```
import random
from threading import Lock
from time import sleep
from typing import Callable, List, Optional

from swarms.structs.agent import Agent
from swarms.structs.base_swarm import BaseSwarm
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="swarm_load_balancer")


class AgentLoadBalancer(BaseSwarm):
    """
    A load balancer class that distributes tasks among a group of agents.

    Args:
        agents (List[Agent]): The list of agents available for task execution.
        max_retries (int, optional): The maximum number of retries for a task if it fails. Defaults to 3.
        max_loops (int, optional): The maximum number of loops to run a task. Defaults to 5.
        cooldown_time (float, optional): The cooldown time between retries. Defaults to 0.

    Attributes:
        agents (List[Agent]): The list of agents available for task execution.
        agent_status (Dict[str, bool]): The status of each agent, indicating whether it is available or not.
        max_retries (int): The maximum number of retries for a task if it fails.
```

```
    max_loops (int): The maximum number of loops to run a task.
    agent_performance (Dict[str, Dict[str, int]]): The performance statistics of each agent.
    lock (Lock): A lock to ensure thread safety.
    cooldown_time (float): The cooldown time between retries.

Methods:
    get_available_agent: Get an available agent for task execution.
    set_agent_status: Set the status of an agent.
    update_performance: Update the performance statistics of an agent.
    log_performance: Log the performance statistics of all agents.
    run_task: Run a single task using an available agent.
    run_multiple_tasks: Run multiple tasks using available agents.
    run_task_with_loops: Run a task multiple times using an available agent.
    run_task_with_callback: Run a task with a callback function.
    run_task_with_timeout: Run a task with a timeout.

"""

def __init__(
    self,
    agents: List[Agent],
    max_retries: int = 3,
    max_loops: int = 5,
    cooldown_time: float = 0,
):
    self.agents = agents
```

```
    self.agent_status = {
        agent.agent_name: True for agent in agents
    }
    self.max_retries = max_retries
    self.max_loops = max_loops
    self.agent_performance = {
        agent.agent_name: {"success_count": 0, "failure_count": 0}
        for agent in agents
    }
    self.lock = Lock()
    self.cooldown_time = cooldown_time
    self.swarm_initialization()

def swarm_initialization(self):
    logger.info(
        "Initializing AgentLoadBalancer with the following agents:"
    )

    # Make sure all the agents exist
    assert self.agents, "No agents provided to the Load Balancer"

    # Assert that all agents are of type Agent
    for agent in self.agents:
        assert isinstance(
            agent, Agent
        ), "All agents should be of type Agent"
```

```
    for agent in self.agents:
        logger.info(f"Agent Name: {agent.agent_name}")

    logger.info("Load Balancer Initialized Successfully!")

def get_available_agent(self) -> Optional[Agent]:
    """
    Get an available agent for task execution.

    Returns:
        Optional[Agent]: An available agent, or None if no agents are available.

    """
    with self.lock:
        available_agents = [
            agent
            for agent in self.agents
            if self.agent_status[agent.agent_name]
        ]
        logger.info(
            f"Available agents: {[agent.agent_name for agent in available_agents]}"
        )
        if not available_agents:
            return None
        return random.choice(available_agents)
```

```
def set_agent_status(self, agent: Agent, status: bool) -> None:
    """
    Set the status of an agent.

    Args:
        agent (Agent): The agent whose status needs to be set.
        status (bool): The status to set for the agent.

    """
    with self.lock:
        self.agent_status[agent.agent_name] = status

def update_performance(self, agent: Agent, success: bool) -> None:
    """
    Update the performance statistics of an agent.

    Args:
        agent (Agent): The agent whose performance statistics need to be updated.
        success (bool): Whether the task executed by the agent was successful or not.

    """
    with self.lock:
        if success:
            self.agent_performance[agent.agent_name][
                "success_count"
```

```
            ] += 1
        else:
            self.agent_performance[agent.agent_name][
                "failure_count"
            ] += 1

def log_performance(self) -> None:
    """
    Log the performance statistics of all agents.

    """
    logger.info("Agent Performance:")
    for agent_name, stats in self.agent_performance.items():
        logger.info(f"{agent_name}: {stats}")

def run(self, task: str, *args, **kwargs) -> str:
    """
    Run a single task using an available agent.

    Args:
        task (str): The task to be executed.

    Returns:
        str: The output of the task execution.

    Raises:
```

```
    RuntimeError: If no available agents are found to handle the request.


"""
try:
    retries = 0
    while retries < self.max_retries:
        agent = self.get_available_agent()
        if not agent:
            raise RuntimeError(
                "No available agents to handle the request."
            )


        try:
            self.set_agent_status(agent, False)
            output = agent.run(task, *args, **kwargs)
            self.update_performance(agent, True)
            return output
        except Exception as e:
            logger.error(
                f"Error with agent {agent.agent_name}: {e}"
            )
            self.update_performance(agent, False)
            retries += 1
            sleep(self.cooldown_time)
            if retries >= self.max_retries:
                raise e
```

```
            finally:
                self.set_agent_status(agent, True)
        except Exception as e:
            logger.error(
                f"Task failed: {e} try again by optimizing the code."
            )
            raise RuntimeError(f"Task failed: {e}")

    def run_multiple_tasks(self, tasks: List[str]) -> List[str]:
        """
        Run multiple tasks using available agents.

        Args:
            tasks (List[str]): The list of tasks to be executed.

        Returns:
            List[str]: The list of outputs corresponding to each task execution.

        """
        results = []
        for task in tasks:
            result = self.run(task)
            results.append(result)
        return results

    def run_task_with_loops(self, task: str) -> List[str]:
```

```
    """
    Run a task multiple times using an available agent.

    Args:
        task (str): The task to be executed.

    Returns:
        List[str]: The list of outputs corresponding to each task execution.

    """
    results = []
    for _ in range(self.max_loops):
        result = self.run(task)
        results.append(result)
    return results

def run_task_with_callback(
    self, task: str, callback: Callable[[str], None]
) -> None:
    """
    Run a task with a callback function.

    Args:
        task (str): The task to be executed.
        callback (Callable[[str], None]): The callback function to be called with the task result.
```

```
    """
    try:
        result = self.run(task)
        callback(result)
    except Exception as e:
        logger.error(f"Task failed: {e}")
        callback(str(e))

def run_task_with_timeout(self, task: str, timeout: float) -> str:
    """
    Run a task with a timeout.

    Args:
        task (str): The task to be executed.
        timeout (float): The maximum time (in seconds) to wait for the task to complete.

    Returns:
        str: The output of the task execution.

    Raises:
        TimeoutError: If the task execution exceeds the specified timeout.
        Exception: If the task execution raises an exception.

    """
    import threading
```

```
        result = [None]
        exception = [None]

        def target():
            try:
                result[0] = self.run(task)
            except Exception as e:
                exception[0] = e

        thread = threading.Thread(target=target)
        thread.start()
        thread.join(timeout)

        if thread.is_alive():
            raise TimeoutError(
                f"Task timed out after {timeout} seconds."
            )

        if exception[0]:
            raise exception[0]

        return result[0]


# if __name__ == "__main__":
#     from swarms import llama3Hosted()
```

```
#     # User initializes the agents
#     agents = [
#         Agent(
#             agent_name="Transcript Generator 1",
#             agent_description="Generate a transcript for a youtube video on what swarms are!",
#             llm=llama3Hosted(),
#             max_loops="auto",
#             autosave=True,
#             dashboard=False,
#             streaming_on=True,
#             verbose=True,
#             stopping_token="<DONE>",
#             interactive=True,
#             state_save_file_type="json",
#             saved_state_path="transcript_generator_1.json",
#         ),
#         Agent(
#             agent_name="Transcript Generator 2",
#             agent_description="Generate a transcript for a youtube video on what swarms are!",
#             llm=llama3Hosted(),
#             max_loops="auto",
#             autosave=True,
#             dashboard=False,
#             streaming_on=True,
#             verbose=True,
#             stopping_token="<DONE>",
```

```
#             interactive=True,
#             state_save_file_type="json",
#             saved_state_path="transcript_generator_2.json",
#         )
#         # Add more agents as needed
#     ]

#     load_balancer = LoadBalancer(agents)

#     try:
#          result = load_balancer.run_task("Generate a transcript for a youtube video on what swarms are!")
#         print(result)

#         # Running multiple tasks
#         tasks = [
#             "Generate a transcript for a youtube video on what swarms are!",
#             "Generate a transcript for a youtube video on AI advancements!"
#         ]
#         results = load_balancer.run_multiple_tasks(tasks)
#         for res in results:
#             print(res)

#         # Running task with loops
#          loop_results = load_balancer.run_task_with_loops("Generate a transcript for a youtube video on what swarms are!")
```

```
#         for res in loop_results:
#             print(res)

#     except RuntimeError as e:
#         print(f"Error: {e}")

#     # Log performance
#     load_balancer.log_performance()
```