Advanced Python Load Balancer with AsyncIO, Caching, and Design Patterns for Cloud-Native Distributed Systems

This script demonstrates advanced Python programming concepts that are highly relevant in modern distributed systems, cloud-native applications, and technical interview scenarios at FAANG/MAANG companies.

Key Concepts & High-Ranking Keywords Covered:

---

1. **Asynchronous Programming (asyncio, await, concurrency, coroutines)**
2. **Load Balancing Algorithms (Round Robin, Least Connections, Random)**
3. **Caching Strategies (in-memory LRU cache using functools.lru_cache)**
4. **Design Patterns (Singleton, Strategy, Factory Method)**
5. **Thread-Safe Operations with asyncio.Lock**
6. **Scalability, High Availability, and Fault Tolerance Principles**
7. **Advanced Error Handling with Custom Exceptions**
8. **Structured Logging for Observability and Monitoring**
9. **Cloud-Native Keywords (microservices, distributed systems, service registry)**
10. **Python Best Practices: Type Hints, Docstrings, Context Managers, Modularization**

"""

import asyncio

import random

```
import logging
from functools import lru_cache
from typing import Callable, Dict, List


# ---------------------------------------------------------------------------
# Logging Configuration (Observability, Monitoring, Debugging)
# ---------------------------------------------------------------------------
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(name)s | %(message)s",
    handlers=[logging.StreamHandler()]
)

logger = logging.getLogger("LoadBalancerSystem")


# ---------------------------------------------------------------------------
# Custom Exceptions for Robust Error Handling
# ---------------------------------------------------------------------------
class ServerOverloadedError(Exception):
    """Raised when a server exceeds its maximum capacity (simulating overload)."""
    pass
```

```
class NoHealthyServersError(Exception):
    """Raised when no servers are available or healthy in the pool."""
    pass



# ---------------------------------------------------------------------------
# Singleton Pattern for Service Registry
# ---------------------------------------------------------------------------
class ServiceRegistry:
    """
    Singleton that stores available servers.
    In real cloud-native systems, this would be a distributed key-value store
    like etcd, Consul, or ZooKeeper.
    """
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(ServiceRegistry, cls).__new__(cls)
            cls._instance.servers: List["Server"] = []
        return cls._instance

    def register(self, server: "Server") -> None:
        """Register a new server to the service registry."""
        self.servers.append(server)
```

```
        logger.info(f"Server {server.name} registered with capacity {server.capacity}")

    def healthy_servers(self) -> List["Server"]:
        """Return only healthy servers (not overloaded)."""
        return [s for s in self.servers if s.is_healthy()]


# ---------------------------------------------------------------------------
# Server Class (Simulating Microservices / Backend Nodes)
# ---------------------------------------------------------------------------
class Server:
    """
    Represents a backend server node in a distributed microservices cluster.
    """
    def __init__(self, name: str, capacity: int):
        self.name = name
        self.capacity = capacity  # Max concurrent requests
        self.current_load = 0     # Active requests counter
        self.lock = asyncio.Lock()  # Concurrency safety

    async def handle_request(self, request_id: int) -> str:
        """
        Simulate handling a request asynchronously.
        Uses asyncio to mimic I/O-bound workload (e.g., DB query, API call).
        """
```

```
        async with self.lock:
            if self.current_load >= self.capacity:
                raise ServerOverloadedError(f"{self.name} is overloaded!")

            self.current_load += 1
            logger.info(f"Server {self.name} handling request {request_id}. Load: {self.current_load}/{self.capacity}")

        # Simulated processing delay
        await asyncio.sleep(random.uniform(0.2, 0.6))

        async with self.lock:
            self.current_load -= 1
            logger.info(f"Server {self.name} completed request {request_id}. Load: {self.current_load}/{self.capacity}")

        return f"Response from {self.name} for request {request_id}"

    def is_healthy(self) -> bool:
        """A server is healthy if not at full capacity."""
        return self.current_load < self.capacity


# ---------------------------------------------------------------------------
# Strategy Pattern: Load Balancing Algorithms
# ---------------------------------------------------------------------------
```

```
class LoadBalancingStrategy:
    """Abstract Strategy for load balancing."""
    def select_server(self, servers: List[Server]) -> Server:
        raise NotImplementedError



class RoundRobinStrategy(LoadBalancingStrategy):
    """Classic Round Robin algorithm."""
    def __init__(self):
        self.index = 0

    def select_server(self, servers: List[Server]) -> Server:
        server = servers[self.index % len(servers)]
        self.index += 1
        return server



class LeastConnectionsStrategy(LoadBalancingStrategy):
    """Selects the server with the least active connections."""
    def select_server(self, servers: List[Server]) -> Server:
        return min(servers, key=lambda s: s.current_load)



class RandomChoiceStrategy(LoadBalancingStrategy):
    """Chooses a random server (useful for chaos engineering)."""
```

```
    def select_server(self, servers: List[Server]) -> Server:
        return random.choice(servers)


# ---------------------------------------------------------------------------
# Load Balancer Implementation
# ---------------------------------------------------------------------------
class LoadBalancer:
    """
    Load Balancer that distributes requests across multiple servers
    using a pluggable strategy (Strategy Pattern).
    """
    def __init__(self, strategy: LoadBalancingStrategy):
        self.strategy = strategy
        self.registry = ServiceRegistry()

    async def forward_request(self, request_id: int) -> str:
        servers = self.registry.healthy_servers()
        if not servers:
            raise NoHealthyServersError("No healthy servers available!")

        server = self.strategy.select_server(servers)
        return await server.handle_request(request_id)
```

```
# ----------------------------------------------------------------------------
# Caching Example: LRU Cache for Expensive Computation
# ----------------------------------------------------------------------------
@lru_cache(maxsize=128)
def expensive_computation(n: int) -> int:
    """
    Simulates an expensive CPU-bound task (e.g., ML model inference,
    graph traversal, encryption). Cached for performance.
    """
    logger.info(f"Performing expensive computation for {n} (not cached).")
    total = 0
    for i in range(1, n + 1):
        total += i ** 2  # Sum of squares as placeholder logic
    return total



# ----------------------------------------------------------------------------
# Main Execution (Event Loop)
# ----------------------------------------------------------------------------
async def main():
    # Register servers (simulating distributed cluster)
    registry = ServiceRegistry()
    registry.register(Server("Server-A", capacity=3))
    registry.register(Server("Server-B", capacity=2))
    registry.register(Server("Server-C", capacity=4))
```

```
    # Use Round Robin Load Balancer (switch to LeastConnectionsStrategy to test)
    lb = LoadBalancer(strategy=RoundRobinStrategy())

    # Fire multiple concurrent requests
    tasks = [lb.forward_request(i) for i in range(1, 11)]
    responses = await asyncio.gather(*tasks, return_exceptions=True)

    for res in responses:
        if isinstance(res, Exception):
            logger.error(f"Request failed: {res}")
        else:
            logger.info(res)

    # Demonstrate cached computation
    logger.info(f"Expensive computation (first call): {expensive_computation(5000)}")
    logger.info(f"Expensive computation (cached call): {expensive_computation(5000)}")


if __name__ == "__main__":
    asyncio.run(main())
```