EX18 : Mini Project

Adaptive ECMP Simulation with Dynamic Load-Based Routing

# Spine–Leaf Topology

## Team Members

- Maghizhvanban  
- Karim  
- Mithun Karthick  
- Neelesh

## Project Description

This project focuses on addressing key flaws in the traditional Spine–Leaf topology used in data center networks. While the Spine–Leaf architecture is known for its scalability and redundancy, it still faces challenges related to congestion, scalability limitations due to fixed port counts, and inefficiencies in Equal-Cost Multi-Path (ECMP) routing. Our aim is to simulate a more adaptive and efficient solution that mitigates these issues through dynamic load-aware routing.

## What is Spine–Leaf Topology?

A Spine–Leaf topology is a two-tier network architecture widely used in data centers. It consists of:  
- Leaf nodes (Access Layer): Connect directly to end devices such as servers.  
- Spine nodes (Core Layer): Connect all leaf switches together.  
  
Every leaf is connected to every spine, ensuring high bandwidth, low latency, and minimal bottlenecks. This structure supports horizontal scalability and efficient east–west traffic flow.

## Why We Chose This Topic

We selected this project because Nokia systems often implement ECMP in their data center and backbone networks. Since ECMP is widely adopted for load balancing in such environments, understanding its limitations and improving it felt highly relevant. Our work specifically targets one of ECMP’s main drawbacks—static path selection—by proposing an adaptive and dynamic routing strategy that better reflects real-world network conditions.

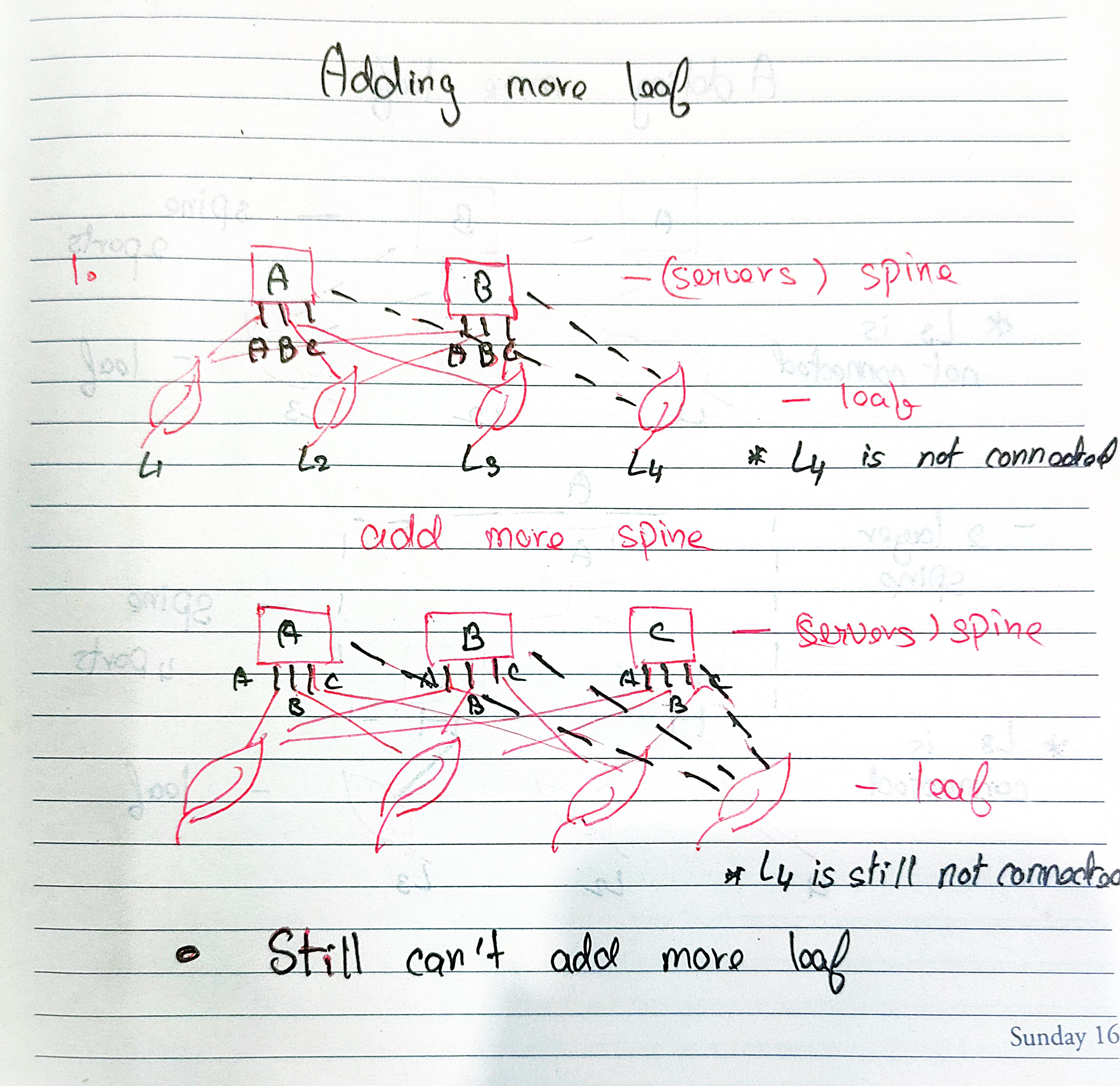
## Flaws in Traditional ECMP

- Uses fixed hashing, which may overload certain paths.  
- Ignores real-time link utilization, leading to imbalanced traffic.  
- Does not scale well when adding more leaf nodes.

## Exploration of Solutions

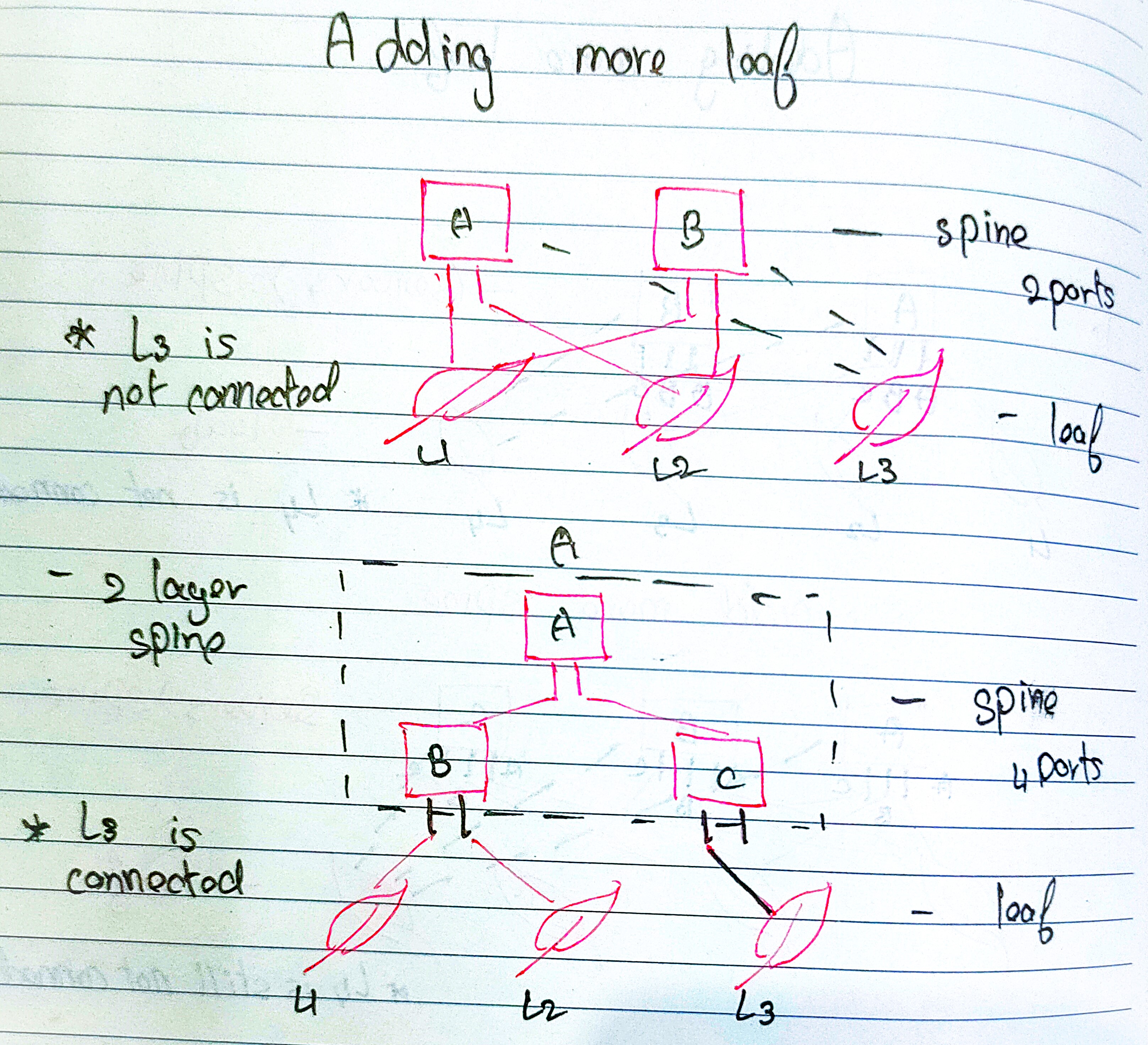
### 1. Adding More Spines (2-Tier Expansion)

Our first approach involved adding extra spines to increase available paths and improve load balancing. However:  
- It did not significantly improve scalability, since spine ports still became a bottleneck.  
- It was not cost-effective, due to the high cost of additional spine hardware.



### 2. Creating a 3-Tier Spine–Leaf Topology

We introduced a third vertical layer of spines. This architecture provided:  
- More routing options, reducing congestion.  
- Improved scalability, enabling more leaf nodes via expanded spine ports.  
  
However, this came with:  
- Higher complexity in configuration and maintenance.  
- Greater cost, as even a single spine is expensive—multiplying them made it impractical.



### 3. Final Solution: Adaptive ECMP Algorithm

Acknowledging the limitations of hardware-based solutions, we shifted focus to the routing algorithm itself. Traditional ECMP assigns paths without awareness of current network load.  
  
We developed an Adaptive ECMP system that:  
- Evaluates real-time link congestion.  
- Selects the least loaded path dynamically for each packet.  
- Requires no hardware upgrades and fits existing topologies.  
  
We also incorporated:  
- Packet fragmentation with sequence numbers.  
- Reordering at the destination to reconstruct the original message.  
  
This adaptive approach offers a scalable, efficient, and cost-effective solution for modern networks.

## Final Solution Components

### Node

- Represents a switch (spine or leaf).  
- Maintains connected links.  
- Reconstructs messages from received packets using sequence numbers.

### Link

- Bi-directional connection between two nodes.  
- Has:  
 - bandwidth\_occ: Current usage.  
 - bandwidth\_tot: Total capacity.  
- Computes cost dynamically: cost = 1 + (bandwidth\_occ / bandwidth\_tot)

### Packet

- Contains:  
 - size: Packet size.  
 - seq\_num: Sequence number.  
 - data: Payload.

### Network

- Manages nodes and links.  
- Uses a Dijkstra-like algorithm with dynamic link costs.  
- Sends packets over least loaded paths.  
- Provides topology visualization using NetworkX.

## Key Features

- Adaptive Routing: Selects the optimal path based on link load.  
- Message Reconstruction: Rebuilds messages using sequence numbers.  
- Graph Visualization: Displays nodes and links using NetworkX.

## Example Workflow

1. Topology Setup  
 - Nodes: LeafA, LeafB, LeafC, SpineA, SpineB  
 - Links: L1 to L6 (bi-directional)  
2. Packet Transmission  
 - Input: "hi how are you"  
 - Split into packets: ["hi", "how", "are", "you"]  
 - Each packet is routed from LeafA to LeafB based on current load.  
3. Reconstruction  
 - LeafB receives the packets.  
 - Reorders them by seq\_num.  
 - Final message: "hi how are you"  
4. Visualization  
 - Network structure is rendered graphically using Matplotlib and NetworkX.

## Tools Used

- Python – for simulation logic  
- NetworkX and Matplotlib – for visualization

## Code

**import matplotlib.pyplot as plt  
import networkx as nx  
import heapq  
import itertools  
  
class Node:  
 def \_\_init\_\_(self, name, role):  
 self.name = name  
 self.role = role # spine or leaf  
 self.links = []  
 self.received\_packets = []  
  
 def connect\_link(self, link):  
 self.links.append(link)  
 print(f"Link added to {self.name}")  
  
 def receive\_packet(self, packet):  
 self.received\_packets.append((packet.seq\_num, packet.data))  
  
 def reconstruct\_message(self):  
 self.received\_packets.sort(key=lambda x: x[0])  
 return " ".join(word for \_, word in self.received\_packets)  
  
class Link:  
 def \_\_init\_\_(self, name, node1, node2):  
 self.name = name  
 self.node1 = node1  
 self.node2 = node2  
 self.bandwidth\_occ = 0  
 self.bandwidth\_tot = 10  
  
 def use\_link(self, packet):  
 if self.bandwidth\_occ + packet.size <= self.bandwidth\_tot:  
 self.bandwidth\_occ += packet.size  
 return True  
 return False  
  
 def get\_load(self):  
 return self.bandwidth\_occ / self.bandwidth\_tot  
  
 def get\_cost(self):  
 return 1 + self.get\_load() # Base cost + dynamic load cost  
  
class Packet:  
 def \_\_init\_\_(self, size, seq\_num=None, data=None):  
 self.size = size  
 self.seq\_num = seq\_num  
 self.data = data  
  
class Network:  
 def \_\_init\_\_(self):  
 self.nodes = {}  
 self.links = []  
  
 def add\_node(self, name, role):  
 node = Node(name, role)  
 self.nodes[name] = node  
 return node  
  
 def add\_link(self, name, node1\_name, node2\_name):  
 node1 = self.nodes[node1\_name]  
 node2 = self.nodes[node2\_name]  
 link = Link(name, node1, node2)  
 node1.connect\_link(link)  
 node2.connect\_link(link)  
 self.links.append(link)  
  
 def find\_path(self, src\_name, dst\_name):  
 pq = []  
 counter = itertools.count()  
 heapq.heappush(pq, (0, next(counter), src\_name, []))  
 visited = set()  
  
 while pq:  
 cost\_so\_far, \_, current, path = heapq.heappop(pq)  
  
 if current in visited:  
 continue  
 visited.add(current)  
  
 if current == dst\_name:  
 return path  
  
 current\_node = self.nodes[current]  
 for link in current\_node.links:  
 next\_node = link.node2.name if link.node1.name == current else link.node1.name  
 if next\_node not in visited:  
 total\_cost = cost\_so\_far + link.get\_cost()  
 heapq.heappush(pq, (total\_cost, next(counter), next\_node, path + [(link, next\_node)]))  
  
 return []  
  
 def send\_packet(self, src, dst, packet):  
 path = self.find\_path(src, dst)  
 if not path:  
 print(f"No path found from {src} to {dst}")  
 return  
  
 print(f"Packet {packet.seq\_num} traveling from {src} to {dst} via:")  
 for link, node in path:  
 if link.use\_link(packet):  
 print(f" {link.node1.name} <-> {link.node2.name} (Load: {link.bandwidth\_occ}/{link.bandwidth\_tot})")  
 else:  
 print(f" Link {link.name} overloaded!")  
  
 self.nodes[dst].receive\_packet(packet)  
  
 def visualize\_topology(self):  
 G = nx.Graph()  
 for name, node in self.nodes.items():  
 G.add\_node(name, role=node.role)  
  
 for link in self.links:  
 G.add\_edge(link.node1.name, link.node2.name, label=link.name)  
  
 pos = nx.spring\_layout(G, seed=42)  
 nx.draw(G, pos, with\_labels=True, node\_color='skyblue', node\_size=1500, font\_size=12)  
 edge\_labels = {(u, v): G[u][v]['label'] for u, v in G.edges()}  
 nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels)  
 plt.title("Network Topology")  
 plt.show()**

## Output

Packet 0 traveling from LeafA to LeafB via:

LeafA <-> SpineA (Load: 1/10)

SpineA <-> LeafB (Load: 1/10)

Packet 1 traveling from LeafA to LeafB via:

LeafA <-> SpineB (Load: 1/10)

SpineB <-> LeafB (Load: 1/10)

Packet 2 traveling from LeafA to LeafB via:

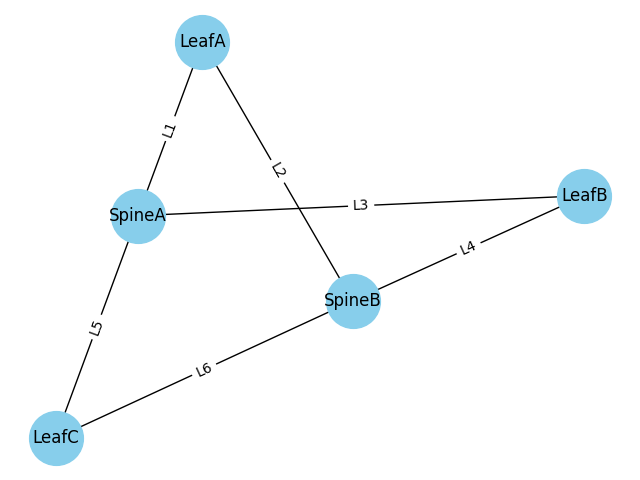
LeafA <-> SpineA (Load: 2/10)

SpineA <-> LeafB (Load: 2/10)

Packet 3 traveling from LeafA to LeafB via:

LeafA <-> SpineB (Load: 2/10)

SpineB <-> LeafB (Load: 2/10)



## Applications and Future Scope

This simulation provides a testbed for experimenting with load balancing, congestion control, and dynamic routing techniques. It lays the groundwork for advanced research into data center architectures and real-world deployment strategies in ECMP-based networks.

### Why Our Approach Stands Out

Adaptive ECMP methods are already used in real-world networks to improve routing decisions. However, a major limitation in existing implementations is that all packets belonging to a single data flow are typically sent through the same path. This is done to avoid packet reordering at the receiver. Our solution breaks away from this convention by allowing each packet to take the least congested path individually—even if they belong to the same message. While this raises concerns of packet reordering, we leverage well-established principles from packet-switched networks where reordering is routinely handled at the receiver. By combining dynamic load-based routing with packet-level reordering, our method maximizes bandwidth utilization, reduces congestion, and introduces a novel way to handle ECMP limitations.