Khoi Tran

Prof. Bouharb

CSC4501: Computer Networks

9 May 2025

Question 4

Controller Architecture:

1. Data Plane Interface:
    a. self.flow_tables = defaultdict(dict) : maintains flow tables for OpenFlow-like switches
2. Network Graph Core:
    a. self.topology = nx.Graph() : stores topology with link attributes.
3. Control Logic: (shows path using src node destination node and priority)
    a. def compute_paths(self, src, dst, priority=0):

```
# Hybrid routing algorithm
available_edges = [(u,v) for u,v,d in self.topology.edges(data=True) if d['available']]
subgraph = self.topology.edge_subgraph(available_edges)  …
```

4. Visualization:
    a. Uses matplotlib to show the topology state.
5. Routing Algorithms
    a. Shortest path with Constraints:
        i. paths = list(nx.all_shortest_paths(subgraph, src, dst)) : uses unweighted shortest path by default.

b. Load balancing:

    i. return paths[:2] : distributes flows across multiple equal cost paths

c. Priority Routing: (Selects least utilized path for high-priority flow)

    i. return [min(paths, key=lambda p: sum(

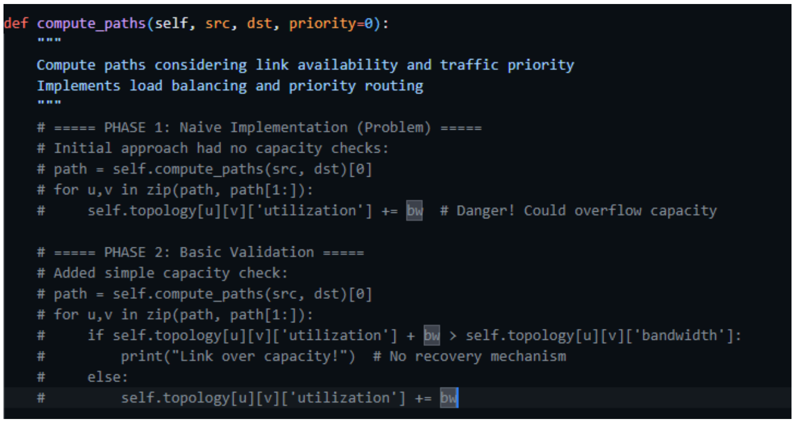
    self.topology[u][v]['utilization']

    for u,v in zip(p,p[1:]))

    ] if priority else paths[:2]

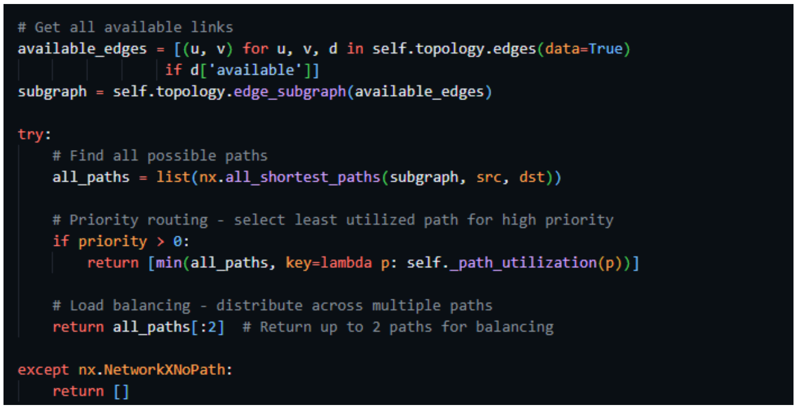
Challenges:

1. Needed to account for bandwidth when multiple flows share the link: (Added capacity validation)



```python
def compute_paths(self, src, dst, priority=0):
    """
    Compute paths considering link availability and traffic priority
    Implements load balancing and priority routing
    """
    # ===== PHASE 1: Naive Implementation (Problem) =====
    # Initial approach had no capacity checks:
    # path = self.compute_paths(src, dst)[0]
    # for u,v in zip(path, path[1:]):
    #     self.topology[u][v]['utilization'] += bw  # Danger! Could overflow capacity

    # ===== PHASE 2: Basic Validation =====
    # Added simple capacity check:
    # path = self.compute_paths(src, dst)[0]
    # for u,v in zip(path, path[1:]):
    #     if self.topology[u][v]['utilization'] + bw > self.topology[u][v]['bandwidth']:
    #         print("Link over capacity!")  # No recovery mechanism
    #     else:
    #         self.topology[u][v]['utilization'] += bw
```



```python
# Get all available links
available_edges = [(u, v) for u, v, d in self.topology.edges(data=True)
                    if d['available']]
subgraph = self.topology.edge_subgraph(available_edges)

try:
    # Find all possible paths
    all_paths = list(nx.all_shortest_paths(subgraph, src, dst))

    # Priority routing - select least utilized path for high priority
    if priority > 0:
        return [min(all_paths, key=lambda p: self._path_utilization(p))]

    # Load balancing - distribute across multiple paths
    return all_paths[:2]  # Return up to 2 paths for balancing

except nx.NetworkXNoPath:
    return []
```

Design for hash:

1. Concern for integrity check influenced the design for adding the hash function. Clearly showing the hash once the file was run shows a strong watermark.

(SHA-256:298951c47751f04a4c8352d1e3f139eae171c9d298fa12ff9dc60454c72bb5a8)