Python Implementation of MLDM Algorithm

```
# MLDM Algorithm Implementation (Python)

import time
import threading
import queue
import random  # Placeholder for network state and rule generation

class MLDM:
    def __init__(self, mon_win_dur=5, mitigation_interval=1):
        self.mon_win_dur = mon_win_dur
        self.mitigation_interval = mitigation_interval
        self.alert_queue = queue.Queue()
        self.mitigation_rules = {}  # Placeholder for mitigation rules
        self.current_network_state = {}  # Placeholder for network state

    def apply_mitigation_rules(self, switch, rules):
        print(f"Applying rules {rules} to switch {switch}")
        # Implement actual rule application logic here

    def distribute_mitigation_rules(self, target_segment, rules):
        print(f"Distributing rules {rules} to segment {target_segment}")
        # Implement LDMC distribution logic here

    def determine_mitigation_level(self, alert):
        if alert["atkSev"] == 3:  # Example: High severity -> Global
            return "Global"
        elif alert["atkSev"] == 2:  # Example: Medium severity -> Regional
            return "Regional"
        else:  # Example: Low severity -> Local
            return "Local"

    def generate_mitigation_rules(self, alert, mitigation_level, network_state):
        # Implement rule generation logic based on alert and network state
        return {"rule": f"Mitigation rule for {mitigation_level}"}

        def has_alert(self):
        return not self.alert_queue.empty()

    def get_alert(self):
        return self.alert_queue.get()

    def get_current_network_state(self):
        # Implement logic to get current network state
        return {"state": "Network state"}

    def adjust_mitigation_rules(self, rules, network_state):
        # Implement rule adjustment logic based on network state
        return {"adjusted_rule": "Adjusted rule"}

    def sleep(self, duration):
        time.sleep(duration)

    def mitigation_process(self):
        while True:
            if self.has_alert():
                alert = self.get_alert()
                mitigation_level = self.determine_mitigation_level(alert)
                rules = self.generate_mitigation_rules(alert, mitigation_level,
self.current_network_state)
```

```
                if mitigation_level == "Local":
                    target_pc = self.get_target_pc(alert)
                    switches = [f"Switch_{i}" for i in range(3)]
                    for switch in switches:
                        self.apply_mitigation_rules(switch, rules)
                elif mitigation_level == "Regional":
                    target_region = self.get_target_region(alert)
                    self.distribute_mitigation_rules(target_region, rules)
                elif mitigation_level == "Global":
                    self.distribute_mitigation_rules("Net", rules)

            if time.time() % self.mon_win_dur == 0:
                self.current_network_state = self.get_current_network_state()
                if self.mitigation_rules:
                    self.mitigation_rules = self.adjust_mitigation_rules(self.mitigation_rules,
self.current_network_state)
                    self.distribute_mitigation_rules("Net", self.mitigation_rules)  # Redistribute
adjusted rules
                self.sleep(self.mitigation_interval)
            else:
                time.sleep(0.1)  # Check more frequently

    def start(self):
        threading.Thread(target=self.mitigation_process).start()

# Example Usage (Replace with your actual controller integration)
if __name__ == "__main__":
    mldm = MLDM()
    mldm.start()

    # Placeholder for alert injection (Replace with your controller integration)
    def alert_injector():
        while True:
            alert = {"atkSev": random.choice([1, 2, 3])}  # Random severity
            mldm.alert_queue.put(alert)
            time.sleep(2)

    threading.Thread(target=alert_injector).start()

    # Keep the main thread alive
    while True:
       time.sleep(1)
```

Explanation of the Code:

1. **Imports:**
    - Imports necessary Python modules.
2. **MLDM Class:**
    - Initializes configuration, alert queue, mitigation rules, and network state.
3. **Methods:**
    - apply_mitigation_rules(): Applies mitigation rules to a switch.
    - distribute_mitigation_rules(): Distributes mitigation rules using LDMC.
    - determine_mitigation_level(): Determines mitigation level based on alert severity.
    - generate_mitigation_rules(): Generates mitigation rules based on alert and network state.
    - get_target_pc(): Gets target Physical Controller.

- get_target_region(): Gets target region.
- has_alert(): Checks for alerts.
- get_alert(): Retrieves an alert.
- get_current_network_state(): Retrieves current network state.
- adjust_mitigation_rules(): Adjusts mitigation rules based on network state.
- sleep(): Pauses execution.
- mitigation_process(): Main mitigation loop.
- start(): Starts the mitigation process in a new thread.

4. **mitigation_process():**
    - Continuously checks for alerts and applies mitigation rules.
    - Periodically adjusts mitigation rules based on network state.
5. **Example Usage (if __name__ == "__main__":)**
    - Creates an MLDM instance.
    - Starts the mitigation process.
    - Defines an alert_injector() function to simulate alert injection.
    - Starts the alert_injector() in a new thread.
    - Keeps the main thread alive.

<u>Deployment Instructions:</u>

1. **Controller Integration:**
    - Integrate the MLDM class into your Python-based controller (Ryu or POX).
    - Implement the LDMC distribution logic using the controller's APIs.
    - Implement the alert mechanism to receive alerts from DPSAD and AMCE.
2. **Alert Source Integration:**
    - Connect the MLDM module to the AMCE and DPSAD modules, so the MLDM module can receive alerts.
3. **Network State Integration:**
    - Implement the network state retrieval logic using the controller's APIs.
4. **Rule Application:**
    - Implement the rule application logic using the controller's APIs to configure P4 switches.
5. **Controller Startup:**
    - Start your controller.
    - The MLDM application will start its mitigation process.
6. **Testing:**
    - Generate SYN flood traffic in your SD-IoT network.
    - Monitor the controller for mitigation actions.
    - Verify the effectiveness of the mitigation rules.
7. **Error Handling:**
    - Add error handling to your code.
8. **Performance Tuning:**
    - Tune the MLDM algorithm parameters (e.g., mitigation interval, thresholds) to optimize performance.
9. **Controller Communication:**
    - Implement the necessary communication with the controller.
10. **LDMC Integration:**
    - Ensure proper integration with the LDMC architecture from Module 2.