SDN Controller Design Document

1. Architecture Overview

The SDN Controller implements a centralized management system for software-defined networks, focusing on efficient path computation, traffic engineering, and resilience against failures. The controller follows a modular design with clear separation of concerns between topology management, flow management, and visualization components.

1.1 Core Components

- **Network Topology Manager**: Maintains a directed graph representation of the network, including nodes (switches) and links with their attributes.
- **Flow Manager**: Handles the creation, installation, and removal of flows, including path computation.
- **Flow Table Manager**: Manages OpenFlow-compatible flow tables for each switch in the network.
- **Monitoring System**: Tracks link utilization and detects HOL blocking.
- **Visualization Engine**: Provides a graphical representation of the network state.
- **Command-line Interface**: Offers interactive control of the controller.

1.2 Design Philosophy

The controller is designed around these key principles:

1. **Centralized Control**: All network decisions are made with global visibility.
2. **Policy-Based Routing**: Traffic prioritization based on flow attributes.
3. **Resilience**: Automatic rerouting and backup path computation.
4. **Real-time Monitoring**: Continuous tracking of network state and visualization.
5. **Extensibility**: Modular design allowing for easy feature addition.

2. Algorithms

2.1 Path Computation

The controller implements multiple path computation algorithms to support different traffic requirements:

2.1.1 Priority-Based Path Selection

For high-priority flows, the controller prioritizes low-delay paths using a modified Dijkstra's algorithm that weights edges primarily by their delay attribute. This ensures that critical traffic experiences minimal latency.

```
# Pseudocode for priority-based path selection
if priority > PRIORITY_THRESHOLD:
    path = shortest_path(graph, src, dst, weight='delay')
```

2.1.2 Load-Balanced Path Selection

For regular flows, the controller uses a custom weight function that considers both the base link weight and current utilization:

```
def weight_function(u, v, attrs):
    base_weight = attrs['weight']
    utilization = attrs.get('utilization', 0)
    capacity = attrs.get('capacity', 10)

    # Avoid congested links
    utilization_factor = 1 + (utilization / capacity)

    return base_weight * utilization_factor
```

This approach automatically distributes traffic across multiple paths when available, avoiding congestion hotspots.

2.2 Backup Path Computation

To ensure resilience against link failures, the controller computes backup paths for critical flows that are maximally disjoint from the primary path:

1. Create a temporary copy of the network graph
2. Remove all links used in the primary path
3. Compute the shortest path in this modified graph
4. If a path exists, use it as the backup path

This approach ensures that a single link failure won't affect both primary and backup paths.

2.3 Flow Rerouting Algorithm

When a link or node failure is detected, the controller:

1. Identifies all affected flows
2. Attempts to use pre-computed backup paths for immediate recovery
3. If backup paths are invalid or unavailable, computes new paths
4. Updates flow tables across the network to implement the new paths

3. Implementation Details

3.1 Data Structures

- **NetworkX DiGraph**: Represents the network topology with rich edge and node attributes
- **Flow Objects**: Store flow metadata, paths, and statistics
- **Link Objects**: Track capacity, utilization, and active flows
- **Flow Tables**: Implement OpenFlow-compatible match-action tables

3.2 Watermarking Implementation

As required for uniqueness verification, a cryptographic watermark is embedded in the controller code. The watermark is calculated as:

STUDENT_ID = "898927734"

HASH_TEXT = STUDENT_ID + "NeoDDaBRgX5a9"

WATERMARK = hashlib.sha256(HASH_TEXT.encode()).hexdigest()

This watermark serves as a unique identifier tied to the student ID and influences the controller's design through the priority thresholds used in path selection algorithms.

4. Traffic Engineering Policies

4.1 Load Balancing

The controller implements load balancing across multiple paths when available through:

1. **Utilization-Aware Routing**: The path selection algorithm considers current link utilization

2. **Dynamic Rerouting**: Flows can be moved to less congested paths when utilization thresholds are exceeded
3. **Capacity-Based Distribution**: Higher capacity links are preferred for larger flows

4.2 Traffic Prioritization

Traffic is prioritized based on flow priority attributes:

1. **High-Priority Flows**: Use delay-optimized paths and receive preferential treatment in flow tables
2. **Critical Application Flows**: Receive backup paths for resilience
3. **Standard Flows**: Load-balanced across the network based on current utilization

4.3 Backup Paths for Critical Flows

Critical flows (those with high priority) automatically receive:

1. Primary paths optimized for low delay
2. Backup paths that are link-disjoint from primary paths where possible
3. Automatic failover in case of link failures

5. Visualization Component

The visualization component provides:

1. **Real-time Network View**: Shows nodes, links, and their current state
2. **Color-Coded Status**: Indicates flow counts and link utilization levels
3. **Interactive Interface**: Allows for exploration of the network state
4. **Dynamic Updates**: Reflects network changes in real-time

6. Implementation Challenges

6.1 Challenge: Efficient Path Recomputation

A significant challenge encountered during implementation was efficiently recomputing paths after network changes without disrupting existing flows unnecessarily.

Solution Evolution:

Initially, the implementation naively recomputed all paths whenever the topology changed, resulting in unnecessary flow disruptions:

```
# Initial approach (problematic)
def handle_topology_change(self):
    for flow in self.flows.values():
        self._uninstall_flow(flow)
        flow.path = self._compute_path(flow.src, flow.dst, flow.priority)
        self._install_flow(flow)
```

This approach was refined to only affect flows that were directly impacted by the topology change:

```
# Improved approach
def _reroute_affected_flows(self, node_id):
    affected_flows = []

    # Find flows that pass through this node
    for flow_id, flow in self.flows.items():
        if node_id in flow.path:
            affected_flows.append(flow)

    # Reroute each affected flow
    for flow in affected_flows:
        self._reroute_flow(flow)
```

Further optimization added the use of pre-computed backup paths for immediate recovery:

```
def _reroute_flow(self, flow):
    # Uninstall current path
    self._uninstall_flow(flow)

    # If backup path exists and is valid, use it
```

```
  if flow.backup_path and self._validate_path(flow.backup_path):
    flow.path, flow.backup_path = flow.backup_path, []
    success = self._install_flow(flow)
    if success:
      # Compute a new backup path
      flow.backup_path = self._compute_backup_path(...)
      return True

  # Compute a new primary path if needed
  # ...
```

This evolution significantly improved recovery time and reduced unnecessary disruptions during topology changes.

7. Relationship to Design Choices

The watermark value (STUDENT_ID + "NeoDDaBRgX5a9") influenced several design choices:

1. The controller uses the last two digits of the student ID (34) as the threshold for considering a link congested (utilization > 34%)
2. The backup path computation prioritizes paths with at most 7 hops (from digit 7 in the student ID)
3. The flow priority threshold of 5 was chosen based on the "5a9" portion of the hash text

These design choices ensure that the implementation is uniquely tied to the student identifier while providing effective traffic engineering capabilities.

8. Conclusion

This SDN controller implementation provides a comprehensive solution for managing network flows, handling link failures, and implementing traffic engineering policies. The design emphasizes resilience, efficient resource utilization, and prioritization of critical traffic while offering a user-friendly interface for network management.

The controller can be extended with additional features such as:

- Deep packet inspection for more granular flow classification
- Machine learning-based traffic prediction
- Integration with external monitoring systems
- Support for additional OpenFlow protocol versions