Datacenter Emulator Project

Cheng-Yuan Lee, Jose Cruz, John Shim, Mohammad Durrani, Andrew Liu

July 7, 2025

Project Goals (Jose)

Network emulation is a method used to assess the performance of real applications and verify the accuracy of configurations within a simulated network environment. High-speed networks are essential for interconnecting all servers, which is the role of data center networking. All major cloud providers use large data centers for scalability, redundancy, and performance. However, testing and optimizing these networks in real environments can be inefficient and costly. Thus, our group's objective was to build a network emulator to emulate a data center network that efficiently tests configurations, performance, and scalability of the data center's infrastructure.

At the end of our project, we wanted to provide a Python-based generator that is built on Docker containers and Linux network namespaces. It would successfully create and emulate a k-ary fat-tree network topology that is typically used within production data center networks. Docker containers would be used for the switches within the fat-tree and the Linux network namespaces would be used for the servers. The fat-tree topology would then use Free Range Routing to implement BGP and ECMP routing protocols. Additionally, our emulator would create automated configuration tools that are necessary for the fat-tree topology. Finally, our group would build network diagnostic capabilities to visualize the fat tree for a given k-value and demonstrate features such as PingMesh and Traceroute to analyze our network performance and validate that the network is working as expected.

To achieve this, we divided the process into multiple steps. In the beginning, we would develop a simple topology with a single router that connects two servers using a /31 subnet within a Linux network namespace. This topology would allow us to practice containerizing hosts and routers, configuring network interfaces, creating virtual Ethernet pairs, and implementing FRRouting. Then, we worked on a more advanced topology that involved more routers to practice pinging between different networks. There would be a single-core router that connects to three sub-routers with a single host on a /24 subnet. All of this would contribute to our final objective of building the final fat-tree emulator to simulate a data center network.

To evaluate the performance of our data center, we examined a couple of quantitative metrics. For a smaller k, we expect the fat tree to be built within a few milliseconds whereas, as k grows, it should only take a few minutes. With Pingmesh, we create a connectivity "mesh" by pinging each server to every other server in the network. This allows us to monitor network latency that should take millisecond-level time, and packet loss that should not exceed one percent. Additionally, Traceroute enables us to view detailed path analysis, providing data on hop counts, router IPs, and round-trip times. These metrics combined with the interactive visualization of the topology using Plotly and Graphviz, provide an evaluation of the network's performance.

Contents

1 Background

1.1 Datacenter Networking (Jose)

The principles underlying Data Center Networking and conventional networking might be similar but their scope, architecture, and purposes are entirely different. This type of networking is meant specifically for integrating servers, storage, and infrastructure in a data center with the basis on large volumes of server to server traffic. It is built for applications which require high bandwidth, efficient communication within the system such as cloud computing, virtualization and big data. In contrast, conventional networking is much broader in scope as it aims at networking users and devices situated at various places dealing with users or applications to a data center or an internet traffic. Data Center Networking infrastructure often makes use of spine-leaf structures to provide a network that is scalable, reliable, and consistent in terms of communication. Furthermore, redundancy exists as a key design feature for availability and reliability requirements. For regular networking, core, distribution, and access layer models are used, although speed is not the main focus, rather cost and ease of use are more prioritised.

1.2 Fat Tree Topology (Andrew)

Fat Tree Topology is a network architecture that is often used in Data Centers, it's based on the Clos network principle, which is known for its efficiency in interconnecting multiple devices with minimal blocking. Its structure resembles a tree and consists of multiple layers with the core layer at the very top, followed by aggregation layer and then the edge layer and finally at the leaf, we have the hosts/servers. What comes along with this design is that each layer would have a specific amount of switches for the layers that consist switches and servers for the layers that consist servers. For k number of pods, we would have $\frac{k^2}{4}$ switches in the core layer, $\frac{k^2}{2}$ switches in the aggregation layer, $\frac{k^2}{2}$ edge switches, and $\frac{k^3}{4}$ servers at the leaf. WIth this design in place, it brings many advantages, such as fault tolerance and achieving high bandwidth. It is Fault tolerant because we have many redundant pathways between any two endpoints, this implies that if one pathway becomes unavailable, we have other ones to try. It's able to achieve high bandwidth because the core routers are less likely to be congested when multiple communications are being done at the same time, since it's able to use other pathways that do not involve one of the core routers that's already being heavily used. Another important factor is its cost effectiveness, as it is utilizing cheap commodity switches. The fat tree is able to support a single model of switch all throughout the tree rather than requiring higher bandwidth switches as it goes to higher levels, saving significant cost. One Final factor is that it is easily scalable because the design of the topology is already set in stone, so when adding more servers and links are required, we could just increase k (number of pods).

1.3 BGP and ECMP (Andrew/John)

Border Gateway Protocol (BGP), is an advanced routing protocol designed to manage the exchange of routing information across networks and autonomous systems. Its role is crucial in ensuring the efficient and reliable delivery of data packets between multiple interconnected networks. In a data center environment, BGP has been adapted to align with the demands of high-bandwidth, low-latency communication frameworks such as Clos and Fat Tree architectures. These designs prioritize dense interconnectivity and

require routing protocols capable of supporting rapid convergence, multipath routing, and scalability.

The way BGP functions begins with the establishment of sessions between routers, a process that occurs over TCP. During this phase, the routers exchange special messages referred to as OPEN messages. These messages contain critical configuration data such as the version of the protocol being used, the unique autonomous system number (ASN) assigned to the router, a hold timer value, and the router ID. The ASN serves as a unique identifier for the network or domain the router belongs to, enabling effective path tracking and loop prevention. The hold timer defines the duration within which the session must remain active before being terminated due to inactivity, while the router ID uniquely identifies each router in the BGP topology.

Once a session is established, BGP uses UPDATE messages to propagate routing information. These messages carry network prefixes that identify specific address ranges, alongside various attributes that influence routing decisions. Among these attributes, the AS_PATH is of paramount importance. It records the sequence of ASNs a route has traversed, ensuring that no routing loops occur by allowing routers to reject routes that attempt to re-enter an AS already listed in the path. Other critical attributes include the NEXT_HOP, which specifies the IP address of the next router along the path, and the LOCAL_PREF, which allows operators to set preferences for routes within their own network. The MULTI_EXIT_DISCRIMINATOR (commonly abbreviated as MED) provides a way to rank multiple entry points into an AS, enabling routers to make informed choices about which path to use when entering a network.

The process of selecting the best path in BGP is governed by a hierarchical algorithm. This algorithm evaluates multiple criteria in a specific order, starting with LOCAL_PREF, which ensures that internal policies take precedence. If this criterion does not yield a decision, BGP considers the AS_PATH length, favoring shorter paths to minimize hops and latency. When paths are otherwise equivalent, the MED value is compared, with lower values being preferred. If no distinction can be made at this point, additional factors such as the origin type of the route and the internal IGP (Interior Gateway Protocol) metric to the next-hop are examined. This multi-step evaluation ensures that BGP consistently selects the most efficient route for data packet delivery.

In a data center, BGP's configuration is often optimized to enable multipath routing, a feature essential for distributing traffic evenly across the multiple equal-cost paths available in Clos or Fat Tree topologies. This involves enabling settings like bestpath as-path multipath-relax, which modifies the path selection process to allow the use of routes with identical AS_PATH lengths but differing ASNs. Such configurations are crucial for ensuring high throughput and fault tolerance. Additionally, eBGP, a variant of BGP, is commonly employed within data centers due to its simplicity and ability to operate efficiently within a single administrative domain. The use of private ASNs further streamlines operations, reducing complexity and preventing issues such as path hunting, where routers repeatedly attempt to rediscover routes after link failures.

Rapid convergence is another critical aspect of BGP in data centers. This is achieved by tuning parameters such as the KEEPALIVE timer, which specifies how often routers exchange keepalive messages to verify connectivity, and the HOLD timer, which defines the duration within which a connection is considered active. Reducing these timers to values like three seconds for KEEPALIVE and nine seconds for HOLD ensures that routers quickly detect and respond to link failures, maintaining network stability and performance.

Equal-Cost Multi-Path (ECMP) is a routing technique designed to maximize network efficiency by utilizing multiple paths with identical cost metrics. In the dense connectivity of Clos and Fat Tree architectures, ECMP plays a pivotal role by ensuring that traffic is evenly distributed across the available links. This distribution minimizes congestion, improves overall bandwidth utilization, and enhances network resilience by providing alternate paths in the event of link failures.

The operation of ECMP revolves around intelligently distributing traffic among the equal-cost paths. One of the most effective methods to achieve this is through hashing, where a mathematical function maps packets to specific paths based on header fields such as source and destination IP addresses, port numbers, and protocol type. Hashing ensures that all packets belonging to the same flow follow the same path, preserving packet order and preventing reordering issues. Another approach involves region splitting, which divides the IP address space into subranges and assigns each range to a different path. While this method provides granular control over traffic allocation, it risks increasing the size of routing tables, potentially straining network resources. Simplistic methods like round-robin or randomization may also be used, but they lack the precision and consistency required for high-performance data center networks precisely due to their inability to account for the unique characteristics of each flow, such as bandwidth requirements, flow duration, or sensitivity to packet reordering. Round-robin, while evenly distributing flows in a sequential manner, does not consider the size or priority of each flow, potentially leading to uneven utilization of paths where large flows overburden specific links while others remain underutilized. Similarly, randomization introduces variability in path selection, which can result in traffic imbalances and increased latency due to unpredictable link congestion. Both methods also struggle with maintaining packet order across flows, a critical requirement in many applications, as packets taking different paths may experience differing latencies, leading to out-of-order delivery. These shortcomings make such simplistic approaches inadequate for the high-traffic, low-latency environments typical of modern data centers, where precision and efficiency are paramount.

When combined with BGP, ECMP unlocks even greater potential in data centers. BGP advertises multiple equal-cost routes for a single prefix, allowing ECMP to distribute traffic seamlessly across all available paths. This integration ensures balanced traffic flow between layers of a Clos network, from leaf to spine and beyond. Moreover, ECMP enhances fault tolerance by quickly rerouting traffic in the event of a path failure, maintaining uninterrupted connectivity. By utilizing the full bandwidth capacity of the network, ECMP enables data centers to scale effectively, supporting the demands of modern applications without compromising performance or reliability.

2 Technology and Tools

2.1 FRRouting / Docker (Jose)

FRRouting is a free and open-source Internet routing protocol suite for Linux platforms that enables us to implement BGP in our emulator. In addition to FRRouting, we use Docker, an open-source platform for containerizing applications. Specifically, it lets us containerize network nodes to isolate environments for switches and servers. In our emulator, FRR is used within the Docker containers to provide the BGP routing functionality that is necessary for the switches in the network. For each switch, a container is created with the frrouting/frr:latest image with its automatically generated configurations based on the connections and the specific switch. These configurations include frr.conf and daemon files which define interface settings, prefix lists, BGP neighbors, and route maps. The Docker containers set the privileged flag to true and with NET_ADMIN and SYS_ADMIN to allow for low-level networking operations, such as managing network interfaces and routing. These files are then mounted to /etc/frr to set up the routing stack. In this setup, containers are linked through established veth pairs with IP addresses that are configured to align with the FRRouting setup. Additionally, Docker's isolated environments ensure independence for all nodes while allowing smooth communication across the emulator.

2.2 Linux Network Namespaces (Mohammad)

Network namespaces are a fundamental part of the Linux kernel that allows us to have isolated network stacks within a single operating system. Each namespace will maintain its own routing table, firewall rules, interfaces, statuses, etc. enabling us to create completely isolated network environments. This sort of isolation is important for network virtualization and forms the foundation of the networking in datacenter network emulators. Our implementation leverages these namespaces indirectly through docker containers, where each container will get its own network namespace that is identified by its Process ID (PID). Namespaces are not explicitly created, the PIDs of running containers are retrieved and we use these to reference their associated namespaces when establishing connections. The emulator creates network links between containers by generating a virtual ethernet pair (veth) between two containers. These interfaces are then moved into their respective container namespaces using the container PIDs, followed by IP address configuration through commands directly executed on the containers. This approach combines Docker's container isolation with direct namespace manipulation, allowing precise control over the network topology while maintaining separation between nodes.

3 Technical Implementation

3.1 High Level Architecture (Mohammad)

The fat tree network emulator implements a complete datacenter network topology through a modular, object-oriented architecture centered around three core classes, Node, Pod, and the FatTree. The Node class serves as the foundation providing networking capabilities like connection management and virtual ethernet (veth) creation. It has two subclasses of Switch and Server to represent the two different kinds of nodes that we have within our datacenter and implement node specific behaviors. This inheritance-based design allows for

clean separation of concerns, with the switches handling BGP routing configuration and servers managing end-host networking, all while sharing common connection management code through their parent class. The Pod class is a wrapper around a grouping of Servers and Switches that is tied to the physical concept of a pod within a fat tree topology and allows for a pod to be connected internally. This enables an easy to understand hierarchical structure within the code. The fat tree class orchestrates the entire topology, working as the central controller that manages topology generation, IP address allocation, configuration deployment, and network validation using a Pingmesh test. It takes a systematic approach to building that first generates and connects nodes, creates addressing and routing configurations, and materializes the network using Docker containers and veth connections. This separation between the logical topology generation and the physical topology generation provides unique flexibility between configurations while ensuring consistent deployment.

3.2 Generation (Mohammad)

The generation phase constructs the logical structure of the fat tree topology through a hierarchical process defined by the intrinsic properties of the topology and a user-defined parameter k. The system creates k2/2 core switches, followed by k pods containing k/2 aggregation switches and k/2 edge switches each, with k/2 servers per edge switch. During creation, each switch is assigned a unique Autonomous System Number (ASN) starting from 65000, incrementing monotonically to ensure distinct BGP domains. The process establishes bidirectional connections by registering node pairs: core switches core switches connect to one aggregation switch in each pod, aggregation switches link to all edge switches within their pod, and edge switches connect to their designated servers. Note that this phase will only handle the logical relationships, creating a graph representation that serves as the foundation for subsequent network configuration steps.

3.3 Configuration (Mohammad)

The configuration phase handles the network addressing scheme and routing configuration for all nodes in the fat tree topology. The system implements IP addressing distribution using /30 subnets from the 172.16.0.0/16 address space, allocating unique address pairs to each point-to-point connection between nodes. For routing configuration, the system generates Free Range Routing (FRR) configurations for each switch, implementing BGP peering relationships based on the assigned ASNs. The FRR configuration includes interface definitions, prefix lists for local network advertisement, and BGP neighbor relationships with appropriate route maps. Each switch's configuration is written to a dedicated directory that will be mounted into its respective container, with both a frr.conf file containing the routing protocol configuration and a daemons file enabling the necessary FRR daemons.

3.4 Containerization (John)

The containerization phase of our datacenter network emulator uses Docker to encapsulate each network node (core, aggregation, and edge switches, as well as servers) into isolated containers, ensuring modularity and operational independence. Each node type is tailored to a specific role, with switches using Free Range Routing (FRR) images for advanced

routing protocol support, such as BGP, and servers using Ubuntu-based images optimized for diagnostics and testing. The creation of containers is fully automated, with a structured process that first generates configuration files specifying node-specific routing protocols, IP addresses, and connectivity details. These configurations are dynamically created and mounted into containers at runtime to set up routing and connectivity while also ensuring precise and reproducible setups. The containerized topology is structured using virtual Ethernet (veth) pairs, which establish realistic point-to-point links while maintaining strict isolation between containers. This ensures that each node operates independently while simulating accurate physical connections. The structure of the network follows the Fat Tree design principles, where core switches are connected to aggregation switches in each pod, and edge switches are connected to servers. This hierarchical network structure is achieved by first generating the core switches, followed by the creation of pods containing aggregation and edge switches, and finally connecting these pods to the core switches. The connections are made to ensure proper distribution and communication within the network, mimicking a real-world data center design.

3.5 Visualization (Sam/John)

To visualize the Fat Tree network, we employed a combination of Python libraries (NetworkX, Graphviz, and Plotly) each serving a distinct purpose in the process. NetworkX was central to modeling the network's structure and topology. By treating the network as a graph, we represented switches and servers as nodes, and their connections as edges. This library provided robust functionality for creating, manipulating, and analyzing the topology while allowing us to assign properties like labels and colors to individual nodes. Once the network topology was defined with NetworkX, we used Graphviz to generate static visualizations in the form of PNG images. With its powerful graph layout algorithms, Graphviz ensured that the nodes and edges were arranged clearly and aesthetically, making the structure easy to interpret. These static images allowed users to visually inspect the Fat Tree topology and better understand the connections between nodes. For dynamic exploration, we turned to Plotly, leveraging its interactive visualization capabilities. This allowed users to zoom, pan, and hover over nodes and edges to reveal additional details such as IP addresses and node types. The resulting interactive graphs offered an engaging way to examine the network, making it possible to dynamically explore complex topologies. After generating the PNG files using Graphviz and the interactive graph with Plotly, we integrated these methods into a website built with Flask and SocketIO. This website provides a graphical interface for users to emulate the Fat Tree topology, allowing them to visually explore and interact with the network. Through this interface, users can create, view, and manipulate different configurations of the network up to $k = 70$, providing a powerful tool for visualizing complex network topologies in real-time.

3.6 Pingmesh (Sam/John)

After having established the containerized fat tree topology and the visualization of the network, the Pingmesh and Traceroute functionalities are employed to perform comprehensive diagnostics, ensuring that the network operates as intended. Pingmesh is used to test the connectivity between all servers in the network by initiating pings from each server to every other server. This generates a "mesh" of connectivity data that reveals any issues such as packet loss or unexpected latency. By pinging each server from others

within the Fat Tree network, the Pingmesh ensures that the virtual Ethernet (veth) pairs connecting the containers are functioning correctly. This is especially important in a containerized setup, where connectivity between containers might not be immediately obvious. Pingmesh helps validate the proper configuration of the network topology by ensuring that each node can reach all others, which is a critical part of confirming the functionality of the containerized network. Traceroute goes a step further by tracing the path that packets take from one server to another. For each pair of source and destination servers, Traceroute reveals the exact route through the network, showing the intermediate hops and providing round-trip time (RTT) information for each hop. This is crucial for identifying routing anomalies within the containerized network, such as misrouted packets or unexpected delays in specific parts of the network. Since the Fat Tree topology involves multiple layers of switches and virtual links, Traceroute helps confirm that packets are taking the correct paths through core, aggregation, and edge switches. For example, Traceroute ensures that a server in one pod is correctly reaching a server in another pod via the core switches, without unintended detours or delays. If there are any irregularities such as prolonged RTTs or incorrect hop sequences Traceroute helps pinpoint where the issue lies, such as in the configuration of a switch or the routing tables within a container. This level of diagnostic detail is essential in a containerized network where the complexity of virtualized connections may mask underlying issues that would otherwise be easy to overlook.

4 Results (Sam)

Our datacenter network emulator project successfully achieved its core objectives, delivering a robust emulation platform for Fat Tree topologies in datacenter networks. The emulator supported the generation of topologies up to 70, representing up to 42,875 servers interconnected through 1,225 core switches and various aggregation and edge switches. By leveraging Docker for containerization and Free Range Routing (FRR) for BGP configuration, the system provided a modular and scalable solution for emulating real-world datacenter networks.

Our implementation included automated IP address assignment, generation of BGP configuration files, and seamless deployment of Docker containers for each node in the network. Point-to-point links were established using virtual Ethernet (veth) pairs, ensuring network isolation and realistic emulation of datacenter traffic flows. The system also incorporated diagnostics tools such as Pingmesh, which evaluated connectivity and latency between servers, and Traceroute, which traced packet paths through the network.

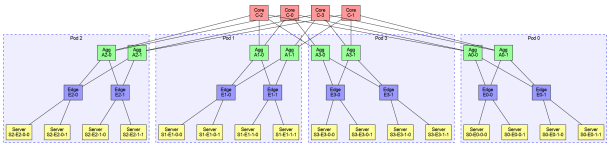

Figure 1. An image showing the fat-tree topology generated from $k = 4$

Visualization tools played a key role in our results, with NetworkX, Plotly, and Graphviz enabling both static and interactive representations of the network. Users could observe the layered structure of the Fat Tree topology, with color-coded nodes (Core: Red, Aggregation: Green, Edge: Blue, Server: Yellow) and dynamically adjusted spacing for scalability.

The project concluded with the development of a web-based interface using Flask and SocketIO. This interface allowed users to generate network topologies interactively, visualize them in real-time, and perform network diagnostics directly from a browser. The platform not only met its intended deliverables but also provided a solid foundation for future expansions, including the integration of additional routing protocols and alternative network topologies.

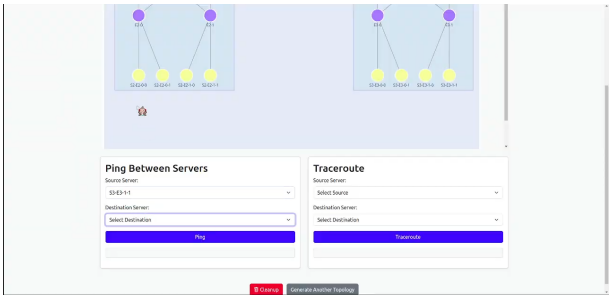

Figure 2. An image showing the final result of the interactive interface

5 Contributions

1. **Jose Cruz**: I worked on building the initial basic fat-tree class that would be expanded on. This includes creating the nodes and connecting them. I helped implement visualization through Graphviz for the fat-tree and worked on the advanced topology prior to the fat-tree.
2. **Mohammad Durrani**: I worked on the design and a large portion of the implementation of the backend for the network emulator, the bash scripts for the initial network prototypes, and did the bring up for docker-compose to get frr-routing working.
3. **Cheng-Yuan (Sam) Lee**: I worked on the website visualization & interaction with Flask & SocketIO. Modified the main script to send messages from each docker container to the main code. Implemented methods in the website to allow pinging/tracerouting between selected servers.
4. **Andrew Liu**: I worked on the initial emulation of the network by representing routers and servers with namespace. I later tested the backend scripts for the network emulation, as well as the website, to identify and debug/alert any errors that occurred. I also worked on the slides for each of our milestone presentations.
5. **John Shim**: I worked on parts of the initial design of the fat tree network and tested it to see if it worked properly. I then tested scripts of the network backend along with the website visualization to look out for any bugs and find room for improvement. I also worked on the slides on the background of our project including data center networking, fat tree topology, and routing protocols such as BGP and ECMP and some on containerization and visualization of our fat tree network topology in our presentation.