocking sends with checks (for instance, the GUI demo

catches exceptions if the send fails due to router unavailability). These practices increase the robustness of inter-module communication.

- **Extensibility for Security:** With the `signature` field in place and the modular structure, adding security layers is more feasible. A future update might include encryption of message payloads or signatures verification on the receiving side (to ensure only authorized modules communicate or to detect tampering). The architecture can evolve to include a **security broker** or **authentication service** on a special channel (or integrated with each router) that verifies credentials of modules when they connect.

Channels and Routing

CMB Channels are logical communication pathways, each identified by an acronym and served by its own message router. Version 3 continues to use the original channels from version 1 and adds any needed new ones. Table below lists the core channels in the system and their roles:

- **CC (Control Channel):** Used for high-level control signals and directives. For example, the Executive module sends commands (e.g., *"start behavior X"*) to subordinate modules via CC.
- **SMC (Symbolic Message Channel):** Handles symbolic or discrete knowledge exchange. Cognitive reasoning modules might share symbolic facts, NLP insights, or logic statements over SMC for higher-level reasoning.
- **VB (Vector Bus):** Carries vectorized data (embeddings, sensory feature vectors, etc.). Perception modules publish processed sensor data (like image feature vectors or audio spectrograms) on the VB for consumption by cognitive or memory modules.
- **BFC (Behavioral Flow Channel):** Manages sequences of actions or behaviors. Planning and behavior coordination messages (task status, next action triggers) flow through BFC to ensure complex behaviors are executed in order.
- **DAC (Diagnostic and Awareness Channel):** Used for diagnostics, logging, and self-awareness signals. Modules send status updates, heartbeat messages, or logs to this channel. A special **CMB Logger** module on DAC records all important events, enabling system monitoring and introspection.
- **EIG (External Interface Gateway):** Interface to the external world or external systems. Any inbound commands from a user interface or API, and outbound messages to external services, pass through EIG. This isolates external I/O at a single gateway channel.
- **PC (Perception Channel):** Conveys raw perceptual inputs and low-level sensory data. Perceptual modules (vision, auditory, etc.) publish their observations or detections on PC, which may be consumed by memory or interpretive modules for further processing.
- **MS (Memory Channel):** Dedicated to memory storage and retrieval operations. Queries to the memory module, memory recall results, or knowledge base updates are transmitted via MC so that other modules can stay informed of changes in the knowledge state.
- **IC (Introspection Channel):** Used by introspective processes that evaluate or analyze the agent's own cognitive state. For example, an introspection module might request explanations for decisions or check system consistency over IC, with relevant modules responding on the same channel.
- **TC (Threat Channel):** Reserved for threat detection and mitigation messages. If any module (or an external sensor) perceives an anomaly or threat, it sends an alert on TC. Security or safety modules subscribe to TC to take appropriate action (such as shutting down a component or alerting an operator).

These channel definitions are reflected in the central configuration. Each channel is assigned a unique base TCP port for its router (e.g. `"CC": 6001` for Control Channel). By default, the router for a channel uses two ports: one for incoming messages and the next for outgoing. For example, if the Control Channel base is 6001, the router listens for incoming messages on `6001` and publishes outgoing messages on `7001`. This scheme ensures no port conflicts and a known mapping from channel name to network port.

Channel Summary

Channel	Name	Purpose	Primary Layers
CC	Control Channel	System-level directives and orchestration	Executive
SMC	Symbolic Message Channel	Symbolic reasoning, language, logic	Cognitive
VB	Vector Bus	High-bandwidth vector and embedding data	Perception, Cognitive
BFC	Behavioral Flow Channel	Behavior execution and sequencing	Behavior
DAC	Diagnostic & Awareness Channel	Health, logging, awareness	Awareness
EIG	External Interface Gateway	External IO boundary	Interface
PC	Perception Channel	Raw and processed sensor data	Perception
MS	Memory Channel	Memory queries and storage	Memory
IC	Introspection Channel	Self-reflection and self-questioning	Awareness
TC	Threat Channel	Safety-critical alerts	Executive

Channel Rules

- A message is sent on exactly one channel
- Channels must not be repurposed
- Modules may participate in multiple channels
- Channel choice encodes intent

Naming Conventions and Port Assignments

Port Range	Channel	Router Socket	Module Socket	Logical Direction
60XX	CMB Channels (Ingress)			
6001	CC	ROUTER	DEALER	Ingress
6002	SMC	ROUTER	DEALER	Ingress
6003	VB	ROUTER	DEALER	Ingress
6004	BFC	ROUTER	DEALER	Ingress
6005	DAC	ROUTER	DEALER	Ingress
6006	EIG	ROUTER	DEALER	Ingress
6007	PC	ROUTER	DEALER	Ingress
6008	MS	ROUTER	DEALER	Ingress
6009	IC	ROUTER	DEALER	Ingress
6010	TC	ROUTER	DEALER	Ingress
70XX	Fanout Channels(Egress)			
7001	CC	PUB	SUB	Egress
7002	SMC	PUB	SUB	Egress
7003	VB	PUB	SUB	Egress
7004	BFC	PUB	SUB	Egress
7005	DAC	PUB	SUB	Egress
7006	EIG	PUB	SUB	Egress
7007	PC	PUB	SUB	Egress
7008	MS	PUB	SUB	Egress
7009	IC	PUB	SUB	Egress
7010	TC	PUB	SUB	Egress
61XX	Acknowledge (Directed/Shared)			
6101	ACK Ingress	ROUTER	DEALER	Ingress
6102	ACK Egress	ROUTER	DEALER	Egress
62XX	Lifecycle/control			
6200	Registration	ROUTER	DEALER	Bidirectional (via router)
6201	Control/Shutdown	ROUTER	DEALER	Bidirectional (via router)

Channel Ownership

- **Ports identify channels, not participants**.
- A channel represents a semantic communication purpose (e.g., COMMAND, BFC, DAC), not a specific module.
- Multiple modules may connect to the same channel using unique identities.

Channel membership is determined by subscription and identity, not by port allocation.

Lifecycle and Control Channels

- Lifecycle channels are reserved for:
  - Module registration
  - Heartbeats and liveness
  - Shutdown and control commands
- These channels are bidirectional **via the router only**.
- Modules must never issue lifecycle commands directly to other modules.

This design enables safe startup, shutdown, and fault handling.

CMB Router

For each channel, a **CMBRouter** instance is responsible for shuttling messages from senders to receivers. The router binds a **ROUTER** (or PULL) socket to the channel's input port to receive messages, and a **PUB** socket to the output port to broadcast messages to subscribers. The router is channel-agnostic – it does not inspect message content beyond looking at the target addresses. It simply ensures that any message arriving on the input is published on the output with the appropriate topic label.

Under the hood, when a module sends a message into the bus, it goes to the channel's ROUTER socket. The router then wraps or queues the message and republishes it via the PUB socket. Subscribers (modules) on that channel receive it if they are subscribed to the matching topic. In the current design, the **message's target field is used as the PUB topic**. The router uses the list of `targets` in the message to send a copy to each target name. Each target name becomes the topic for one PUB message containing the original message bytes. Modules subscribe to their own name (or other relevant topic) to receive messages intended for them. This acts like a **direct addressing scheme** on the bus:

- If a message has `targets: ["behavior"]`, only the module(s) subscribed to topic `"behavior"` will get it (typically the Behavior module itself).
- If multiple targets are listed (e.g. `["memory", "behavior"]`), the router will publish it twice, once with topic `"memory"` and once with `"behavior"`, delivering to both modules.

- (Future feature:) A special target like `"all"` could be used for broadcast to every subscriber, though by default modules only subscribe to their own name. In version 2, broadcast could be achieved by having modules also subscribe to a shared topic (if configured) or by explicitly listing all intended recipients.

Router concurrency and identity: The router's receiving socket is a ZeroMQ ROUTER type, which allows addressing and asynchronous handling of multiple senders. In v2, the router currently doesn't use the identity feature beyond what ZeroMQ needs internally (we read and ignore the sender identity frame). However, this design decision paves the way for future enhancements such as:

- **Direct Request-Reply:** Modules could send a message and await a routed reply addressed back to them via the router (using the ROUTER/DEALER pattern).
- **Authentication or Filtering:** Identities could be used to authenticate modules or filter messages (e.g., only allow certain modules to send on certain channels).

Each router runs in its own thread (or process) for scalability. In the provided demo setup, a router is launched for the Control Channel to enable executive-behavior communication. In a full system, one would run a router instance per channel to activate the entire bus. This could be done by launching multiple router processes (one per channel) or a single process creating multiple router threads via the `CMBRouter.start()` method. Once running, routers require no further intervention – they continuously forward messages as they arrive.

Router Authority

- The **CMB Router is the sole authority that binds network ports**.
- All modules, including GUI, Executive, Behavior, Memory, and Cognitive subsystems, **must connect only**.
- Modules **must never bind ports**, regardless of channel type.

This invariant ensures centralized control, predictable topology, and consistent enforcement of routing, logging, and policy decisions.

Identity-Based Routing

- All directed communication relies on **ZeroMQ socket identity**, not port numbers.
- Each module **must set a stable, human-readable identity** equal to its logical module name.
- No two live modules may share the same identity on the same channel.

Socket identity is a transport-level routing label and is **not part of the application payload**.

Ingress and Egress Terminology

Ingress

- **Ingress** refers to traffic **entering the router**.
- Ingress ports are always **bound by the router**.
- Modules send messages into ingress ports using **DEALER sockets**.

Ingress channels represent requests, commands, data submissions, or state updates originating from modules.

Egress

- **Egress** refers to traffic **leaving the router**.
- Egress ports are always **bound by the router**.
- Modules receive messages from egress ports using **SUB sockets** (for fanout) or **DEALER sockets** (for directed delivery).

Egress channels are used to distribute routed messages, broadcast events, or deliver responses.

Bidirectional (via Router)

- Bidirectional communication **never implies direct module-to-module connections**.
- All bidirectional flows are implemented as:
    - Module → Router (Ingress)
    - Router → Module (Egress)

This applies primarily to lifecycle, control, and acknowledgment channels.

Socket Usage Rules

ROUTER / DEALER (Directed Communication)

- ROUTER/DEALER is used for **all directed communication**, including:
    - Channel ingress
    - ACKs and responses
    - Lifecycle control
- ROUTER sockets provide identity-aware routing and policy enforcement.
- DEALER sockets are asynchronous, non-blocking endpoints with no imposed protocol.

Directed traffic must **never** use PUB/SUB.

PUB / SUB (Fanout Communication)

- PUB/SUB is used **only** for one-to-many fanout.
- PUB sockets are bound by the router.
- SUB sockets are used by modules to receive broadcast messages.

PUB/SUB is appropriate for: - Fanout of semantic channels - Events - Logging streams

PUB/SUB must **never** be used for acknowledgments, replies, or control signals.

Summary of Binding Rules (Quick Reference)

- Router binds all ports
- Modules only connect
- Ports identify channels
- Identities identify modules
- ROUTER/DEALER for directed traffic
- PUB/SUB for fanout only
- ACKs are directed and shared
- No module-to-module sockets

Module Endpoints

Modules interface with the CMB through a **Module Endpoint**, which abstracts the underlying ZeroMQ sockets. The `ModuleEndpoint` class manages a pair of sockets: a **PUSH socket** for sending messages into the bus, and a **SUB socket** for receiving messages from the bus. This hides the complexity of socket setup and provides simple `send()` and `receive()` methods to the module developer.

When a module starts up, it creates a ModuleEndpoint, providing its unique module name and specifying which channel's ports to connect to. For example, a Behavior module that listens on the Control Channel (CC) might configure its endpoint as follows:

```
cc_pub = get_channel_port("CC") + 1  # outbound PUB port for CC
cc_push = get_channel_port("CC") + 0  # inbound PUSH/ROUTER port for CC
behavior_endpoint = ModuleEndpoint("behavior", pub_port=cc_pub,
push_port=cc_push)
```

Source: behavior_stub.py

In this snippet, `ModuleEndpoint("behavior", ...)` subscribes its SUB socket to topic `"behavior"` on the Control Channel's PUB port and connects its PUSH socket to the Control Channel's input port. From this point on, the **Behavior module** can receive any message addressed to `"behavior"` on CC, and it can send messages out via CC by calling the endpoint's send function. Similarly, an **Executive module** or any other module would instantiate its endpoint with the appropriate channel ports and module name.

Sending a message: To send a message, a module creates a `CognitiveMessage` object (or uses the `CognitiveMessage.create()` helper) and calls the endpoint's `send(message)`. The endpoint handles serializing the message to bytes and pushing it to the channel router. For example, the Executive module might do the following to command the Behavior module:

```
# In Executive module, send a directive to Behavior over Control Channel
msg = CognitiveMessage.create(
    source="executive",
    targets=["behavior"],
    payload={"directive": "start_behavior", "behavior": "explore_area"},
    priority=70
)
executive_endpoint.send(msg)
```

Source: executive_stub.py

Here, the Executive's message specifies its own name (`source="executive"`) and the intended recipient (`targets=["behavior"]`), along with a payload containing the instruction details. The `priority=70` indicates this is a high-priority message (on a scale that typically defaults to 50). When `executive_endpoint.send(msg)` is called, the message is forwarded to the CC router, which will route it to the Behavior module as described earlier. The sending call is non-blocking; the Executive can continue doing other work without waiting for a response.

Receiving a message: On the reciving side, the Behavior module's endpoint will deliver the message to the behavior when it calls `receive()`. The module might have a loop waiting for incoming messages. For example:

```
# In Behavior module, receiving messages from Control Channel
msg = behavior_endpoint.receive()  # blocking call, waits for next message
print( f"Received message from {msg.source}: {msg.payload}")
# ... process the message ...
```

Source: behavior_stub.py

In this snippet, `behavior_endpoint.receive()` blocks until a message tagged for `"behavior"` is published by the router. The returned object `msg` is a `CognitiveMessage` instance that the Behavior module can inspect. In this case, it would find `msg.source == "executive"` and `msg.payload == {"directive": "start_behavior", "behavior": "explore_area"}` as sent above. The behavior module would then act on that directive (e.g., initiate the requested behavior).

Publish/Subscribe Mechanism: Note that a module only receives messages for topics it subscribes to. By default, `ModuleEndpoint` subscribes to the module's own name (ensuring it gets direct messages). Modules can subscribe to additional topics if needed by creating additional `ModuleEndpoint` instances or by extending the subscription (the current implementation binds one subscription per endpoint). For instance, a logging or monitoring module might subscribe to multiple modules' topics or use wildcards (if supported) to capture broader traffic. This is analogous to a **Selective Consumer** pattern where each module is only interested in certain message types or senders.

Multiple Senders and Receivers: The CMB supports multiple modules sending simultaneously. ZeroMQ's non-blocking sockets and the router design allow concurrent message emission. The router will fairly queue and distribute messages to subscribers. Likewise, multiple subscribers can receive the same published message if they share the topic. For example, if two different modules both subscribe to `"memory"` on the Memory Channel, and the memory module publishes an update targeted to `"memory"`, both subscribers would receive it (assuming the memory module uses a generic target like a category; however, typically modules target specific recipients to avoid unintended listeners).

Messaging Communications Model

Cognitive Message Format

All CMB communication uses the **CognitiveMessage** schema. This schema is mandatory and versioned.

Message Structure (Python)

```python
@dataclass
class CognitiveMessage:
    message_id: str              # Global unique identifier
    schema_version: str          # Message schema version
    msg_type: str                # Semantic intent
    msg_version: str             # Message-type version
    source: str                  # Sending module
    targets: list[str]           # Intended recipients
    context_tag: str | None      # Goal / task context
    correlation_id: str | None   # Request–response linkage
    payload: dict                # Message content
    priority: int                # 0–100
    timestamp: float             # Epoch seconds
    ttl: float                   # Time-to-live (seconds)
    signature: str | None        # Optional integrity/auth
```

This message schema is standardized to ensure interoperability across modules. In version 2, it is implemented as a Python dataclass for convenience and clarity. The key fields in a CognitiveMessage include:

- **message_id:** A unique identifier (UUID) for the message instance, allowing tracking and correlation of messages.
- **schema_version**; Enables controlled evolution of the bus. Provides compatibility.
- **msg_type:** Primary semantic discriminator (no payload inference).
- **msg_version:** Allows evolution of message-specific schemas
- **source:** The name of the module that generated the message (e.g., "executive"). Recipients can use this to understand who sent the information or to send a response back.
- **targets:** A list of one or more target module names for whom the message is intended (e.g., ["behavior"]). These correspond to subscription topics on the bus. Multiple targets can be specified for multi-cast; if the list contains "all" or similar convention, it could be used for broadcast (this convention can be defined by the system).
- **context_tag:** Identifies the active goal, plan, or episode.
- **correlation_id:** Binds multi-step flows and responses.

- **payload:** A dictionary containing the content of the message. This can be any JSON-serializable data structure (text, numbers, lists, nested dicts). The payload carries substantive information or command — for example, a directive, a sensory observation, or a query result.
- **priority:** An integer indicating the message priority or importance. By default, messages might have a priority of 50 (neutral), while critical messages could have higher values. In future, routers or modules could use priority to order message processing or to decide dropping low-priority messages under load.
- **timestamp:** A sending time (epoch time in seconds) recorded when the message is created. This can be used for measuring latency or ordering events.
- **ttl (Time-To-Live):** A duration (in seconds) that the message is considered valid. For example, ttl=10.0 means the message content expires 10 seconds after its timestamp. Receivers or routers can check this field to ignore or discard stale messages. In the current implementation, a helper method is_expired() is provided to check if the message's TTL has elapsed. Future versions might have routers automatically drop expired messages instead of delivering them.
- **signature:** A field for a cryptographic signature or hash. This is currently an empty string by default, but the intent is to allow messages to be signed for authenticity and integrity. In a future iteration, sending modules could sign the payload (or the entire message) with a private key, and receiving modules (or a security layer) could verify the signature to ensure the message was not tampered with and truly comes from the claimed source.

The `CognitiveMessage` class also provides convenience methods to convert to/from JSON or bytes for transmission. For instance, `to_json()` and `to_bytes()` serialize the message, and `from_bytes()` reconstructs a `CognitiveMessage` from raw bytes. The CMB uses these to send messages over sockets. Internally, when a message is sent via `ModuleEndpoint.send()`, it calls `CognitiveMessage.to_bytes()` and the router uses `CognitiveMessage.from_bytes()` when receiving on the other end. This ensures that the message structure remains consistent and no information is lost in transit.

By enforcing a standard message format, the CMB architecture ensures that all modules **"speak the same language."** This is crucial for an integrative AGI system – perception outputs, executive commands, memory queries, etc., all share a common envelope, making it easier to log, debug, or extend the system.

Message Types

The examples only reference the msg_type component of the CognitiveMessage sent on the CMB. The complete message contains all the field defined for the CognitiveMessage.

Directive Messages

msg_type: `directive.start_behavior`

```
{
  "msg_type": "directive.start_behavior",
  "payload": {
  "behavior_name": "explore_area",
  "parameters": {}
  }
}
```

Purpose: Instruct downstream modules to initiate controlled actions.

Query / Response Messages

- `memory.retrieve`
- `memory.store`

Responses must include `correlation_id` referencing the request.

Diagnostic Messages

- `diagnostic.status`
- `diagnostic.alert`

Emitted on DAC for awareness and logging.

Message Enforcement and Validation

- Messages missing required fields are invalid
- msg_type is mandatory and authoritative
- Routers may reject expired or malformed messages
- Payload must conform to msg_type schema

Acknowledgment (ACK)

ACK Channel Scope

- ACKs are **directed messages**, not broadcasts.
- ACKs use shared ROUTER/DEALER channels, not per-channel ports.
- There is exactly:
    - One ACK ingress channel (Module → Router)
    - One ACK egress channel (Router → Module)

This avoids port explosion and duplicated logic.

Design Assumptions (Explicit)

- **Router is transport authority**
    - Confirms receipt
    - Confirms delivery
    - Logs all transitions
- **Execution authority belongs to the receiving module**
- **Sender owns orchestration and timeout logic**
- **All ACKs are messages** (not socket-level signals)
- **All messages share a correlation_id**

This prevents:

- Tight coupling
- Hidden blocking
- Socket misuse
- Ambiguous responsibility

Each ACK must reference the original message via a **correlation_id**.

Where ACKs should travel

DO NOT send ACKs back on the same PUB channel.

ACKs should be:

- **Directed**
- **Routable**
- **Non-broadcast**

Use a **dedicated ACK / RESPONSE channel**:

- Socket type: DEALER → ROUTER → DEALER

This avoids:

- ACK storms
- Accidental fanout
- Feedback loops

Timers (Critical Design Detail)

Each phase has **independent timers**:

Timer	Purpose
router_ack_timer	Router responsiveness
delivery_ack_timer	Routing completion
execution_timer	Module execution

Timers **must not overlap ambiguously**.

This avoids a common bug:
"Execution timed out" when delivery never occurred.

Threading Model

Minimal Safe Model

- **One ACK State Machine thread per outbound request**
- Or:
    - One central event loop with correlation-based routing

NOT Recommended

- Blocking socket waits
- Shared mutable state without locks
- One thread handling multiple active exchanges without correlation

---

Why This Must Be a State Machine (Not Callbacks)

Without a state machine:

- ACK ordering becomes implicit
- Error handling becomes scattered
- Timeouts become unreliable
- Debugging becomes impossible

With a state machine:

- Every ACK has meaning
- Every failure is classified
- Logging becomes deterministic
- You can formally test it

Logging and Observability (Non-Optional)

Every transition should log:

[MSG_ID][STATE] → [STATE] (EVENT)

Example:

[abc-123] SEND_PENDING → ROUTED (ROUTER_ACK)

This gives you:

- Replayable traces
- GUI timeline views
- Patent-grade determinism

ACK message structure (concrete)

Use the **same CognitiveMessage schema**, just with:

```
{

  "msg_type": "ACK",

  "ack_type": "ROUTER_ACK | DELIVERY_ACK | EXECUTION_ACK|TIMEOUT|CANCEL",

  "status": "SUCCESS | FAILURE | REJECTED | IN_PROGRESS",
```

```
 "source": "behavior",

 "targets": ["executive"],

 "correlation_id": "same-as-original",

 "payload": {

  "details": "optional"

 }

}
```

Why correlation_id is mandatory

It lets the module / Executive:

- match ACK → command
- handle retries
- detect timeouts

What NOT to ACK (very important)

❌ Do **not** ACK:

- every PUB hop
- every internal queue operation
- every internal state change

To avert drowning in noise.

ACKs should represent **meaningful milestones**.

Minimal ACK policy

If you want the **smallest useful system**, do this first:

1. **ROUTER_ACK**
    - sent immediately
    - validates ingress
2. **EXECUTION_ACK**
    - sent once
    - includes SUCCESS / FAILURE

Timeout & retry policy (brief but critical)

A module should:

- Start a timer after sending COMMAND
- Expect:
  - ROUTER_ACK within milliseconds
  - EXECUTION_ACK within TTL
- If ROUTER_ACK missing → router problem
- If EXECUTION_ACK missing → module problem

This gives you **fault isolation**..

Protocol invariant

Every COMMAND must produce at least one ACK.
Every ACK must reference a correlation_id.

That alone gives you:

- traceability
- debuggability
- future replay and audit

Ack Protocol

1. First principle: ACKs are about *state*, not transport

ZeroMQ already guarantees **best-effort transport**.
ACKs should represent **semantic progress**, not "packet arrived".

2. ACKs you should support (minimum viable, scalable)

ACK Types (Canonical)

ACK Type	Sent By	Meaning
ROUTER_ACK	Router	Message accepted into router
DELIVERY_ACK	Router	Message delivered to target module
EXECUTION_ACK	Target Module	Execution result or status
PROGRESS_ACK	Target Module	
TIMEOUT	Sender	Local failure due to timeout
CANCEL	Sender	Abort sequence

- **Important**: The router never reports execution status.
  The module never reports routing status.

ROUTER_ACK — "Message accepted into the bus"

Who sends it: CMB Router
When: Immediately after parsing and validating the message
Meaning: "I received this message, validated it, and placed it onto the channel."

Why it matters:

- Confirms module → CMB delivery
- Detects router down / schema invalid
- Fast (no downstream dependency)

This is your first ACK.

DELIVERY_ACK — "Target module received the message"

Who sends it: Target module (e.g., Behavior)
When: After SUB socket receives and deserializes message
Meaning: "I got the message and it was addressed to me."

Why it matters:

- Confirms PUB/SUB fanout worked
- Detects missing subscribers
- Important for reliability without blocking

EXECUTION_ACK — "Command executed (success or failure)"

Who sends it: Target module
When: After attempting execution
Meaning: "I attempted the command; here's the result."

This ACK **must include status**:

- SUCCESS
- FAILURE
- REJECTED
- DEFERRED

This is the **business-level ACK**.

PROGRESS / HEARTBEAT ACK

Who sends it: Target module
When: Long-running tasks
Meaning: “I’m still working on it.”

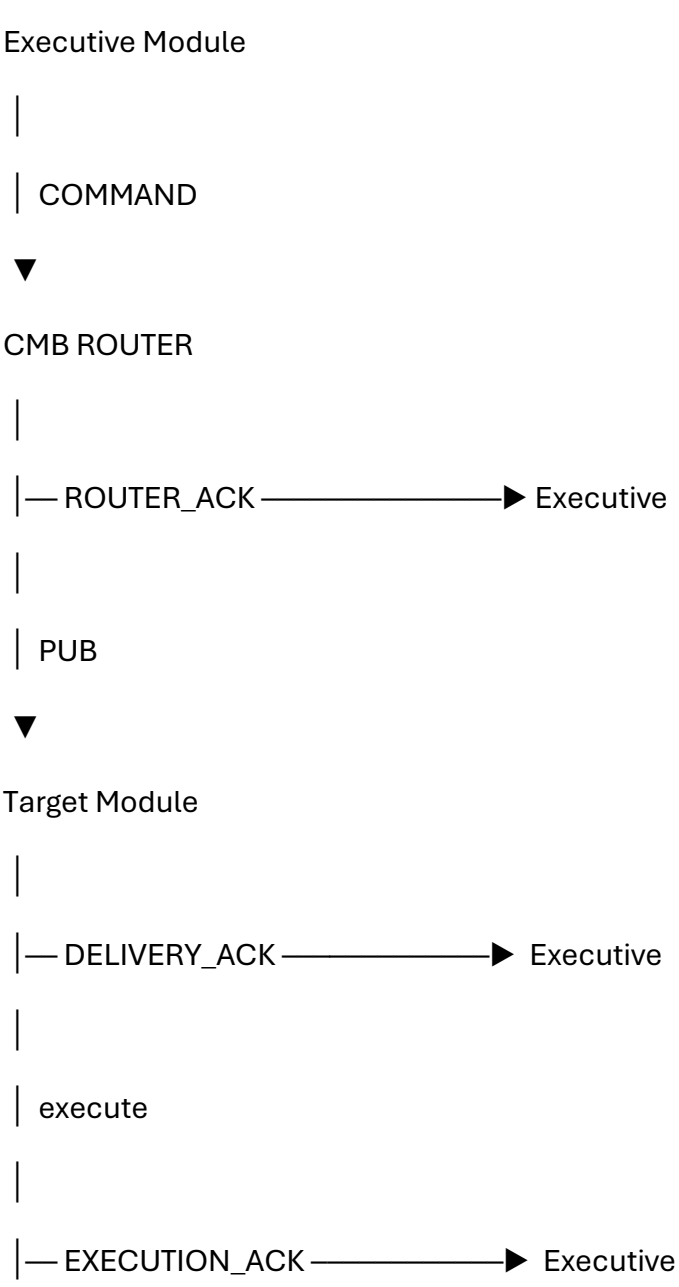
TIMEOUT ACK????

Who sends it: Sender?? Who gets it??
When: time has expired,
Meaning: “I haven’t heard anything back”

CANCEL ACK

Who sends it: Sending module
When: Abort sequence
Meaning: “Stop working on it.”

ACK flow diagram



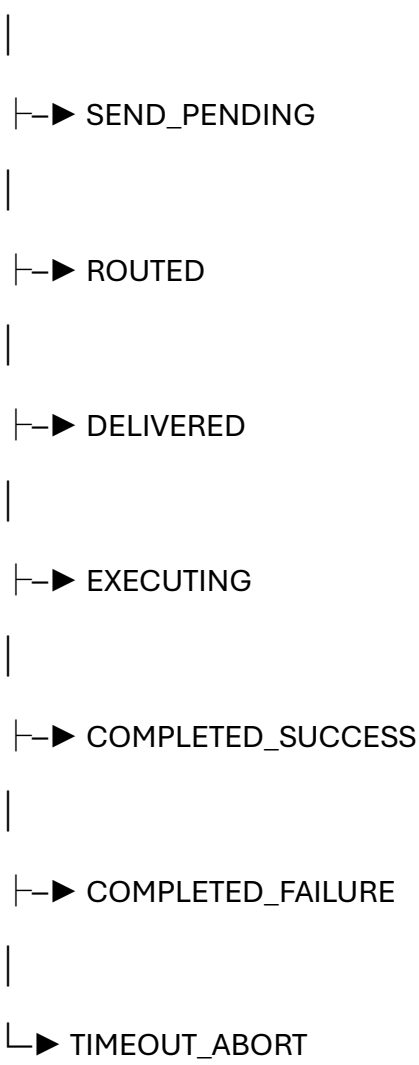
This keeps:

- Transport ACKs fast
- execution ACKs asynchronous
- module responsive

Sender-Side State Machine

State Enumeration

IDLE



State Definitions and Transitions

IDLE

Description

- No active message exchange
- State machine dormant

Entry Condition

- System startup
- Previous exchange completed

Exit Trigger

- Application requests send

SEND_PENDING

Description

- Message sent to router
- Awaiting ROUTER_ACK

Actions

- Send message to router
- Start router_ack_timer

Transitions

Event	Next State
ROUTER_ACK	ROUTED
Timeout	TIMEOUT_ABORT

ROUTED

Description

- Router has accepted the message
- Message is now router-owned

Actions

- Stop router_ack_timer
- Start delivery_ack_timer

Transitions

Event	Next State
DELIVERY_ACK	DELIVERED
Timeout	TIMEOUT_ABORT

DELIVERED

Description

- Router confirms module2 received message
- Execution responsibility now transferred

Actions

- Log delivery confirmation
- Start execution_timer

Transitions

Event	Next State
EXECUTION_ACK(status=in_progress)	EXECUTING
EXECUTION_ACK(status=success)	COMPLETED_SUCCESS
EXECUTION_ACK(status=failure)	COMPLETED_FAILURE
Timeout	TIMEOUT_ABORT

EXECUTING

Description

- Target module acknowledged execution start
- Long-running operation in progress

Actions

- Continue waiting
- Optionally update UI / telemetry

Transitions

Event	Next State
EXECUTION_ACK(status=success)	COMPLETED_SUCCESS
EXECUTION_ACK(status=failure)	COMPLETED_FAILURE
Timeout	TIMEOUT_ABORT
CANCEL	TIMEOUT_ABORT

Important:
in_progress ACKs reset or extend execution timers.

COMPLETED_SUCCESS

Description

- Target module reports success

Actions

- Finalize workflow
- Notify upstream logic
- Persist result if needed

Next State

- IDLE

COMPLETED_FAILURE

Description

- Target module reports failure

Actions

- Log error
- Trigger recovery or retry policy
- Notify UI

Next State

- IDLE (or retry loop if policy allows)

TIMEOUT_ABORT

Description

- Sender-side timeout or cancel

Actions

- Log failure
- Optionally send CANCEL to router/module
- Clean up resources

Next State

- IDLE

Timers (Critical Design Detail)

Each phase has **independent timers**:

Timer	Purpose
router_ack_timer	Router responsiveness
delivery_ack_timer	Routing completion
execution_timer	Module execution

Timers **must not overlap ambiguously**.

This avoids a common bug:
“Execution timed out” when delivery never occurred.

CMB Supporting Modules

The following modules are part of the CMB. They provide functionality for the system associated with communication with the CMB.

Diagnostics

- Dedicated DAC logging module (`cmb_logger`)
- Optional global trace capture
- Awareness modules analyze message patterns

Logging

This module maintains a system of event logging.

- Record major events
- Allow for event tracing
- Allow for root cause
- Record sequence of events
- Additional functions TBD

Awareness

This module addresses the awareness of the system (there maybe 2 levels of awareness; Bus/System).

- Are routers alive
- Are modules alive
- Are modules and routers functioning properly
- What is the state of the system? (Define)
- Are there any performance issues (Define)
- Additional functions TBD

Threat

High priority module that addresses analysis of threats from input data and system performance.

- Does input data present a threat to system performance
- Does data present a threat to system reliability
- Does NLP data present a threat

- Additional functions TBD

Registry

The registry module addresses configuration values. By using the configuration module the user can configure the bus. A separate registry module is necessary for the overall system. This module is GUI based and allows for real time changes.

Tracer

The tracer module is similar to the logger module in its ability to determine the sequence of events. For example, an error in the system would be traced to the sequence of messages leading to the error. This s akin to python's traceback error messages. By tracing the message sequence we could determine if there is a problem with the sequence of messages or the data exchange in the message sequence.

Example Workflow – to be finished

To illustrate how the CMB architecture operates in practice, consider a simple scenario where the **Executive module commands the Behavior module** to perform an action, and the Behavior responds or logs the action. This interaction uses the **Control Channel (CC)** and the **Diagnostic Channel (DAC)**:

1. **Executive Sends a Command:** The Executive decides to trigger a behavior (e.g., *"explore the area"*). It creates a `CognitiveMessage` with `source="executive"` and `targets=["behavior"]` on the Control Channel. The message's payload might be `{"directive": "start_behavior", "behavior": "explore_area"}`. The Executive's ModuleEndpoint sends this message into the CMB via the CC router. (As shown in the code snippet earlier, this is a non-blocking send.) The executive can then continue its own processing or optionally wait for a response.
2. **Control Channel Routing:** The CC router (which was started for the Control Channel) receives the message on its ROUTER socket. It unwraps the frames, reconstructs the `CognitiveMessage`, and then iterates through the `targets` list. For the target `"behavior"`, the router publishes the message on its PUB socket with topic `"behavior"`. Any module subscribed to `"behavior"` on CC will get this message. In our case, the Behavior module is listening on CC for its name. The routing happens almost instantly and in a separate thread, so the Executive isn't blocked. The router logs a debug output like *"Routed message from executive to behavior via CC"*, which helps in tracing the flow (and this could also be captured by a logging module on DAC, if configured).
3. **Behavior Receives the Command:** The Behavior module's endpoint, which is subscribed to topic `"behavior"` on CC, picks up the published message. The `behavior_endpoint.receive()` call unblocks and returns the message to the Behavior's code. The Behavior module inspects the message (sees the source and payload) and recognizes it as a directive from the Executive. It then proceeds to carry out

the requested behavior (e.g., initiating a series of actions to explore the area). For our purposes, the Behavior stub simply prints a log: *"Received message from executive with payload: {directive: 'start_behavior', ...}"*. In a real system, this is where the behavior logic would take over.

4. **Behavior Responds or Logs (Optional):** After acting on the command, the Behavior module might need to send a confirmation or result back. There are multiple ways this could happen in CMB:
    - **Reply on Control Channel:** The Behavior could send a response message with `source="behavior"` and `targets=["executive"]` via the Control Channel, perhaps with payload `{"status": "started", "behavior": "explore_area"}`. The CC router would route this to the Executive (topic `"executive"`), allowing the Executive to receive it as a reply. This would be a simple request-reply over the bus (though not a direct socket reply, it's an asynchronous message reply).
    - **Log to Diagnostic Channel:** Alternatively (or additionally), the Behavior might send a log message to the Diagnostic and Awareness Channel (DAC) to record that it has started the behavior. For example, it could create a message with `source="behavior"`, `targets=["cmb_logger"]` on DAC, with payload `{"event": "Behavior started", "behavior": "explore_area"}`. The `cmb_logger` (a logging module subscribed on DAC) would receive and log this event. In the provided perception module stub, we saw a similar pattern where the Perception module sends status messages to a `cmb_logger` target. Logging via DAC ensures that there is a persistent record of actions and important state changes, which is invaluable for debugging and for the system's self-monitoring.
    - **Trigger Other Channels:** If the Behavior execution leads to other cognitive processes, it might send messages on other channels. For instance, starting a behavior might involve querying memory (sending a question on MC – Memory Channel) or updating the world model (sending data on SMC or VB). Each of those would involve constructing new messages and sending them through the respective channel routers in a similar fashion.
5. **Executive and Others Continue:** The Executive, after sending the command, could carry on with other tasks. If it expects a reply, it would be listening on CC (or whichever channel) for a response targeted to `"executive"`. Other modules in the system remain unaffected by this exchange because they are not subscribed to the "behavior" topic on CC. They might be busy with their own channel communications. For example, a Vision module might be streaming data on PC -> VB, the Memory module might be sending knowledge updates on MC, etc., all in parallel. The channels operate independently, but since modules can have multiple endpoints (one per channel if needed), information can still flow between different parts of the system in a coordinated way via the Executive or specialized mediator modules.

This workflow demonstrates the **publish-subscribe messaging paradigm** in action, coordinated by the CMB. It highlights how the architecture achieves decoupling (Executive doesn't call Behavior directly, they communicate via messages) and flexibility (easy to log, monitor, or extend the interaction). It also shows how **Version 2** improvements (like having a logger on

DAC, using a structured message with TTL/priority) provide a more robust framework for building complex AI behaviors.

Future Directions

In terms of **future directions**, after finalizing Version 3, the next step would be to create a more elaborate demo showcasing multiple channels and modules working together. For example, a scenario could involve a Perception module sending data on the Perception Channel, a Memory module retrieving relevant info on Memory Channel, an Executive making a decision and issuing a command on Control Channel, and a Behavior module acting on it, all coordinated through the CMB. A visual dashboard could subscribe to the Diagnostic channel to display the message flow in real-time. Such a demo would validate the architecture's design and illustrate its capabilities in a tangible way.

Version 3 of the CMB architecture thus provides a solid, well-documented foundation for building complex, modular AI systems. By incorporating structured messaging, multiple topic channels, and clear interfacing patterns, it addresses the shortcomings of the initial version. Modules can now communicate in a flexible yet organized manner, and developers have a clear guide on how to use the infrastructure (thanks to the examples and documentation). As the project moves forward, the CMB can be extended with new features (security, direct queries, load balancing across duplicate modules, etc.) without altering its core design. The current architecture is both **comprehensive and adaptable**, striking a balance that is crucial for the evolving needs of cognitive architectures and AGI research.

Security and Trust (Planned)

- Signature validation
- Channel-level trust rules
- Executive-only authority for certain msg_types