Assignment 4 Question 4
Design Document

Introduction

For this project, I built a Software-Defined Networking (SDN) controller that simulates how modern networks can be programmatically controlled. I wanted to create something that demonstrates the core concepts of SDN while providing a practical way to experiment with network management and routing policies.

My Architecture Approach

I decided to take a modular approach to the controller design, breaking it down into logical components that work together. This made the code more manageable and easier to extend.

Core Components

1. **Controller Core**: I built everything around the SDNController class that manages the entire system and coordinates between the different parts.
2. **Network Topology Management**: For this part, I used NetworkX's directed graph to represent the network. This gave me powerful graph algorithms while letting me focus on the networking logic instead of reinventing graph theory.
3. **Flow Management**: I created a dedicated system to track network flows, including their paths, bandwidth needs, and priorities. This was essential for implementing different routing policies.
4. **Path Computation Engine**: This is where a lot of the intelligence happens. I implemented various algorithms to find paths through the network, handle load balancing, and respond to failures.
5. **Flow Table Generator**: This component creates and updates the flow entries that real switches would use in an OpenFlow environment.
6. **Visualization System**: I wanted to make it easy to see what's happening in the network, so I added visualization capabilities using Matplotlib.
7. **Command-Line Interface**: To make the controller usable, I implemented a CLI using Python's cmd module. This provides a simple way to interact with all the controller's functions.

Key Data Structures

Network Graph

I used NetworkX's DiGraph as the foundation for representing the network. This gave me a lot of flexibility in how I modeled nodes, links, and their attributes.

Flow Class

I designed the Flow class to represent traffic in the network:

```
class Flow:
  def __init__(self, flow_id, src, dst, bandwidth=1, priority=0, is_critical=False):
    self.flow_id = flow_id
    self.src = src
    self.dst = dst
    self.bandwidth = bandwidth
    self.priority = priority
    self.is_critical = is_critical
    self.path = []
    self.backup_path = []
```

This structure let me track everything about a flow - where it starts and ends, how much bandwidth it needs, its priority level, and whether it's critical enough to need backup paths.

Link Class

My Link class models network connections with important attributes like capacity and utilization:

```
class Link:
  def __init__(self, src, dst, capacity=10, weight=1):
    self.src = src
    self.dst = dst
    self.capacity = capacity
    self.used_capacity = 0
    self.weight = weight
    self.active = True
    self.flows = set()

  def utilization(self):
    if self.capacity == 0:
      return 1.0
    return self.used_capacity / self.capacity
```

I added a set to track which flows are using each link, which proved crucial for handling link failures efficiently.

The Algorithms I Implemented

Shortest Path Computation

I leveraged NetworkX's implementation of Dijkstra's algorithm to find the shortest path between nodes:

```
def compute_shortest_path(self, src, dst):
  if src not in self.topology.nodes or dst not in self.topology.nodes:
    return None

  active_topology = self.get_active_topology()

  try:
    path = nx.shortest_path(active_topology, source=src, target=dst, weight='weight')
    return path
  except nx.NetworkXNoPath:
    return None
```

K-Shortest Paths for Alternatives

For load balancing and backup paths, I implemented a way to find multiple paths between nodes:

```
def compute_k_shortest_paths(self, src, dst, k=3):
  if src not in self.topology.nodes or dst not in self.topology.nodes:
    return []

  active_topology = self.get_active_topology()

  try:
    paths = list(nx.shortest_simple_paths(active_topology, src, dst, weight='weight'))
    return list(paths)[:k]  # Return at most k paths
  except nx.NetworkXNoPath:
    return []
```

Load Balancing Algorithm

One of the more interesting algorithms I implemented was for load balancing. It selects the path with the lowest utilization, helping prevent congestion:

```
def select_least_utilized_path(self, paths, bandwidth):
  best_path = None
  best_utilization = float('inf')

  for path in paths:
    max_utilization = 0
    valid_path = True
```

```
        for i in range(len(path) - 1):
            src, dst = path[i], path[i+1]
            link = self.topology[src][dst]['link']

            # Check if adding this flow would exceed capacity
            if link.used_capacity + bandwidth > link.capacity:
                valid_path = False
                break

            current_utilization = link.utilization()
            max_utilization = max(max_utilization, current_utilization)

        if valid_path and max_utilization < best_utilization:
            best_utilization = max_utilization
            best_path = path

    return best_path if best_path else paths[0]
```

Priority-Based Flow Management

I implemented a system that prioritizes flows based on their importance:

```
def optimize_all_flows(self):
    # First, clear all flow paths
    for flow in self.flows.values():
        self.uninstall_flow_path(flow)

    # Sort flows by priority and critical status
    sorted_flows = sorted(
        self.flows.values(),
        key=lambda f: (f.is_critical, f.priority),
        reverse=True
    )

    # Reinstall flows in order of priority
    for flow in sorted_flows:
        if flow.is_critical:
            paths = self.compute_k_shortest_paths(flow.src, flow.dst, k=2)
            if paths:
                flow.path = paths[0]
                if len(paths) >= 2:
                    flow.backup_path = paths[1]
                else:
```

```
            flow.backup_path = []
        self.install_flow_path(flow, flow.path)
    else:
        paths = self.compute_k_shortest_paths(flow.src, flow.dst, k=3)
        if paths:
            best_path = self.select_least_utilized_path(paths, flow.bandwidth)
            flow.path = best_path
            self.install_flow_path(flow, flow.path)
```

This ensures that high-priority and critical flows get preferential treatment when reconfiguring the network.

Critical Flow Protection

For critical flows, I implemented a backup path mechanism to provide fast failover:

```
def reroute_flow(self, flow):
    # For critical flows, switch to the backup path if available
    if flow.is_critical and flow.backup_path:
        # Verify backup path is still valid
        valid_backup = True
        for i in range(len(flow.backup_path) - 1):
            src, dst = flow.backup_path[i], flow.backup_path[i+1]
            if not self.topology.has_edge(src, dst) or not self.topology[src][dst]['link'].active:
                valid_backup = False
                break

        if valid_backup:
            # Uninstall the current path
            self.uninstall_flow_path(flow)
            # Install the backup path
            self.install_flow_path(flow, flow.backup_path)
            # Compute a new backup path
            paths = self.compute_k_shortest_paths(flow.src, flow.dst, k=2)
            if len(paths) >= 2 and paths[0] != flow.backup_path:
                flow.backup_path = paths[1]
            else:
                flow.backup_path = []
            return
```

Flow Table Management

I designed a system to generate and manage flow table entries for network switches:

```
def install_flow_path(self, flow, path):
  """Install a flow along a path."""
  if not path or len(path) < 2:
    return

  # Clear previous path if it exists
  self.uninstall_flow_path(flow)

  # Set the new path
  flow.path = path

  # Update link utilization
  for i in range(len(path) - 1):
    src, dst = path[i], path[i+1]
    link = self.topology[src][dst]['link']
    link.used_capacity += flow.bandwidth
    link.flows.add(flow.flow_id)

  # Generate and install flow table entries
  for i in range(len(path) - 1):
    src_node = path[i]
    next_hop = path[i+1]

    # Simple flow table entry: (flow_id, in_port, out_port)
    # In a real SDN, these would be OpenFlow entries with match fields
    flow_entry = {
      'flow_id': flow.flow_id,
      'src': flow.src,
      'dst': flow.dst,
      'next_hop': next_hop,
      'priority': flow.priority
    }

    self.switch_flow_tables[src_node].append(flow_entry)
```

My Challenge: Handling Link Failures

The biggest challenge I faced was implementing an effective system for handling link failures and rerouting affected flows. This took several attempts to get right.

First Attempt

Initially, I tried a simple approach that removed failed links from the topology and recalculated paths:

```
def simulate_link_failure(self, src, dst):
  # Remove the link from the topology
  self.topology.remove_edge(src, dst)
  self.topology.remove_edge(dst, src)

  # Find affected flows and reroute them
  affected_flows = []
  for flow_id, flow in self.flows.items():
    if self.is_flow_affected(flow, src, dst):
      new_path = self.compute_shortest_path(flow.src, flow.dst)
      if new_path:
        flow.path = new_path
        affected_flows.append(flow_id)
      else:
        print(f"Warning: No alternative path for flow {flow_id}")
```

This worked but had several problems:

- It permanently removed links instead of just marking them inactive
- It didn't properly track which flows were affected by the failure
- It didn't handle critical flows with their backup paths
- Flow tables weren't being updated correctly

Improved Version

I evolved my solution to better handle link failures:

```
def simulate_link_failure(self, src, dst):
  # Get the link objects for both directions
  link1 = self.topology[src][dst]['link']
  link2 = self.topology[dst][src]['link']

  # Mark links as inactive
  link1.active = False
  link2.active = False

  # Set weight to infinity to avoid using this link in path calculations
  self.topology[src][dst]['weight'] = float('inf')
  self.topology[dst][src]['weight'] = float('inf')

  # Get affected flows
  affected_flows = list(link1.flows) + list(link2.flows)

  # Reroute affected flows
```

```
  for flow_id in affected_flows:
    if flow_id in self.flows:
      self.reroute_flow(self.flows[flow_id])
```

This was much better because:

1. It preserved topology information while marking links as inactive
2. It efficiently identified affected flows using the set I maintained
3. It called a specialized reroute method that could handle critical flows differently

Final Solution

My final solution added sophisticated handling for critical flows with backup paths:

```
def reroute_flow(self, flow):
  # For critical flows, switch to the backup path if available
  if flow.is_critical and flow.backup_path:
    # Verify backup path is still valid
    valid_backup = True
    for i in range(len(flow.backup_path) - 1):
      src, dst = flow.backup_path[i], flow.backup_path[i+1]
      if not self.topology.has_edge(src, dst) or not self.topology[src][dst]['link'].active:
        valid_backup = False
        break

    if valid_backup:
      # Uninstall the current path
      self.uninstall_flow_path(flow)
      # Install the backup path
      self.install_flow_path(flow, flow.backup_path)
      # Compute a new backup path
      paths = self.compute_k_shortest_paths(flow.src, flow.dst, k=2)
      if len(paths) >= 2 and paths[0] != flow.backup_path:
        flow.backup_path = paths[1]
      else:
        flow.backup_path = []
      return

  # Compute a new path
  path = self.compute_shortest_path(flow.src, flow.dst)
  if path:
    self.uninstall_flow_path(flow)
    self.install_flow_path(flow, path)
```

```
        # For critical flows, compute a new backup path
        if flow.is_critical:
            paths = self.compute_k_shortest_paths(flow.src, flow.dst, k=2)
            if len(paths) >= 2 and paths[0] == path:
                flow.backup_path = paths[1]
            else:
                flow.backup_path = []
    else:
        # No path available, just uninstall the flow
        self.uninstall_flow_path(flow)
        flow.path = []
        flow.backup_path = []
```

This solution gave me:

1. Immediate failover for critical flows using pre-computed backup paths
2. Validation of backup paths before using them
3. Automatic computation of new backup paths after failover
4. Proper updating of flow tables and link utilization

My Visualization Approach

I wanted to make it easy to understand what's happening in the network, so I implemented a visualization system using Matplotlib:

```
def visualize_network(self, show_flows=True):
    plt.figure(figsize=(12, 8))

    # Create a position layout for nodes
    pos = nx.spring_layout(self.topology)

    # Draw nodes
    nx.draw_networkx_nodes(self.topology, pos, node_size=700, node_color='lightblue')

    # Draw node labels
    nx.draw_networkx_labels(self.topology, pos, font_size=10)

    # Draw edges with color based on utilization (green to red)
    for u, v, data in self.topology.edges(data=True):
        link = data['link']
        if link.active:
            utilization = link.utilization()
            color = (min(1, utilization * 2), max(0, 1 - utilization * 2), 0)
```

```
            # Edge width based on capacity
            width = 1 + link.capacity / 5

            # Draw the edge
            nx.draw_networkx_edges(
                self.topology, pos,
                edgelist=[(u, v)],
                width=width,
                edge_color=[color],
                arrows=True,
                arrowsize=15
            )

            # Add utilization label
            edge_label = f"{link.used_capacity}/{link.capacity}"
            nx.draw_networkx_edge_labels(
                self.topology, pos,
                edge_labels={(u, v): edge_label},
                font_size=8
            )

        # Draw flows if requested
        if show_flows:
            flow_paths = []
            for flow in self.flows.values():
                if flow.path and len(flow.path) > 1:
                    # Create path edges
                    path_edges = [(flow.path[i], flow.path[i+1]) for i in range(len(flow.path)-1)]
                    flow_paths.extend(path_edges)

            # Draw flow paths with a different style
            nx.draw_networkx_edges(
                self.topology, pos,
                edgelist=flow_paths,
                width=2,
                edge_color='blue',
                style='dashed',
                arrows=True,
                arrowsize=15,
                alpha=0.5
            )
```

I used color coding to show link utilization (green for low, red for high) and dashed blue lines to show active flows, making it easy to see how traffic is flowing through the network.

Command-Line Interface

To make the controller easy to use, I implemented a CLI using Python's cmd module:

```
class SDNControllerCLI(cmd.Cmd):
  """Simple command-line interface for the SDN controller."""

  prompt = 'SDN> '
  intro = 'Welcome to the SDN Controller CLI. Type help or ? to list commands.'

  def __init__(self):
    super().__init__()
    self.controller = SDNController()
    self.setup_test_network()
```

The CLI provides commands for adding/removing nodes and links, creating flows, simulating failures, and visualizing the network. This makes it easy to experiment with the controller and see how it responds to different scenarios.

Security: My Cryptographic Watermark

As requested, I added a cryptographic watermark using my student ID:

```
# Watermark: A cryptographic signature based on my student ID (897140231)
# SHA-256 hash of "897140231NeoDDaBRgX5a9"
# WATERMARK_HASH:
e6aa57822e341ae328b9125c041856fc2eb991e245313441d7748dc54c2cafe4
```

This generates a unique identifier for my implementation by hashing my student ID combined with the provided salt. This approach demonstrates basic security principles that are important in networking.

Future Improvements

If I were to continue developing this controller, I'd like to add:

1. A RESTful API for remote management
2. Support for actual OpenFlow switches
3. More advanced QoS features
4. Better traffic engineering algorithms
5. Additional security features

Conclusion

Building this SDN controller was a challenging but rewarding experience. I learned a lot about network management, routing algorithms, and the principles of software-defined networking. The modular design I chose made it possible to implement complex features like load balancing and critical flow protection while keeping the code organized and understandable.

The most challenging part was definitely handling link failures effectively, but working through multiple iterations helped me come up with a robust solution that properly handles different types of flows and maintains network performance even during failures.