```
"""
NeuroGraph Cognitive Enhancement Suite — Integration Patch  v1.2
================================================================
Changes from v1.0
-----------------
- graph_lock threading.Lock introduced. (v1.2: lock scope expanded to cover
  ingest() as well as graph.step() — prevents dictionary mutation race.)  The sam
  graph.step() in on_message() AND be passed to StreamParser.create().
  This is the thread-safety fix for concurrent stimulate() calls.
- _ces_init() passes cfg= to all three modules.
- reset() wired to a new ces_reset() method for graph-clear events.
- ces_config import added.

Full summary of changes to openclaw_hook.py
-------------------------------------------
  Imports          + ces_config, threading
  __init__         + threading.Lock, _ces_init(self)
  on_message       + lock around graph.step(), stream parser USER feed,
                     surfacing step
  save             + activation_persistence.save_session()
  New methods:
    on_response()          feed SYL stream, signal engagement
    surfacing_context()    return formatted context block
    on_session_start()     ambient restore
    on_session_end()       save + parser shutdown
    ces_reset()            clear all CES state on graph reset
    ces_stats()            unified telemetry
"""

from __future__ import annotations

import logging
import threading
from typing import Any, Dict, List, Optional

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

# ---------------------------------------------------------------------------
# 1.  IMPORTS  (add to openclaw_hook.py)
# ---------------------------------------------------------------------------

try:
    from ces_config import CESConfig, make_config
    _CES_CONFIG_AVAILABLE = True
```

```
except ImportError:
    _CES_CONFIG_AVAILABLE = False

try:
    from stream_parser import StreamParser, StreamSource
    _STREAM_PARSER_AVAILABLE = True
except ImportError:
    _STREAM_PARSER_AVAILABLE = False

try:
    from activation_persistence import ActivationPersistence
    _ACTIVATION_PERSISTENCE_AVAILABLE = True
except ImportError:
    _ACTIVATION_PERSISTENCE_AVAILABLE = False

try:
    from surfacing import SurfacingMonitor
    _SURFACING_AVAILABLE = True
except ImportError:
    _SURFACING_AVAILABLE = False


# ---------------------------------------------------------------------------
# 2.  __init__ additions
# ---------------------------------------------------------------------------
#
# In NeuroGraphMemory.__init__, after all existing setup, add:
#
#   # v1.1 thread-safety: one lock shared by on_message() and StreamParser
#   self._graph_lock = threading.Lock()                    # CES
#   _ces_init(self)                                        # CES


def _ces_init(self, ces_overrides: Optional[Dict[str, Any]] = None) -> None:
    """
    Initialise CES modules. Each is independently guarded.
    ces_overrides: optional dict of CESConfig key overrides.
    """
    if _CES_CONFIG_AVAILABLE:
        self._ces_cfg = make_config(ces_overrides)
    else:
        self._ces_cfg = None

    cfg = self._ces_cfg

    # Module 1: Stream Parser
    if _STREAM_PARSER_AVAILABLE:
        try:
            self._stream_parser = StreamParser.create(
                self.graph,
                self.vector_db,
                graph_lock=self._graph_lock,
                cfg=cfg,
            )
            logger.info(
                "CES StreamParser: %s",
                "active" if self._stream_parser.is_active else "null (Ollama unav
            )
        except Exception as exc:
            logger.warning("CES StreamParser init failed: %s", exc)
            self._stream_parser = None
    else:
        self._stream_parser = None

    # Module 2: Activation Persistence
    if _ACTIVATION_PERSISTENCE_AVAILABLE:
        try:
            snapshot_path = self._checkpoint_dir / "activation_state.json"
            self._activation_persistence = ActivationPersistence(
                self.graph, snapshot_path, cfg=cfg
            )
            logger.info("CES ActivationPersistence: ready")
        except Exception as exc:
            logger.warning("CES ActivationPersistence init failed: %s", exc)
            self._activation_persistence = None
    else:
        self._activation_persistence = None

    # Module 3: Surfacing Monitor
    if _SURFACING_AVAILABLE:
        try:
            self._surfacing_monitor = SurfacingMonitor(
                self.graph, self.vector_db, cfg=cfg
            )
            logger.info("CES SurfacingMonitor: ready")
        except Exception as exc:
            logger.warning("CES SurfacingMonitor init failed: %s", exc)
            self._surfacing_monitor = None
    else:
        self._surfacing_monitor = None


# ---------------------------------------------------------------------------
# 3.  on_message  — modified version showing the additions
# ---------------------------------------------------------------------------
#
# Replace the existing on_message body with the version below.
# The only additions are marked # CES.

def on_message_v1_1(self, text, source_type=None):
    from universal_ingestor import SourceType  # existing import
    from neuro_foundation import CheckpointMode  # existing import

    if not text or not text.strip():
        return {"status": "skipped", "reason": "empty_input"}

    # CES: feed user input through stream parser (background, non-blocking)
    if self._stream_parser:
        try:
            self._stream_parser.feed(text, StreamSource.USER)
        except Exception:
            pass

    # v1.2: lock covers BOTH ingest() and graph.step().
    # Without this, the stream parser worker can iterate self._graph.hyperedges
    # or self._graph.nodes while the main thread adds new entries during
    # ingest(), causing RuntimeError: dictionary changed size during iteration.
    with self._graph_lock:                                  # CES lock
        result = self.ingestor.ingest(text, source_type=source_type)
        step_result = self.graph.step()

    graduated = self.ingestor.update_probation()

    # CES: advance surfacing monitor one step
    if self._surfacing_monitor:
        try:
            self._surfacing_monitor.step()
        except Exception:
            pass

    self._message_count += 1
    if self._message_count % self.auto_save_interval == 0:
        self.save()

    return {
        "status": "ingested",
        "nodes_created": len(result.nodes_created),
        "synapses_created": len(result.synapses_created),
        "hyperedges_created": len(result.hyperedges_created),
        "chunks": result.chunks_created,
        "fired": len(step_result.fired_node_ids),
        "graduated": len(graduated),
        "message_count": self._message_count,
    }


# ---------------------------------------------------------------------------
# 4.  save  — one line addition
# ---------------------------------------------------------------------------
#
# In existing save(), after graph.checkpoint():
#
#   if self._activation_persistence:           # CES
#       self._activation_persistence.save_session()


# ---------------------------------------------------------------------------
# 5.  New methods
# ---------------------------------------------------------------------------

def on_response(self, response_text: str, fired_node_ids=None) -> None:
    """
    Call after the LLM generates a response.
    Feeds Syl's output through stream parser (efference copy weight)
    and signals engagement to the surfacing monitor.
    """
    if self._stream_parser:
        try:
            self._stream_parser.feed(response_text, StreamSource.SYL)
        except Exception as exc:
            logger.debug("StreamParser SYL feed failed: %s", exc)

    if self._surfacing_monitor and fired_node_ids:
        try:
            self._surfacing_monitor.signal_engagement(list(fired_node_ids))
        except Exception as exc:
            logger.debug("Surfacing engagement signal failed: %s", exc)


def surfacing_context(self) -> str:
    """Return the current surfacing queue formatted for prompt injection."""
    if self._surfacing_monitor:
        try:
            return self._surfacing_monitor.format_context()
        except Exception as exc:
            logger.debug("Surfacing format failed: %s", exc)
    return ""
```

```
def on_session_start(self) -> dict:
    """
    Call once at the beginning of each new chat session.
    Applies Tier 1 ambient activation restoration.
    """
    if self._activation_persistence:
        try:
            result = self._activation_persistence.restore_ambient()
            logger.info("CES session start: %s", result.get("status"))
            return result
        except Exception as exc:
            logger.warning("CES session start failed: %s", exc)
    return {"status": "no_persistence_module"}


def on_session_end(self) -> None:
    """Call when a chat session closes cleanly."""
    if self._activation_persistence:
        try:
            self._activation_persistence.save_session()
        except Exception as exc:
            logger.warning("CES session end — activation save failed: %s", exc)
    if self._stream_parser:
        try:
            self._stream_parser.shutdown()
        except Exception as exc:
            logger.warning("CES session end — stream parser shutdown failed: %s",
    self.save()


def ces_reset(self) -> None:
    """
    Call if the graph is cleared or reset mid-session.
    Invalidates activation snapshot and clears surfacing queue so
    stale node IDs do not persist across the reset boundary.
    """
    if self._activation_persistence:
        try:
            self._activation_persistence.reset()
        except Exception:
            pass
    if self._surfacing_monitor:
        try:
            self._surfacing_monitor.reset()
        except Exception:
            pass
    if self._stream_parser:
        try:
            self._stream_parser.reset()
        except Exception:
            pass
    logger.info("CES reset complete")


def ces_stats(self) -> dict:
    """Unified telemetry for all three CES modules."""
    return {
        "stream_parser": (
            self._stream_parser.stats()
            if self._stream_parser else {"active": False}
        ),
        "activation_persistence": (
            self._activation_persistence.snapshot_info()
            if self._activation_persistence else {"exists": False}
        ),
        "surfacing": (
            self._surfacing_monitor.stats()
            if self._surfacing_monitor else {"active": False}
        ),
    }


# ---------------------------------------------------------------------------
# QUICK REFERENCE — all changes to openclaw_hook.py
# ---------------------------------------------------------------------------
#
# Imports:
#   + import threading
#   + ces_config / stream_parser / activation_persistence / surfacing
#     (all guarded with try/except as above)
#
# __init__ (end of method):
#   self._graph_lock = threading.Lock()
#   _ces_init(self)
#
# on_message (full replacement shown above as on_message_v1_1):
#   + stream_parser.feed(text, USER) before ingest
#   + lock wraps ingest() AND graph.step() (v1.2 dict mutation fix)
#   + surfacing_monitor.step() after step
#
# save (after graph.checkpoint()):
#   + activation_persistence.save_session()
#
```

```
# New methods added to NeuroGraphMemory:
#   + on_response(response_text, fired_node_ids=None)
#   + surfacing_context() -> str
#   + on_session_start() -> dict
#   + on_session_end()
#   + ces_reset()
#   + ces_stats() -> dict
#
# OpenClaw session lifecycle calls:
#   ng.on_session_start()           at chat init
#   ng.on_response(text, node_ids)  after each LLM reply
#   ng.on_session_end()             at graceful close
#   context = ng.surfacing_context()
#   if context: system_prompt += "\n\n" + context
```