```
"""
NeuroGraph Cognitive Enhancement Suite — Module 1: Stream Parser  v1.1
======================================================================
Changes from v1.0
-----------------
- Thread safety: a shared threading.Lock guards all graph.stimulate() and
  hyperedge completion calls.  The lock is passed in at construction so the
  same lock can be shared with on_message() in openclaw_hook.py, preventing
  race conditions between the daemon worker and the main STDP step.
- Expanded phrase boundary detection: em-dashes, ellipses, opening quotes,
  and parenthetical breaks added to _BOUNDARY_RE.
- max_chunks_per_feed guard: very long messages (>10k chars) no longer
  generate unbounded chunk floods.  Configurable via CESConfig.
- All magic numbers replaced with CESConfig lookups.
- _phrase_chunks() deduplicates near-identical windows within a single feed
  to prevent spammy nudges from sliding-window overlap on long flat text.
"""

from __future__ import annotations

import hashlib
import logging
import math
import queue
import re
import threading
import time
from enum import Enum, auto
from typing import Any, Dict, List, Optional, Set

from ces_config import CESConfig, make_config

logger = logging.getLogger("neurograph.stream_parser")

# Sentinel for queue shutdown
_SHUTDOWN = object()


# ---------------------------------------------------------------------------
# Source enum
# ---------------------------------------------------------------------------

class StreamSource(Enum):
    USER = auto()   # external — full weight (1.0)
    SYL  = auto()   # self-generated — efference copy weight (oscillating)


# ---------------------------------------------------------------------------
# Phrase boundary detector  (v1.1: extended)
# ---------------------------------------------------------------------------

_BOUNDARY_RE = re.compile(
    r'(?<=[.!?])\s+'                      # sentence end
    r'|(?<=\.\.\.)\s+'                    # ellipsis
    r'|(?<=—)\s*'                         # em-dash
    r'|(?<=,)\s+'                         # comma pause
    r'|(?<=[;:])\s+'                      # semicolon / colon
    r'|(?<=["\u201c\u201d])\s+'           # closing quote (straight + curly)
    r'|(?<=\))\s+'                        # closing parenthesis
    r'|\s+(?:and|but|or|so|yet|because|although|however|therefore)\s+',
    re.IGNORECASE,
)

# Short fingerprint for near-duplicate detection
def _chunk_fp(text: str) -> str:
    return hashlib.md5(text.lower().split().__str__().encode()).hexdigest()[:8]


def _phrase_chunks(text: str, cfg: CESConfig) -> List[str]:
    """
    Split *text* into phrase-boundary-aware chunks within configured limits.
    Deduplicates near-identical windows to prevent nudge spam on long
    flat prose.  Respects max_chunks_per_feed.
    """
    min_c: int = cfg["sp_min_chunk_chars"]
    max_c: int = cfg["sp_max_chunk_chars"]
    advance: int = cfg["sp_window_advance_chars"]
    max_total: int = cfg["sp_max_chunks_per_feed"]

    parts: List[str] = [p.strip() for p in _BOUNDARY_RE.split(text) if p.strip()]
    chunks: List[str] = []
    seen_fps: Set[str] = set()
    buffer = ""

    def _emit(s: str) -> bool:
        """Add s to chunks if unique and within limit. Returns False if capped.""
        nonlocal chunks
        if len(chunks) >= max_total:
            return False
        fp = _chunk_fp(s)
        if fp in seen_fps:
            return True
        seen_fps.add(fp)
        chunks.append(s)
        return True

    for part in parts:
        candidate = (buffer + " " + part).strip() if buffer else part

        if len(candidate) >= min_c and len(candidate) <= max_c:
            if not _emit(candidate):
                break
            buffer = ""
        elif len(candidate) > max_c:
            if buffer and len(buffer) >= min_c:
                if not _emit(buffer):
                    break
            for start in range(0, len(part), advance):
                window = part[start:start + max_c].strip()
                if len(window) >= min_c:
                    if not _emit(window):
                        break
            buffer = ""
        else:
            buffer = candidate

    if buffer and len(buffer) >= min_c and len(chunks) < max_total:
        _emit(buffer)

    return chunks


# ---------------------------------------------------------------------------
# Ollama embedding client
# ---------------------------------------------------------------------------

class _OllamaEmbedder:
    def __init__(self, base_url: str, model: str, timeout: int) -> None:
        self._url = f"{base_url}/api/embeddings"
        self._model = model
        self._timeout = timeout

    def embed(self, text: str) -> Optional[List[float]]:
        import json, urllib.request, urllib.error
        payload = json.dumps({"model": self._model, "prompt": text}).encode()
        req = urllib.request.Request(
            self._url, data=payload,
            headers={"Content-Type": "application/json"},
        )
        try:
            with urllib.request.urlopen(req, timeout=self._timeout) as resp:
                return json.loads(resp.read()).get("embedding")
        except (urllib.error.URLError, OSError, Exception) as exc:
            logger.debug("Ollama embed failed: %s", exc)
            return None


def _ollama_available(base_url: str, timeout: int = 2) -> bool:
    import urllib.request, urllib.error
    try:
        urllib.request.urlopen(f"{base_url}/api/tags", timeout=timeout)
        return True
    except (urllib.error.URLError, OSError):
        return False


# ---------------------------------------------------------------------------
# Null fallback
# ---------------------------------------------------------------------------

class NullStreamParser:
    """Returned by StreamParser.create() when Ollama is unreachable."""

    def feed(self, text: str, source: StreamSource) -> None:
        pass

    def shutdown(self) -> None:
        pass

    @property
    def is_active(self) -> bool:
        return False

    def stats(self) -> Dict[str, Any]:
        return {"active": False, "reason": "ollama_unavailable"}


# ---------------------------------------------------------------------------
# Stream Parser
# ---------------------------------------------------------------------------

class StreamParser:
    """
    Background stream processor.
```

```
Parameters
----------
graph       neuro_foundation.Graph
vector_db   SimpleVectorDB
graph_lock  threading.Lock shared with openclaw_hook's main thread.
            MUST be the same lock used around graph.step() calls.
            This is the v1.1 thread-safety fix.
cfg         CESConfig (optional; uses defaults if omitted)
embedder    injectable for testing
"""

def __init__(
    self,
    graph: Any,
    vector_db: Any,
    graph_lock: threading.Lock,
    cfg: Optional[CESConfig] = None,
    embedder: Optional[_OllamaEmbedder] = None,
) -> None:
    self._graph = graph
    self._vector_db = vector_db
    self._lock = graph_lock
    self._cfg = cfg or make_config()
    self._embedder = embedder or _OllamaEmbedder(
        base_url=self._cfg["sp_ollama_base_url"],
        model=self._cfg["sp_ollama_embed_model"],
        timeout=self._cfg["sp_ollama_timeout"],
    )

    self._chunk_counter: int = 0
    self._queue: queue.Queue = queue.Queue(
        maxsize=self._cfg["sp_queue_maxsize"]
    )
    self._active = True
    self._thread = threading.Thread(
        target=self._worker, daemon=True, name="ng-stream-parser"
    )
    self._thread.start()

    # Telemetry
    self._chunks_processed: int = 0
    self._chunks_skipped: int = 0
    self._nudges_applied: int = 0

    logger.info(
        "StreamParser v1.2 initialised (model=%s)",
        self._cfg["sp_ollama_embed_model"],
    )

# ------------------------------------------------------------------
# Factory
# ------------------------------------------------------------------

@classmethod
def create(
    cls,
    graph: Any,
    vector_db: Any,
    graph_lock: threading.Lock,
    cfg: Optional[CESConfig] = None,
) -> "StreamParser | NullStreamParser":
    """
    Safe factory.  Returns NullStreamParser if Ollama is unreachable.
    graph_lock MUST be the same lock used in openclaw_hook.on_message().
    """
    c = cfg or make_config()
    if not _ollama_available(c["sp_ollama_base_url"]):
        logger.warning(
            "Ollama not reachable at %s — StreamParser disabled.",
            c["sp_ollama_base_url"],
        )
        return NullStreamParser()
    return cls(graph, vector_db, graph_lock, cfg=c)

# ------------------------------------------------------------------
# Public API
# ------------------------------------------------------------------

def feed(self, text: str, source: StreamSource) -> None:
    """Queue text for background processing. Non-blocking."""
    if not text or not self._active:
        return
    try:
        self._queue.put_nowait((text, source))
    except queue.Full:
        logger.debug("StreamParser queue full — chunk dropped")
        self._chunks_skipped += 1

def shutdown(self) -> None:
    self._active = False
    self._queue.put(_SHUTDOWN)
    self._thread.join(timeout=5)
    logger.info(
        "StreamParser shutdown: processed=%d skipped=%d nudges=%d",
        self._chunks_processed, self._chunks_skipped, self._nudges_applied,
    )

def reset(self) -> None:
    """Clear internal state. Call if the graph is reset mid-session."""
    self._chunk_counter = 0
    # Drain the queue without processing
    while not self._queue.empty():
        try:
            self._queue.get_nowait()
        except queue.Empty:
            break
    logger.info("StreamParser reset")

@property
def is_active(self) -> bool:
    return self._active and self._thread.is_alive()

def stats(self) -> Dict[str, Any]:
    return {
        "active": self.is_active,
        "chunks_processed": self._chunks_processed,
        "chunks_skipped": self._chunks_skipped,
        "nudges_applied": self._nudges_applied,
        "current_efference_weight": round(self._efference_weight(), 4),
    }

# ------------------------------------------------------------------
# Internal
# ------------------------------------------------------------------

def _efference_weight(self) -> float:
    cfg = self._cfg
    phase = (
        (self._chunk_counter % cfg["sp_efference_period_steps"])
        / cfg["sp_efference_period_steps"]
    )
    sine = math.sin(2 * math.pi * phase)
    mid = (cfg["sp_efference_min"] + cfg["sp_efference_max"]) / 2
    amp = (cfg["sp_efference_max"] - cfg["sp_efference_min"]) / 2
    return mid + amp * sine

def _worker(self) -> None:
    while True:
        item = self._queue.get()
        if item is _SHUTDOWN:
            break
        text, source = item
        # Broad guard: a single malformed chunk must never kill the daemon.
        try:
            self._process(text, source)
        except Exception as exc:
            logger.warning(
                "StreamParser worker: unhandled exception in _process "
                "(chunk dropped): %s", exc, exc_info=True,
            )
        self._queue.task_done()

def _process(self, text: str, source: StreamSource) -> None:
    weight = (
        1.0 if source == StreamSource.USER
        else self._efference_weight()
    )
    cfg = self._cfg

    for chunk in _phrase_chunks(text, cfg):
        self._chunk_counter += 1
        embedding = self._embedder.embed(chunk)
        if embedding is None:
            self._chunks_skipped += 1
            continue

        try:
            neighbours = self._vector_db.similarity_search(
                embedding, k=cfg["sp_neighbour_k"]
            )
        except Exception as exc:
            logger.debug("Vector search failed: %s", exc)
            self._chunks_processed += 1
            continue

        default_threshold = self._graph.config.get("default_threshold", 1.0)

        # --- Lock acquired for all graph mutations in this chunk ---
        with self._lock:
            for neighbour in neighbours:
                similarity = neighbour.get(
                    "similarity", neighbour.get("score", 0)
                )
                if similarity < cfg["sp_neighbour_threshold"]:
                    continue
                node_id = neighbour.get("node_id")
                if not node_id or node_id not in self._graph.nodes:
                    continue
                nudge = (
                    default_threshold
                    * cfg["sp_neighbour_nudge"]
                    * weight
                    * similarity
                )
                try:
                    self._graph.stimulate(node_id, nudge)
                    self._nudges_applied += 1
                except Exception as exc:
                    logger.debug("Stimulate failed for %s: %s", node_id, exc)

            self._check_hyperedge_completion(embedding, weight, default_thres
        # --- Lock released ---

        self._chunks_processed += 1

def _check_hyperedge_completion(
    self,
    embedding: List[float],
    weight: float,
    default_threshold: float,
) -> None:
    """
    Called inside the graph lock.  Primes inactive hyperedge members
    when a partial pattern is detected.
    """
    cfg = self._cfg
    min_ratio: float = cfg["sp_he_completion_min_ratio"]
    max_ratio: float = cfg["sp_he_completion_max_ratio"]

    try:
        for he in self._graph.hyperedges.values():
            if he.refractory_remaining > 0:
                continue

            active_members = [
                nid for nid in he.member_node_ids
                if nid in self._graph.nodes
                and self._graph.nodes[nid].voltage > default_threshold * 0.4
            ]

            if not active_members:
                continue

            ratio = len(active_members) / max(len(he.member_node_ids), 1)
            if not (min_ratio <= ratio < max_ratio):
```

```
            continue

        inactive = [
            nid for nid in he.member_node_ids
            if nid not in active_members
            and nid in self._graph.nodes
        ]
        nudge = (
            self._graph.config.get("he_pattern_completion_strength", 0.3)
            * weight
            * ratio
        )
        for nid in inactive:
            try:
                self._graph.stimulate(nid, nudge)
                self._nudges_applied += 1
            except Exception:
                pass
except Exception as exc:
    logger.debug("Hyperedge completion check failed: %s", exc)
```