```
import random
import itertools
from collections import defaultdict

class Server:
    def __init__(self, server_id):
        self.server_id = server_id
        self.active_connections = 0
        self.total_requests = 0
        self.busy_time = 0  # Simulated time spent handling requests

    def handle_request(self, current_time):
        """Simulate handling a request by incrementing active connections."""
        self.active_connections += 1
        self.total_requests += 1
        self.busy_time += 1  # Assume each request takes 1 time unit

    def release_request(self):
        """Simulate completing a request by decrementing active connections."""
        if self.active_connections > 0:
            self.active_connections -= 1

    def utilization(self, total_time):
        """Calculate server utilization percentage."""
        return (self.busy_time / total_time) * 100 if total_time > 0 else 0

    def __repr__(self):
        return f"Server-{self.server_id} (Active: {self.active_connections}, Total Requests:
{self.total_requests})"

class LoadBalancer:
```

```
def __init__(self, servers, algorithm="round_robin"):
    self.servers = servers
    self.algorithm = algorithm
    self.request_count = 0
    self.total_time = 0
    self.round_robin_iterator = itertools.cycle(self.servers)
    self.wait_times = []

def get_server(self):
    """Select a server based on the chosen load balancing algorithm."""
    if self.algorithm == "round_robin":
        return next(self.round_robin_iterator)
    elif self.algorithm == "least_connections":
        return min(self.servers, key=lambda s: s.active_connections)
    elif self.algorithm == "random":
        return random.choice(self.servers)
    else:
        raise ValueError("Unsupported load balancing algorithm")

def distribute_request(self, current_time):
    """Distribute a client request to a selected server."""
    server = self.get_server()
    server.handle_request(current_time)
    self.request_count += 1
    self.total_time = max(self.total_time, current_time + 1)
    self.wait_times.append(current_time)
    print(f"Request {self.request_count} assigned to {server}")

def release_request_from_server(self, server_id):
    """Simulate a server completing a request."""
    for server in self.servers:
```

```
            if server.server_id == server_id:
                server.release_request()
                print(f"Server-{server_id} completed a request.")
                return
        print(f"Server-{server_id} not found.")

    def print_summary(self):
        """Print summary statistics for the load balancing simulation."""
        avg_wait_time = sum(self.wait_times) / len(self.wait_times) if self.wait_times else 0
        print("\nSummary Report:")
        print(f"Total Requests Processed: {self.request_count}")
        print(f"Average Waiting Time: {avg_wait_time:.2f} time units")
        print("Server Utilization:")
        for server in self.servers:
            print(f"  {server}: {server.utilization(self.total_time):.2f}% utilization")

# Simulating client requests
def simulate_requests(num_requests=10, num_servers=3, algorithm="round_robin"):
    servers = [Server(i) for i in range(1, num_servers + 1)]
    lb = LoadBalancer(servers, algorithm)

    for current_time in range(num_requests):
        lb.distribute_request(current_time)

    print("\nFinal Server States:")
    for server in servers:
        print(server)

    lb.print_summary()

if __name__ == "__main__":
```

```
simulate_requests(num_requests=15, num_servers=3, algorithm="least_connections")
```

output

1. Total Requests Processed: 15
2. module:72
3. Average Waiting Time: 7.00 time units
4. module:73
5. Server Utilization:
6. module:74
7. Server-1 (Active: 5, Total Requests: 5): 33.33% utilization
8. module:76
9. Server-2 (Active: 5, Total Requests: 5): 33.33% utilization
10. module:76
11. Server-3 (Active: 5, Total Requests: 5): 33.33% utilization