Vrije Universiteit Amsterdam

Bachelor Thesis

OpenDCN: Designing and Developing a Unified Framework for Compute-Network Co-Simulation with Trace-Based, Interactive, and Traffic-Pattern Network Simulation

---

Author: Alessio Leonardo Tomei (2777731)

1st supervisor: Jesse Donkervliet
2nd reader: Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science.

June 8, 2026

Abstract

Dataceners are becoming of ingrasing importance in modern society, poering cloud services, AI workloads, and large-scale data processing. For research and planning there is an increasing need for accurate and comprehensive simulation tools. Networking within datacenters has been largely overlooked in favor of compute analysis, despite networking contributing to a significant amount of total datacenter energy expenditure, and the significant impact that compute-network interdependencies can have on system behavior. We introduce OpenDCN (ODCN), an extension of OpenDC (ODC), a peer-reviewed, state-of-the-art datacenter compute simulator. ODCN adds support for datacenter network simulation, designed to serve research, academic, and industry while enabling integrated compute-network co-simulation for more accurate and comprehensive system analysis. The framework supports a broad range of customizable network topologies, both trace-driven and synthetic traffic-pattern-based workloads, modeling of network protocols and policies, and interactive simulation features for educational use. OpenDCN adopts a high-level, flow-based abstraction level, which aligns well the more coarse-grained nature of compute simulation. Through a series of case studies, we demonstrate the utility of ODCN in analyzing performance trade-offs and inefficiency that a compute-only model is unable to detect.

Contents

1
Introduction

The strain on network infrastructures is escalating due to a rapid increase in internet traffic, driven by factors such as the rise of high-definition content, the proliferation of connected devices, and the growth of data-intensive applications. Each day, over 400 exabytes of data are generated, and in 2025, this number is projected to surge to almost 500 exabytes per day (≈23% increase) [4, 51]. Nokia's *Global Network Traffic Report* [3] predicts that global network traffic demand will reach between 2,443 to 3,109 exabytes per month in 2030, with a *compounded annual growth rate* (CAGR) up to 32%. Datacenters have become indispensable in order to support such an increase in online traffic, being central to the operations of virtually every online service. Moreover, the continuously accelerating demand for computing power has also driven significant expansion of datacenter infrastructure [45]. Energy consumption and carbon footprint of such expanding facilities have come under scrutiny [18, 42, 59]. This has led to increased focus on assessing their environmental impact.

Simulation plays a key role in enabling analysis and comparison of datacenter technologies at scale [44]. Importantly, simulation-based evaluation drastically reduces the environmental impact compared to real-world experimentation. Recent studies estimate energy consumption savings on the order of 1:116 billion [29]. Although many datacenter simulators are available, they frequently omit the networking component, a critical factor contributing significantly to both energy consumption [17] and overall system functionality. We argue that accurately modeling both compute and network components, along with their complex interdependencies, is essential for producing more reliable and precise simulation results. In this work, we develop OpenDCN, a state-of-the-art, flexible network simulator that supports both traditional workload-based simulations and an innovative interactive simulation environment, distinguishing it from existing tools. We implement OpenDCN as an extension of OpenDC [30, 44], a peer-reviewed datacenter compute simulator. We then perform experiments using combined compute-network simulation to assess differences compared to conventional compute-only approaches.

1.1 Problem Statement

Despite accounting for about 20% of the total datacenter energy consumption [17], datacenter's networking is often overlooked in the field. Evaluation methods, such as benchmarking, require substantial time and financial investment and offer limited scalability. Additionally, it is advantageous to predict the performance and energy impact of a given configuration prior to its deployment,

reducing the risk of non-optimal performance and ensuring that the system meets expectations. Datacenter simulators have become essential tools in the design, scaling, and development of datacenter infrastructures. Numerous simulation frameworks have been proposed and adopted to support these objectives [14, 13, 30, 31, 47]. Simulation offers a cost-effective and scalable alternative to physical experimentation, enabling stakeholders to evaluate ICT infrastructure performance, reliability, and energy consumption under diverse scenarios. *Mastenbroek et al.* report an energy efficiency ratio of approximately 1:116,000,000,000 when comparing simulation-based experimentation to equivalent real-world deployments [29].

Numerous generic network-only simulators have been developed, each operating at varying levels of abstraction. Some simulators, such as ns-3 [28] and Mininet [39], provide fine-grained, low-level representations of network traffic. Others, like SimGrid [14], adopt a higher-level abstraction, focusing on scalability and task scheduling. Public datacenter traces are often limited in scope and lack essential data for comprehensive simulation. Researchers frequently require the ability to isolate compute or network behavior. A unified simulation framework that supports both integrated and decoupled modeling, while maintaining consistent abstraction, would significantly enhance flexibility and applicability. CloudSim [13] offers partial compute-network integration but models their interactions in a highly abstract and loosely coupled manner. It also lacks support for standalone network simulations, limiting its use for detailed or isolated network analyses. Therefore we raise the following Problem Statement (**PS**):

PS1 Absence of a unified tool supporting decoupled simulation of compute and network workloads.

Beyond the need for decoupled modeling, some evaluations benefit from *combined* compute–network modeling. Coarse bandwidth–delay abstractions can be inaccurate, and can bias energy attribution between compute and network. Existing frameworks attempt to address combined simulation. CloudSim [13], ICanCloud [47], EdgeCloudSim [58] have coarse-grained compute–network co-simulation. GreenCloud [37] integrates packet-level networking with compute and energy but targets smaller scales. SimGrid [14] provides scalable communication abstractions without detailed datacenter network semantics, while ns-3 [28] and OMNeT++ [61] have packet level network models without compute. Integrated, cross-layer co-simulation, while remaining practical at datacenter scale, is still limited. This raises **PS2**:

PS2 Lack of integrated, cross-layer compute–network co-simulation frameworks at datacenter scale.

There is growing interest in *interactive* network simulation that supports live control of a running experiment, such as topology editing, fault injection, traffic control, and state inspection) [40]. Such capabilities benefit education [32], research (rapid prototyping and debugging), and development.

Existing network and datacenter simulators try to address these requirements but often provide limited runtime interactivity or adopt an abstraction level not suitable studies at scale. For example, ns-3 [2], packet-level modeled, has limited interactivity, requiring pre-scheduled events or custom code. OMNeT++ [61] provides time control and state inspection, but has limited live editing capabilities of topologies and traffic. Moreover, these tools operate at low levels of abstraction (packet-level or emulation), which makes very large-scale or long-trace studies computationally expensive. This motivates **PS3**:

PS3 Limited availability of open-source, mid-level-abstraction simulators with interactive capabilities.

1.2 Research Questions

The problem statements listed in Section 1.1 lead to the Main Research Question (**MRQ**) and we break it into three subquestions (**RQ1** to **RQ3**):

MRQ *How should mid-to-high level network and compute–network co-simulation be modeled in datacenters, and how does network modeling affect conclusions on performance, energy, and task-level QoS relative to compute-only baselines?*

RQ1 How to model network resources at a high-level-abstraction across datacenters and generic network topologies, enabling trace-based, traffic-pattern-based and interactive network simulation?

Our goal is to model network resources and behavior at a higher level of abstraction than most existing simulators, so that large-scale systems can be simulated over prolonged periods of time. Supporting trace-based, traffic-pattern, and interactive simulations within a single tool enables complementary approaches and equally important applications to coexist under one framework. Capturing fine-grained network interactions in a representative high-level model poses challenges for both fidelity and scalability Chapter 3. Achieving this across all three modes further necessitates a general-purpose, well-structured, and highly extensible simulation architecture.

RQ2 How can a compute-centric simulator be extended with a network layer to support both joint compute–network co-simulation and standalone network and compute studies?

Achieving compute-network integration allows for comprehensive evaluation of datacenter infrastructures, considering both computational and networking components (Section 4.7).One of the main challenges is the incorporation of a network simulation module with a compute module. The resulting framework should still support standalone network simulation, experimentation, and evaluation.

RQ3 How does datacenter networking affect the performance and overall system behavior on datacenter workloads?

This study aims to evaluate the benefits of considering network interactions in datacenter simulation through a series of case studies (Section 5.4). Our aim is to assess changes in scheduling latency, execution and completion times, and related system-level outcomes. The findings have the potential to reshape current practices in datacenter simulation. We know that networking can have significant impact on system performance, it is often simplified or omitted in datacenter models, as mentioned in Section 1.1, and opportunities to perform such combined analysis remain limited at the moment. A practical challenge is that most public traces lack explicit networking/topology information. We therefore consider representative topology/routing assumptions and discuss their implications for the results.

1.3 Contributions

To address the research questions outlined in Section 1.2, we present the following main contributions:

C1 We introduce a state-of-the-art simulator, to simulate event-driven large-scale datacenter network workloads, designed to support the modeling and simulation of arbitrary network sce-

narios.

The simulator, developed following modern engineering practices, offers advanced features, such as support for various topologies, energy models, routing protocols and QoS policies. Our tool supports different modes of network simulation, including trace-driven (Section 4.4), traffic-pattern (Section 4.6), and interactive (Section 4.5) simulations. The tool is designed to promote modularity and extensibility, facilitating addition, sharing, and reuse of components, experiments, and policies. These design choices promote reproducibility of results, ease of adoption, and applicability across a broad range of use cases, including research, teaching, and practical experimentation.

C2 We introduce an innovative approach for simulating network scenarios via an interactive REPL-based simulation environment (Section 4.5).

The simulator provides an interactive REPL-based interface that offers fine-grained control over simulation events, enabling users to build, test, import, and export components, policies, and scenarios dynamically. This environment is especially valuable for educational purposes and for iterative testing and development of new components.

C3 We present OpenDC(N) as a pioneering datacenter simulator capable of compute-network combined simulation.

Integrating OpenDCN (network) with OpenDC (compute), a widely adopted datacenter compute simulator that has been actively developed and validated over more than seven years [44, 30] we present a tool that is capable of more comprehensive datacenter simulations that combine both compute and network components for higher results accuracy.

C4 We empirically demonstrate the impact of network modeling on datacenter performance, energy, and QoS, through a series of reproducible case studies.

We demonstrate the impact of modeling network dependencies through three experiment sets: traffic-pattern validation against BookSim (Section 5.2), trace-driven topology/routing studies (Section 5.3), and compute–network co-simulation of datacenter workloads (Section 5.4).We quantify effects on accepted rate and runtime scalability, energy attribution , and task-level QoS (delays and slowdowns), showing that network-aware modeling can materially change conclusions for comprehensive datacenter simulations, especially in the tail. To enable replication and extension, we provide experimental artifacts and configurations.

2
Background

We now introduce concepts and terminology necessary to understand later sections. In Section 2.1, we outline the architecture of a conventional data center network. In Section 2.2 we list widely used network topologies, presenting their taxonomy. Section 2.3 provides an overview of routing, including its possible categorizations and enlisting the most commonly used in practice. Section 2.4 discusses a set traffic patterns used in network simulation to assess routing performance under different topologies and routing protocols. Finally, we enlist performace metrics relevant to later experiments and specifically DCNs (Section 2.5).

2.1 Datacenter: Role and Traditional Architecture

A datacenter (DC) is a facility containing a network of computing and storage resources used to support the delivery of digital services. These facilities are the backbone of modern information and communication technologies (ICT), hosting websites, cloud applications etc. Datacenters consist of interconnected servers, storage systems, and networking equipment, to ensure scalability, performance and efficiency.

In recent years, the role of datacenters is of increasing importance due to the exponential growth of internet traffic, mobile devices, and cloud computing platforms. Demand for digital services continues to rise across sectors such as e-commerce, video streaming, finance, and scientific research, datacenters have become essential for ensuring reliable access to computational power and data storage. Both research and industry have increasing interest in optimizing performance and energy efficiency of datacenters.

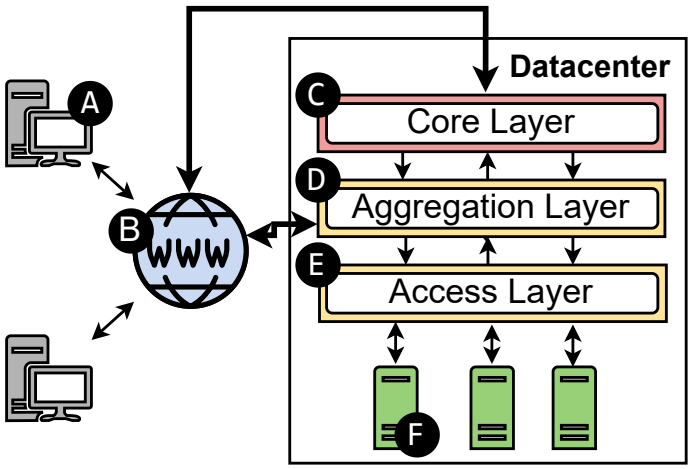


Figure 2.1: Traditional Datacenter Design Overview.

Figure 2.1 illustrates a traditional three-tier datacenter architecture, commonly used to explain the functional components and the data flow from end-user to hosted services. The *user* (A) performs service requests, which traverse the *Internet* (B) and eventually enter the datacenter through edge routers.. Once within the datacenter boundary, traffic is often handled by the *core layer* (C), which acts as the backbone of the datacenter network. This layer is optimized for throughput, fault tolerance, and minimal latency, and typically utilizes high-performance switches operating at 10 Gigabit Ethernet or higher [27]. The *aggregation layer* (D) (also referred to as the *distribution layer*) is under the core layer, and is important in managing east-west traffic [11] within the datacenter and serves as a bridge between the access and core layers. In some designs, gateway functionality is implemented at the aggregation layer. The *access layer* (E) is the point at which the network physically connects to compute and storage. This layer is designed to connect network and *hosts* (F), the physical machines that provide computing power. Each access switch usually connects 10-20 different servers with 1GE ports [27, 46]. The hosts themselves are physical servers with virtualization, enabling the creation of multiple isolated execution environment [8]. These environments may consist of Virtual Machines (VMs), which are capable of running independent applications or services [53].

It is important to note that this three-tier architecture represents a generalized and conventional topology. In practice, completely different topologies are in use, with ongoing research focusing on novel topologies that improve scalability, fault tolerance, and energy efficiency. Alternative datacenter topologies and their characterization is presented in the following section.

2.2 Network Topologies and Classifications

DCNs topologies have evolved significantly to attempt to scale at the increasing demand. We now highlight the importance of a DCN simulator capable of building and simulating arbitrary topologies, enabling researchers and practitioners to conduct evaluation and optimization of both established and emerging architectures.

Name(s)	Abbrv.	Description
Nodes (Vertices)	V	Total number of nodes in the (datacenter) network.
Hosts (Terminals)	N	Number of host machines in the (datacenter) network.
Switches (Routers)	R	Number of switches that forward traffic in the (datacenter) network.
Links (Edges)	E	Number of links interconnecting the nodes.

Table 2.1: Terminology and abbreviations of DCN parameters used throughout this work.

2.2.1 Topology Terminology and Abbreviations

We define a set of abbreviations representing the main characteristics of DCN topologies. Table 2.1 enlists these abbreviations, which are the same in all topologies. In theoretical models, the entities of a network are described as vertices, edges, with vertices subclassing in routers and terminals. In computer practice, the same notions are referred to as nodes (including switches and hosts) and links. To avoid ambiguity, we present both conventions side by side. Additional abbreviations for parameters specific to particular topologies will be introduced in later sections.

2.2.2 Structural Pattern Classification

Modern DCN topologies can be broadly categorized by their structural patterns. This section surveys the major classes of topologies found in contemporary and experimental datacenter networks.

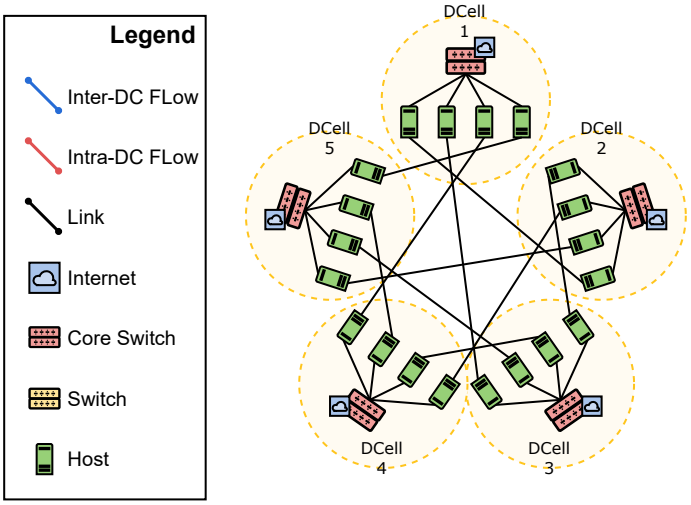


Figure 2.2: DCell Topology.

Recursive-defined. Recursive defined topologies (DCell [25] (shown in Figure 2.2), BCube [24], FiConn [41], introduce partially *server-centric* use recursive rules in order to build scalable and modular networks. These kinds of topologies are usually *hybrid* topologies, introducing server-centricity, where servers act as both computation endpoints and intermediate forwarding nodes.

Flattened. Flattened topologies such as Dragonfly [36], Slimfly [12], PolarFly [38], Flattened Butterfly [35], focus on minimizing hierarchy, reducing network diameter.They are typically *switch-centric* and rely on complicated group connections strategies to balance throughput, cost, and path diversity.

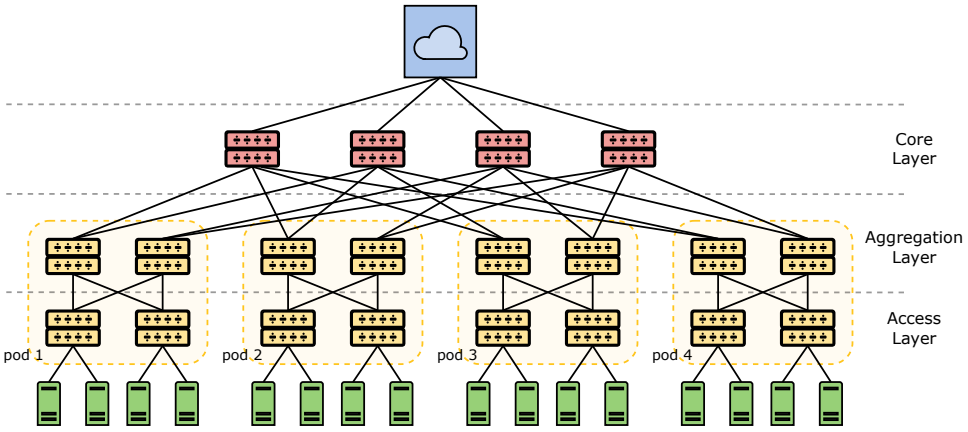


Figure 2.3: FatTree with n=4.

Random. Random topologies, such as Jellyfish [57], build networks using random or expander graph principles. Their non-deterministic nature enables incremental scalability, but often suffer from high performance variability.

Grid-like. Grid-like topologies, such as 2D/3D Mesh and Torus [16], HyperCube [48], connect nodes in regular, multi-dimensional geometric patterns. These topologies are often used in environments due to their low-latency characteristics for local traffic.

Hierarchical. Hierarchical topologies (VL2 [23], two-tier leaf-spine topology [7]) are widely used in practice, with the servers as leaf nodes and switches as intermediate nodes. *Fat-Tree* is one of the most famous examples, illustrated in Figure 2.3 and introduced by *Al-Fares et al.* [6]. Fat-Tree employs multiple roots enhancing fault tolerance and bisection bandwidth. These are all examples of *switch-centric* architectures.

2.2.3 Functional Classification

DCN topologies can also be classified by which components, handle routing and packet transmission.

Switch-centric. Routing and packet forwarding responsibilities are delegated entirely to network switches. Common examples include traditional hierarchical and flatten topologies.

Server-centric. Server-centric architectures, also called *direct networks*,leverage servers for routing. In these designs, servers are connected directly to each other, which can reduce deployment costs by reducing number of switches. This adds computational and energy overhead on the hosts, as servers must remain active. These architectures can complicate power management strategies and often exhibit a less predictable worst-case latencies. Examples include CamCube [5] and NovaCube [62].

Hybrid. Hybrid architectures combine elements of both models: servers and switches collaboratively forward packets.

2.3 Routing in Data Center Network

Routing protocols determine how traffic is directed across the network. These protocols can have a significant impact on throughput, load balance and tail latency. Some determine paths statically,

other use node-local or global state information (e.g., SDN controller). We categorize protocols by the decision scope in Section 2.3.1, and enumerating the most used protocols in Section 2.3.2.

2.3.1 Categorization

In this Section, we present a high level categorization of routing strategies based on where and how routing decisions are made within the network.

Static routing. Routing paths are determined statically and do not depend on any state information. These paths remain fixed, without adapting to live changes, presenting minimal overhead but lacking flexibility to respond to diverse traffic patterns.

Locally adaptive routing. Locally adaptive routing dynamically adjusts routing decisions based on real-time information available on the node, such as queue occupancy or link utilization. These protocols operate in a distributed manner and aim to reduce congestion by rerouting flows at individual switches. However, since decisions are made without global visibility, local adaptations can lead to suboptimal global outcomes.

Globally adaptive routing. Globally adaptive routing makes decision based on network global state, including end-to-end traffic demand, topology information, or link-level congestion. This approach is characteristic of centralized control planes in *Software-Defined Networking* (SDN). While it enables better load balancing and congestion mitigation, it may also introduce control plane complexity and reduced responsiveness.

2.3.2 Common Routing Protocols in Research and Practice

In this section, we present an overview of widely used routing protocols in both research and practice, and will be used in Chapter 5. The protocols are described using abstract router notation, without distinguishing between hosts and switches, in line with common practices in theoretical network research. Throughout this discussion, we use R_s to denote the source router and R_d to denote the destination router.

Open Shortest Path First (OSPF or MIN). OSPF, also called *minimal static routing* (MIN) is a static routing protocol that routs a packet from R_s over the minimal path to its destination router R_d.

Equal Cost Multi Path (ECMP or RAND MIN). ECMP is a routing strategy that combines MIN with randomization (*RAND MIN*) or deterministic of multiple minimal equal cost paths. Specifically, router R_s selects one of the minimal-hop paths to R_d. Common selection implementations use hash-based selection on packet headers, round-robin, or uniform random choice to achieve static load balancing across the paths.

Valiant Routing (VAL). VAL, proposed by *Valiant et al.* [60], introduces randomized path diversity to reduce network congestion. For each packet, a random intermediate router R_i is selected, such that $R_i \neq R_s$ and $R_i \neq R_d$. MIN routing is then used to rout the packet from R_s to R_i, and subsequently from R_i to R_d. VAL helps to avoid potential hotspots in the network, but may reduce the effective bandwidth and increase latency due to the longer paths that are used.

Universal Globally-Adaptive Load-balanced (UGAL). UGAL, introduced by *Kim et al.* [36] in the context of Dragonfly topology, is designed to balance traffic across global channels, those connecting different groups of nodes. For each packet, the protocol chooses between MIN and VAL paths to load-balance the network. UGAL is separated in a local version *UGAL-L*, which uses local

queue information at the current router node, and *UGAL-G*, which uses queue information for all the global channels.

While the protocols discussed here are widely used, many other exist, including topology-specific variants of ones mentioned above.

2.4 Traffic Patterns

For network evaluation, evaluating how it responds under a variety of traffic patterns is required. Researchers commonly use a set of canonical traffic patterns that abstract different real-world scenarios while stressing various aspects of the network. These patterns are designed to expose bottlenecks, test load balancing, and evaluate routing strategies under both benign and adversarial conditions. In this section, we describe several widely adopted traffic patterns, that will be used in 5.2.

Uniform Random. In this pattern, each packet's source selects a destination uniformly at random among all other nodes.

Random Permutation. A permutation of source destination mapping is selected uniformly at random from the set of all possible permutations. Each node sends traffic to exactly one destination and receives from exactly one source (one-to-one mapping).

Bit-Permutation. Bit-permutation patterns map each source address to a destination by applying a deterministic transformation to its binary representation. Common examples are *bit-complement*, *bit-reversal*, *bit-transpose* and *bit-shuffle*. We define these in Equations (2.1) to (2.4).

$$d_i = \overline{s_i}, \quad \text{for } i = 0, \ldots, n-1 \qquad \text{(bit-complement)} \tag{2.1}$$

$$d_i = s_{n-1-i}, \quad \text{for } i = 0, \ldots, n-1 \qquad \text{(bit-reversal)} \tag{2.2}$$

$$d = (s_{\frac{n}{2}-1}, \ldots, s_0,\ s_{n-1}, \ldots, s_{\frac{n}{2}}) \qquad \text{(bit-transpose)} \tag{2.3}$$

$$d_i = s_{(2i) \bmod n}, \quad \text{for } i = 0, \ldots, n-1 \qquad \text{(bit-shuffle)} \tag{2.4}$$

where s and d denote binary representations of the source and destination addresses respectively, and s_i and d_i refer to the i-th bit of s and d.

Tornado. In the Tornado pattern [56], each router i sends traffic to router $(i + \frac{N}{2}) \bmod N$, where N is the total number of routers. This often results in traffic being directed halfway across the network, especially in grid-like topologies,. Tornado is useful for exposing bottlenecks and testing the network's ability to handle structured, high-stress patterns.

Adversarial. Adversarial traffic patterns are intentionally designed to stress the network and reveal worst-case performance scenarios. They are topology specific and are often defined alongside new topology designs in research to highlight limitations.

2.5 Relevant Datacenter Evaluation Metrics

This Section outlines key metrics used to evaluate datacenter and network performance. These include energy efficiency indicators, network-specific throughput and power metrics, demand satisfaction measures and carbon emission.

2.5.1 Energy Effectiveness

Power Usage Effectiveness (PUE). A variety of metrics are currently employed to evaluate datacenters, but one particular metric has emerged as a predominant industry standard. PUE, introduced in 2006 [43], has become the most widely utilized measure for assessing datacenters energy efficiency [15, 34]. The PUE, defined in Equation (2.5), quantifies the proportion of energy utilized specifically by IT equipment relative to the total energy consumption of a datacenter.

$$PUE = \frac{P_T}{P_{IT}} \tag{2.5}$$

where P_T denotes the total power consumption of the datacenter, and P_{IT} represents the power consumed by the IT infrastructure.

Datacenter Performance Efficiency (DCPE). Derived from PUE, DCPE is a metric used to measure the computational efficiency of datacenters. DCPE was introduced by *Belady et al.* [9] and used to capture the fraction of energy used for computation.

$$DCPE = \frac{U_{IT}}{PUE} = \frac{U_{IT} \cdot P_{IT}}{P_T} \tag{2.6}$$

where U_{IT} denotes the IT Equipment Utilization.

2.5.2 Network Specific Metrics

We proceed by presenting relevant network metrics that will be used throughout evaluation and experimentation (Chapter 5).

Network Power Usage Effectiveness (NPUE). In this work we also use a variation of PUE, namely *Network Power Usage Effectiveness* (NPUE), introduced by *Popoola et al.* [49]. This metric represents the ratio of overall IT power to power utilized by the network modules, defined in Equation (2.7).

$$NPUE = \frac{P_{IT}}{P_N} \tag{2.7}$$

where P_N denotes the power consumption of the network infrastructure.

Network Power Effectiveness (NPE). Despite very similar names, NPE and NPUE are very different metrics. NPE, introduced by *Shang et al.* in 2015 [52], is defined as the ratio between the aggregate network throughput and the total network power consumption (Equation (2.8)). This metric quantifies the end-to-end efficiency of transmitting data. Consequently, NPE reflects the tradeoff between power consumption and network throughput in datacenters.

$$T^{\tau}_{\text{agg}} = \sum_{f \in \mathcal{F}} T^{\tau}_f \qquad\qquad NPE = \frac{T^{\tau}_{\text{aggr}}}{P^{\tau}_N} \tag{2.8}$$

where T^{τ}_{agg} is the total aggregate throughput of all flows $f \in \mathcal{F}$ at time instant τ, T^{τ}_f denotes the throughput of flow f at time τ, and P^{τ}_N represents the power consumed by the network at that same instant.

Communication Network Energy Efficiency (CNEE). CNEE, is effectively the inverse of NPE and quantifies the efficiency with which the network converts electrical energy into information

transmission [21]. This metric measures how effectively the network performs the task of data delivery. Equation (2.9) defines CNEE for time instant τ:

$$CNEE = \frac{P_N^\tau}{T_{\text{aggr}}^\tau} \tag{2.9}$$

Network Latency. In the context of data center network evaluation, *network latency* is defined from the application's perspective, as articulated by *Guo et al.* [26]. Specifically, network latency refers to the elapsed time interval from the moment an application A on one server transmits a message until it is received by an application B on a peer server. While we recognize the importance of latency for network performance, our primary focus in this study is directed towards the analysis of network throughput, the assessment of demand satisfaction, the detection of congestion events, and the evaluation of energy consumption within the networking infrastructure. Additionally, we investigate how task dependencies on network resources might impact their completion times and overall system efficiency.

Demand Satisfaction Ratio (DSR). DSR quantifies the extent to which the bandwidth requirements of a network flow are fulfilled by the underlying infrastructure. It is defined as the ratio between the actual allocated throughput and the requested (demanded) throughput for a given flow. Formally, for a flow f, the DSR is given by:

$$DSR_f = \frac{T_f^{\text{alloc},\tau}}{T_f^{\text{req},\tau}} \tag{2.10}$$

where $T_f^{req,\tau}$ is the network demand of flow f at time instant τ, and $T_f^{alloc,\tau}$ is the actual throughput obtained by f at that same instant.

This definition can be extended to assess satisfaction over a time interval $[\tau_0, \tau_1]$, accounting for variations in throughput and demand:

$$DSR_f^{[\tau_0,\tau_1]} = \frac{\int_{\tau_0}^{\tau_1} T_f^{\text{alloc}}(\tau)\, d\tau}{\int_{\tau_0}^{\tau_1} T_f^{\text{req}}(\tau)\, d\tau} \tag{2.11}$$

where $T_f^{alloc}(\tau)$ and $T_f^{req}(\tau)$ represent the allocated and requested throughput of flow f as functions of time.

To capture an overall view of network performance, we define the average DSR across all flows $\mathcal{F}$ over the interval $[\tau_0, \tau_1]$ as:

$$\overline{DSR}^{[\tau_0,\tau_1]} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \text{DSR}_f^{[\tau_0,\tau_1]} = \frac{1}{|\mathcal{F}|} \sum_{f \in \mathcal{F}} \frac{\int_{\tau_0}^{\tau_1} T_f^{\text{alloc}}(\tau)\, d\tau}{\int_{\tau_0}^{\tau_1} T_f^{\text{req}}(\tau)\, d\tau} \tag{2.12}$$

where $\mathcal{F}$ denotes the set of all active flows during the interval $[\tau_0, \tau_1]$, and $|\mathcal{F}|$ is the cardinality of that set.

3

Design of a Flow-Based Network Simulator

This chapter presents the design of OpenDCN. The goal is to identify design requirements in Section 3.1 and translate them into an architecture whose overview of its modular components is detailed in Section 3.2. We introduce the flow-level abstraction used to model network traffic in Section 3.3. Section 3.4 discusses the mechanisms for managing policies such as routing, energy efficiency, and QoS. Finally in Section 3.5 we present a taxonomy of alternative abstraction levels used in network simulation.

3.1 Requirements

In this section, we determine requirements that our datacenter network simulator should address.

R1 Performance Some radndom bullshit.

R2 Query Validity Some radndom bullshit.

R3 Support the most well-known as well as user-defined topologies. To facilitate research and experimentation in the field, as well as educational purposes, the system should be designed to be generic, capable of supporting arbitrary network topology, this includes fine-grained user-defined configurations as well as convenient abstractions for building common topologies using a small set of parameters. While this flexibility enables a broad range of experimentation, it also introduces significant design and implementation complexity.

R4 Enable reuse and sharing of designs of topologies and single components. To reduce the complexity of datacenter network design and lower the barrier to entry, the simulator should support modular reuse and sharing of both complete topologies and individual components. This feature also facilitates *repeatability* and *reproducibility* of results [1], enabling consistent evaluation across shared scenarios. However, this introduces the challenge of defining a non-volatile representation for a wide range of network components, ensuring that they can be easily loaded, exported and composed within the simulator environment.

R5 Support easy addition and modification of policies and protocols. The system should offer pre-defined routing protocols, as well as QoS and energy-saving policies, reducing the barrier to entry. These policies should be applicable on the network, node and job level, to enable *Software Defined Networking* () and network aware strategies. The system must allow users to define, implement, and evaluate custom policies with minimal effort. This flexibility

introduces complexity, necessitating tooling for testing and debugging user-defined protocols and strategies.

R6 Support educational purposes with interactive simulation interface. The simulator should provide an interactive mode tailored for educational and training purposes, enabling users to conduct guided, small-scale experiments. This mode provides fine-grained control over simulation behavior, allowing the user to dictate every event, including advancement of virtual time, enhancing observability. Such capabilities offer a significantly deeper understanding of network behavior compared to traditional simulation. Supporting this interactive functionality requires a distinct set of features and expands the overall scope and complexity of the system.

R7 Permissive trace format and synthetic traffic patterns. Item **R7** addresses the need to accommodate a broad range of publicly available traffic traces, many of which are non-standardized or contain missing values. However, this permissive format adds complexity to the system, as it must be capable of converting diverse formats into valid network workloads that can be interpreted and executed by the simulator. In addition, the simulator should support the generation and evaluation of synthetic traffic patterns (as described in Section 2.4), a common practice in networking research [33]. Many existing simulators offer synthetic traffic as their primary or sole mode of operation, making its inclusion essential for comparative and exploratory studies.

R8 Provide comprehensive output metrics with adjustable granularity. The system should offer a comprehensive range of output metrics, covering both QoS and energy consumption. Users must be able to selectively specify which metrics to export and define the desired export interval. This flexibility reduces unnecessary output, minimizes file sizes, and streamlines post-simulation analysis.

R9 Adhere to modern software development standards. The system should be designed with long-term evolution and extensibility in mind, enabling future research contributions and further development. To achieve this, it must follow professional software engineering practices, including modularity, clear documentation, and automated testing. Item **R9** ensures maintainability, scalability, and reliability of OpenDCN over time.

R10 Ensure high performance and scalability of the simulator. The system must handle large-scale simulations with limited performance overhead, with chosen level of abstraction that balances accuracy and efficiency Section 3.3. Specifically, it should support the simulation of millions of network events across thousands of nodes within seconds to a few minutes. Achieving this level of performance is particularly challenging given the computational and memory demands of simulating complex datacenter networks, in conjunction with the flexibility and extensibility that we aim for.

3.2 Design Overview

In this section, we discuss the high-level architecture of the simulator, as shown in Figure 3.1. At the highest level, we identify four main components: an experiment runner for workload-based experiments; the REPL environment, which facilitates experimentation, the creation of new pre-fabricated components, as well as testing and debugging; a set of structures responsible for controlling the network's discrete event simulation and providing snapshots of the system's state; and, finally, the datacenter network model itself. Additionally, the figure illustrates, at a high level, how a compute simulator module could interact with our simulator. The policy management aspects will be dis-

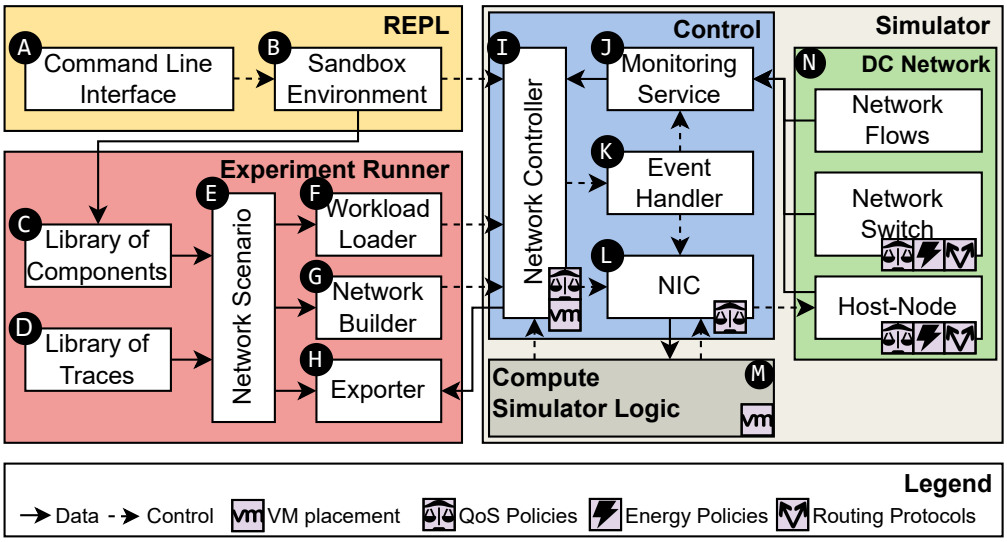


Figure 3.1: An overview of the architecture of the DC Network Simulator.

cussed in Section 3.4. We now proceed to discuss each of these components and their respective subcomponents.

REPL (addresses requirements R5, R6): The *Command Line Interface* (A) is beneficial for users as it allows efficiently set up, configure, and reproduce experiments. It is also through the CLI that users can benefit from the *REPL Environment*. The *Sandbox Environment* (B) (Section 4.5) provides an interactive environment for event-driven simulations as well as for building and modifying components and topologies. It allows users to rapidly test newly defined policies and protocols in a more debugging-friendly environment, addressing **R5**. This feature is particularly valuable for educational purposes and facilitates plug-and-play simulations, exploring network behaviors with ease, addressing **R6**.

Experiment Runner (addresses requirements R3, R4, R7, R8): The *Library of Components* (C) offers built-in network components and topologies, simplifying the design process. The library enables construction of complex datacenter networks without the need to design each element from scratch, minimizing the barrier to entry. The library includes Fat-Tree, Spine-Leaf, Dragonfly, Flattened Butterfly, and other widely used topologies and components. The library can be extended by adding user-defined components and topologies, facilitating their reuse and sharing, addressing **R4**. The *Library of Traces* (D) provides a set of built-in network traces. This feature allows to run experiments without generating custom network traces, which may be time-consuming. Producing specific traces may yield more representative results for specific environments. The *Network Scenario* (E) serves as an aggregator that encapsulates all the necessary information required for a given simulation, including monitor granularity and selection of output metrics, partly addressing **R8**. The network scenario functions as the final input format for the simulator (Section 4.4). The *Workload Loader* (F) converts a wide range of publicly available input traces into a standardized workload executable by the simulator (Section 4.4.2), addressing **R7**. The *Network Builder* (G) converts network components and topologies, that may be defined recursively, or manually, from raw JSON files to their runtime representation, partly addressing **R3** The *Exporter* (H) (Section 4.3) handles exporting tracked metrics to output files, according to configurations provided by the

network scenario. It allows to selectively choose which metrics are to be exported among the many available and with which granularity, enhancing the control the user has over the output.

Control (addresses requirement R2, R8, R10): The network logic is controlled by this module, which handles the simulation and the monitoring of resources, while ensuing scalability and minimal performance degradation, addressing **R10**. The *Network Controller* (Ⓘ) serves as the interface that triggers every event within the network, including connection and disconnection of components, while supplying real-time monitoring data to higher-level components. This component also supports integration with an external compute simulator, delegating control over network behavior and providing real time feedback for possible workload dependencies between the two modules, addressing **R2**. The *Monitoring Service* (Ⓙ) provides real-time monitoring of a wide range of network QoS and energy consumption metrics (Section 4.3), at the level of individual flows, nodes, and the entire network, addressing **R8**. The *Event Handler* (Ⓚ) handles network events ensuring their order of execution and consistency with the simulation virtual time (Section 3.3.3). The *Network Interface Controller* (Ⓛ) serves as an abstraction provided by certain network nodes that enables to directly manage their network behavior, while providing real-time information about network performances, allowing adjustment in response to changing conditions. It is especially used in combined compute-network simulations, addressing **R2**.

Compute Simulator Logic (addresses requirement R9): Our design facilitates not only independent network simulations but also the integration of networking and computing simulations. This capability is illustrated in the figure by the *Compute Simulator Logic* (Ⓟ), which can manage network operations either through the Network Controller or via NICs. This approach enables a detailed representation of the interactions between networking and computing components.

DC Network (addresses requirements R5, R9): The *DC Network* Ⓠ is built of all different network components (e.g. switches, host-nodes, links), and accurately simulates their interactions. These interactions are modeled in a highly generic manner, enabling the representation of virtually any network topology, provided it is manually specified, addressing **R3**.

3.3 Simulating Network Traffic with Flow Abstraction

This section introduces our approach to modeling network traffic using a flow-based abstraction. In Section 3.3.1 we present the base concepts of our model. Section 3.3.2 details how we represent traffic congestion using flows rather than explicit packet queuing mechanisms. Finally, in Section 3.5, we discuss alternative abstraction levels commonly in network simulation. A simple example illustrating flow behavior is shown in Figure 3.2 and is used throughout this section to support the explanations

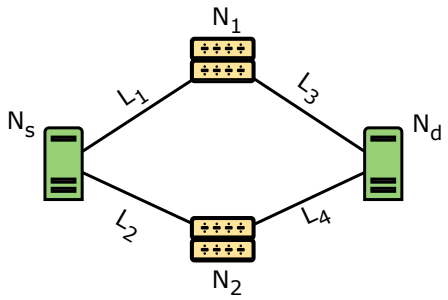


Figure 3.2: Illustration used to explain flow-congestion modeling.

3.3.1 Model Overview

In this work, we adopt a flow-level abstraction. A network traffic flow has a source node, a destination node, and an associated demand. The source and destination nodes may reside within the datacenter, representing intra-datacenter (also called *East-West*) communication or connect to external entities representing inter-datacenter (also called *North-South*) communication. It is important to support intra-datacenter flows, as East-west traffic is steadily increasing [10]. Each end-to-end flow may be decomposed into multiple sub-flows across different paths. The decomposed flow can then be reassembled at the destination. This decomposition enables support protocols that distribute traffic across multiple paths,.

An example of this behavior is depicted in Figure 3.2, where flow f, with N_s and N_d as source and destination nodes respectively, is decomposed across links L_1 and L_2, and then recomposed at N_d. Each sub-flow may experience different levels of congestion, and the end-to-end throughput of the flow is determined by the aggregate rate at which its sub-flows successfully arrive at the destination. A formal computation of end-to-end throughput and congestion for the example illustrated in Figure 3.2, is provided in Section 3.3.2

3.3.2 Simulating Network Congestion

We proceed with a formally defining how end-to-end throughput is computed in the example shown in Figure 3.2, focusing on how network congestion is simulated in our model. Consider a flow f originating from node N_s and destined to node N_d. We compute the data rates that N_s attempts to transmit over links L_1 and L_2. The protocol in use determines these attempted transmission rates. To give an example that includes multipath routing, we assume a static multipath protocol, such as ECMP, described in Section 2.3.2. We will first consider the path through N_1. The equation in Equation (3.1) define how the tentative output data rate for N_s on link L_1 is computed.

$$\mathsf{out}^{f}_{N_s,L_1} = \text{proto}(d, N_s, N_d, L_1) \tag{3.1}$$

where $out^{f}_{N_s,L_1}$ us the output data rate for flow f on link L_1 by node N_s; and $proto(d, N_s, N_d, L_1)$ is the protocol function that based on sender node N_s, destination node N_d, output link L_1, and demand d, determines the resulting output data rate.

Since the routing protocol does not consider congestion, the output data rate computed above represents the intended transmission rate rather than the actual achieved throughput. To simulate congestion, each flow on a given link is assigned a throughput based on the following rule: if the aggregate tentative data rate does not exceed the link's capacity, each flow receives its full attempted

data rate; otherwise, the link's capacity is shared among flows proportionally to their attempted transmission rates. This results in the following expression:

$$\mathsf{tput}^f_{L_1} = \min\left(\frac{\mathsf{out}^f_{N_s,L_1}}{\sum_{f'\in\mathcal{F}_{L_1}} \mathsf{out}^{f'}_{N_s,L_1}} \cdot c_{L_1},\quad \mathsf{out}^f_{N_s,L_1}\right) \qquad \textbf{(congestion equation)} \tag{3.2}$$

where c_{L_1} is the capacity of link L1; $\mathcal{F}_{\mathcal{L}_\infty}$ is the set of flows passing through link L_1; and $tput^f_{L_1}$ is the actual throughput through link L_1.

As N_1 receives flow f only through link L_1, the total data rate received at N_1 is given by the throughput of f on L_1, as follows:

$$\mathsf{in}^f_{N_1} = \mathsf{tput}^f_{L_1} \tag{3.3}$$

where $in^f_{N_1}$ is the incoming data rate of flow f to node N_1.

We compute the tentative output rate of N_1 on link L_3, as well as the resulting throughput after accounting for congestion, with the same logic as in the previous steps:

$$\begin{aligned} \mathsf{out}^f_{N_1,L_3} &= \text{proto}(\mathsf{in}^f_{N_1}, N_1, N_d, L_3) \\ \mathsf{tput}^f_{L_3} &= \min\left(\frac{\mathsf{out}^f_{N_1,L_3}}{\sum_{f'\in\mathcal{F}_{L_3}} \mathsf{out}^{f'}_{N_1,L_3}} \cdot c_{L_3},\quad \mathsf{out}^f_{N_1,L_3}\right) \end{aligned} \tag{3.4}$$

We can repeat Equations (3.1) to (3.4) for the other path throgh N_2, obtaining the throughput of f through link L_4. Finally, the end-to-end throughput of f is equal to the received data rate of f by N_d which is equal to the aggregated throughputs of f on L_1 and L_2, as described in Equation (3.5).

$$\mathsf{tput}^f = \mathsf{in}^f_{N_d} = \mathsf{tput}^f_{L_3} + \mathsf{tput}^f_{L_4} \tag{3.5}$$

where $tput^f$ is the end-to-end throughput of flow f.

event	invalidation	description
`flowstart`(n_s, n_d, d, t)	✓	initiates a new flow from source node n_s to destination node n_d, with demand d and target data transfer size t.
`flowchange`(f, d)	✓	updates the demand of flow f to a new value d.
`flowstop`(f)	✓	terminates flow f
`linkmake`(n_1, n_2, b)	✓	adds a link connecting nodes n_1 and n_2.
`linkremove/fail`(l)	✓	removes link l
`nodemake`(p, s)	✗	adds a new node to the network, with number of ports p and maximum port speed s.
`noderemove/fail`(n)	✓	removes node n
`nodeoff`(n)	✓	node n remains physically connected but no traffic can pass through and power consumption is set to 0.
`nodeon`(n)	✓	inverse of `nodeoff`.

✓= may cause invalidation of network state and propagation of updates

Table 3.1: Set of network events in our model.

3.3.3 Network Stability and Events

Our model abstracts away network latency by assuming that all changes in the network state occur instantaneously. As a result, simulation time advances only when the network reaches a *stable* state.

Network Stability. A network state is considered stable when all pending updates have been completely propagated and applied across the system, such that, without any externally triggered events, the system remains unchanged over time. Given that time advances only when the network is in a stable state, under the assumption of deterministic policies, protocols, and failure models, the overall system behavior becomes theoretically fully deterministic.How this is ensured and implementation is explained in Section 4.2.2.

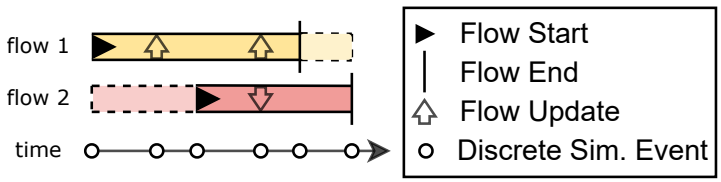


Figure 3.3: Timeline Example of two Flows' Events.

Despite the level of abstraction, we argue that the model mantains sufficient fidelity for the purposes of our evaluation, specifically in the analysis of congestion dynamics, energy consumption, bandwidth allocation, and QoS metrics, as will be demonstrated in Chapter 5. We proceed by formalizing the class of network events that, within our model, may invalidate the network state and trigger update propagation. The set of such events is summarized in Table 3.1, while Figure 3.3 shows the timeline of two flows and the update propagations that they trigger.

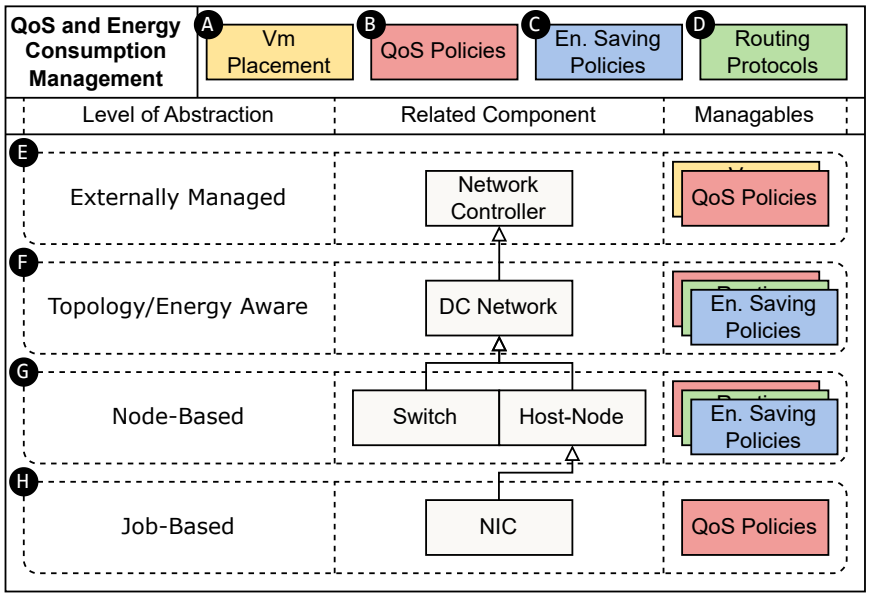


Figure 3.4: Overview of Policy Management. [TO BE ADAPTED]

3.4 Managing Routing, Energy Saving and QoS Strategies

We describe the various mechanisms through which the simulator enables users to implement custom strategies for routing, energy efficiency, and QoS management. An overview of the policy management model is illustrated in Figure 3.4.

The *VM Placement* (A) is the process of selecting an appropriate host machine, based on resource available on such machine, for executing a specific job/task. The *QoS Policies* (B) define how network resources, typically bandwidth, are allocated to each flow or job, according to some algorithm. The *Energy Saving Policies* (C) implement energy-efficient strategies, such as powering down or adjusting maximum speeds of ports and switches, as well as rerouting network flows to reduce overall energy consumption. The *Routing Protocols* (D) determine how flows are routed through the network. These protocols typically balance achieving high performance by distributing the workload across multiple paths, with the need to exclude certain links to enable the energy-saving policies to power down certain devices. As a result, routing protocols and energy-saving strategies are closely correlated, often influencing each other's behavior and effectiveness. We evaluate the impact of different routing protocols in Section 5.3.1. *Vm Allocation* and *QoS Policies* can be *Externally Managed* (E), and applied through the Network Controller interface. In particular, Vm Allocation can both be pre-defined by certain trace formats or determined dynamically at runtime through a provisioning process, managed by the compute simulator logic. The Network controller offers real-time insights on the performance of each component of the network, enabling adjustments to data rates based on these metrics. *QoS Policies*, *Routing Protocols* and *Energy Saving Policies* can be managed on the *Topology/Energy Aware* level of abstraction (F). These policies are configurable on the DC Network component and offer access to the entire network state, useful for optimizing performance and energy efficiency across the system. The same three managements can be performed on the *Node Level* (G). In this case, each node makes autonomous decisions based solely on the information it possesses. *QoS Policies* can also be applied on the *Job Level* (H). Each job running on a node possesses full information about the job's connections and network flows,

allowing it to manage and optimize its own network resources allocation independently.

3.5 Alternative Abstraction Levels: a Taxonomy

We present alternative abstraction level for network simulation. Different levels present tradeoffs between computational overhead and result accuracy.

Bit-Level. Also known as *link-level* or *physical-layer* (PHY) simulation, bit-level models simulate transmission of individual bits, including effects like modulation, interference, and noise. Due to its high computational cost, it is uncommon in datacenter simulations and is mainly used in specialized domains such as wireless or optical networks.

Flit-Level. A *flit* (flow control digit) is the smallest unit of data that can be transferred in a single cycle within *Network-on-Chip* (NoC) architectures and cycle-accurate simulations. Flit-level simulation is typically used for evaluating micro-architectural details, including buffer utilization, flow control mechanisms, and arbitration. BookSim [33] is a prominent example of a flit-level simulator, and will be used as a point of comparison in 5.2.

Packet-Level. Packet-level simulation models the behavior of individual network packets, capturing aspects such as routing, queuing, congestion control, and packet-level loss. Popular simulators at this level include ns-3 [28], OMNeT++[61]. Packet-level simulation can still be computationally intensive at large scales.

Flow-Level. Flow-level simulation abstracts network traffic as flows, used mostly to evaluate bandwidth sharing, QoS etc. This approach is well-suited for analyzing bandwidth sharing, QoS, and traffic scheduling. Examples of flow-level simulators include SimGrid [14].

Message-Level Message-level simulation abstracts communication at the level of application messages. Delays and congestion are typically ignored, focusing on latency.

4

Prototype Implementation and Integration of OpenDCN

This chapter outlines the implementation of OpenDCN and its integration with OpenDC. Section 4.1 explains how the simulator was developed to be highly parallel, to enable simulation at scale. Section 4.2 explains the simulation scope, which encapsulate all components of the simulation environment, detailing the most important ones, while showing its non-volatile representation. Section 4.3 lists the most important network metrics that can be exported with each simulation. High level explenation of the four simulation modes (trace-based, traffic-pattern-based, interactive, and compute-network) are explained in Sections 4.4 to 4.7.

4.1 Implementing for Performance

This section introduces the parallelisation principles of OpenDCN. The system was designed to scale effectively with large topologies and workloads. ODCN leverages Kotlin *coroutines*[1], user-space constructs for asynchronous programming. Introduced in Kotlin 1.1 and stabilized in version 1.3, coroutines provide a more efficient and scalable alternative to traditional threads. During simulation, each major component executes its logic within its own coroutine. For example, every network node is assigned a dedicated coroutine, enabling concurrent and independent execution. Communication between components is achieved through a system of flyweight objects [22]: *messages* and *events*.

Message (`Msg`). A `Msg` represents a directed communication sent to a specific component. It typically contains a state update or a command to trigger an action.

Event (`Evnt`). An `Evnt` is a broadcast-style communication emitted by a component in response to a specific condition or internal transition. Events may be received and processed by multiple components independently.

We now explain how each parrallelized network component works. Each component has a *message channel* (Ⓐ) buffers incoming `Msg` objects directed to the component. These messages typically represent commands or state updates issued by other components or external control logic. A different coroutine is initialized to run each component logic, receiving and handling `Msgs` and emitting `Evnts` to be handled by other components. Emitted events are queued in the an *event channel*, where

[1] *Kotlin Coroutines*. Available at: `https://kotlinlang.org/docs/coroutines-overview.html`.

Component	Code Name	Section	Role/Objective
Network	`Network`	Section 4.2.1	Contains nodes, switches, hosts, links and flows.
Stability Barrier	`NetSimBarrier`	Section 4.2.2	Ensures *network stability* and *consistency* during simulation.
Flyweight Pool	`FWPool`	Section 4.2.3	Enables reuses of objects to cut allocations and GC overhead.
Address Manager	`AddrMngr`	Section 4.2.4	Assigns and manages addresses/subnets
Configuration	`NetSimConfig`	Section 4.2.5	Container for simulation scope configurations and defaults.
Routing Policy	`RoutPolicy`	-	Determines routing throughout the simulation.
Energy Recorder	`EnRecorder`	-	Records power and energy consumption of all components throughout the simulation.
Network Exporter	`NetExporter`	-	Handles export of a user-selected set of metrics to parquet output.
Network Time Source	`NetSimTmSrc`	-	Manages the network simulation virtual time, which may be desync. from the compute one.

Table 4.1: Overview of main components in the network simulation scope (`NetSimScope`).

they remain available to registered *listeners*. Each listener processes events independently, enabling modular and decoupled reactions to internal state changes. Once all listeners have completed their handling of an event, the event object is returned to its flyweight pool (explained in Section 4.2.3) for reuse. All messages and events buffered at a given simulation step are processed within the same virtual time instant, as explained in Section 3.3.3. We proceed giving a new definition of network *stability*, from the messages and events perspective.

Network Stability (revised). The network is considered *stable* if and only if all message and event channels across all components are empty. Further discussion on stability and its enforcement can be found in Section 4.2.2.

Our goals of *modularity* and *extensibility* are addressed by this parallelized approach, allowing new features and behaviors to be integrated with little to no effort. Furthermore, it significantly improves performance over the sequential implementation, as demonstrated in the experimental evaluation in 5.3.2.

4.2 Network Simulation Scope

The Network Simulation Scope (`NetSimScope`) represents the core context for executing network simulations in OpenDCN. It encapsulates all resources, configurations, and runtime state relevant to a single simulation instance. It implements `kotlin.coroutines.CoroutineScope`, enabling concurrent execution independent of any external control logic. Structured concurrency is supported, as subscopes can be initiated. Each scope, root or nested, can be individually cancelled at runtime. Cancellation halts all coroutines within the scope and releases associated resources.

Table 4.1 illustrates the internal composition of the `NetSimScope`, which contains all entities tied to

Topology	Parameters	Routers/Switches (R)	Terminals/Hosts (N)	Edges/Links (E)
FatTree[6]	k (pods/router-radix)	$\frac{5 \cdot k^2}{4}$	$\frac{k^3}{4}$	$3 \cdot N$
Dragonfly[36]	a (routers per group), $g = a \cdot h + 1$ (groups), $p = a/2$ (terminals per router), $h = a/2$ (intergroup links per router)	$a \cdot g$	$a \cdot p \cdot g$	$R \cdot p + \frac{R \cdot (a-1)}{2} + \frac{R \cdot h}{2}$
Flattened Butterfly[35]	n (dimensions), k (routers per dimension), $c = k$ (terminals per router)	k	$R \cdot c$	$\frac{(k-1) \cdot n \cdot R}{2} + c \cdot R$

Table 4.2: Overview of some of the built-in topologies provided in OpenDCN, with some size and complexity indicators.

the simulation's lifecycle and configuration. Sections 4.2.2 to 4.2.5 provide an overview of the scope main components. Finally, Section 4.2.5 describes the non-volatile JSON-based representation of the scope.

4.2.1 Network Construction in Practice

In this section, we describe the methods available for constructing network topologies. In Section 4.2.1.1 we present our current selection of built-in topologies, widely used designs that can be constructed with a small set of parameters, simplifying simulation setup. In Section 4.2.1.2 we show how to build your own custom topology, enabling exploration of new architectures, including switch-centric, server-centric or hybrid designs (explained in Section 2.2.3).

Additional configuration options available through `NetSimConfig`, such as defining default node settings, are discussed in Section 4.2.5. These global properties simplify topology definitions and make it easy to update the configuration of all nodes consistently.

4.2.1.1 Minimized Setup with Built-In Topologies

To facilitate simulation setup and lower the barrier to entry, we provide a built-in set of topologies, widely used in research and practice, that can be constructed with as few as a single parameter. An overview of some of the available topologies, their defining parameters, and indicative metrics characterizing the resulting network size and complexity is presented in Table 4.2. In the following, we describe the key characteristics of these topologies and outline the steps required to construct them.

```
{
    "type": "ftree",
    "k": 4,
}
```

(a) FatTree

```
{
    "type": "dragonfly",
    "a": 4,
    "g": 10,
    "h": 4,
    "p": 4
}
```

(b) Dragonfly

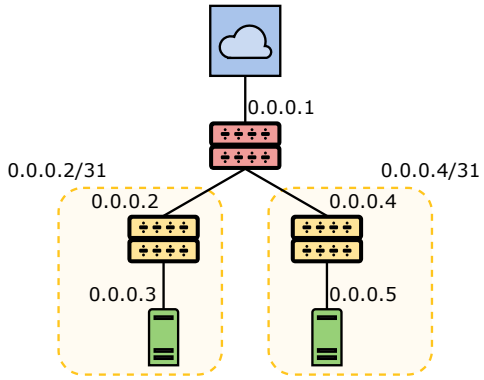
```
{
    "type": "flatfly",
    "n": 2,
    "k": 4,
    "c": 4

}
```

(c) Flattened Butterfly

Figure 4.1: JSON representations of some of the built-in topologies.

```
{
   "type": "custom",
   "terminalSpecs": [
      {
         "nPorts": 10,
         "portSpeed": "1Gbps",
         // ip and its integer repre-
         // -sentation are interchangable.
         "ip": "0.0.0.2" // Or 2
      },{
         "nPorts": 10,
         "portSpeed": "1Gbps",
         "ip": 4 // Or "0.0.0.4"
      }
   ],
   "switchSpecs": [
      {
         "nPorts": 10,
         "portSpeed": "2Gbps",
         "ip": "0.0.0.1",
         // Global implies direct or
         // indirect internet access.
         // In some topologies these
         // are known as Core Switches.
         "global": true
      },{
         "nPorts": 10,
         "portSpeed": "1Gbps",
         "ip": "0.0.0.3"
      },{
         "nPorts": 10,
         "portSpeed": "1Gbps",
         "ip": 5
      }
   ],
   "links": [
      [2, 1],
      [3, 1],
      [4, 2],
      [5, 3]
   ],
   "subnets": [
      "0.0.0.2/31",
      "0.0.0.4/31"
   ]
}
```

(a) JSON representation of a custom network topology in OpenDCN.



(b) The visual representation of the custom topology defined in Figure 4.2a.

Figure 4.2: Custom network topology in OpenDCN.

The non-volatile JSON representations of such topologies in OpenDCN are shown in Figure 4.1. Node configurations can be set globally, through `NetSimConfig`, as explained in Section 4.2.5.

4.2.1.2 Design Without Limits: Build Your Own Topology

To support research and the exploration of novel designs, our simulator allows the construction of arbitrary, user-defined topologies. Figure 4.2 illustrates the persistent (non-volatile) representation of such topologies, which are defined by a set of terminals (or hosts), routers (or switches), edges (or links) and subnets. Each node can be individually configured or inherit settings from globally defined defaults specified via `NetSimConfig`.

Alternatively, users can construct custom topologies interactively using the REPL environment. This approach allows for step-by-step definition of components with fine-grained control, and the resulting topology can be exported to JSON for reuse. The most important commands available in the REPL environment are explained in Section 4.5.

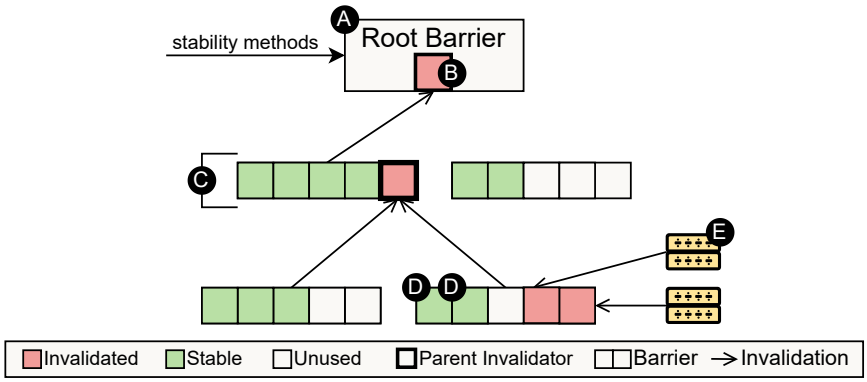


Figure 4.3: Visual representation of the Network Stability Barrier.

4.2.2 Synchronization Mechanism: Stability Barrier

In Section 3.3.3, we introduced the concept of network stability and described how our model revolves around it. A revised definition was provided in Section 4.1. In this section, we detail the key component responsible for maintaining and enforcing network stability: the *Network Stability Barrier*.

Network Stability Barrier (`NetSimBarrier`). The `NetSimBarrier` is a component of `NetSimScope` that monitors and enforces network stability through a structured synchronization mechanism. Conceptually, it functions as a hierarchical (tree-like) barrier, where components can *reach* the barrier, for example indicating they have finished processing all messages, or *unreach* it, if new messages arrive.

Network Stability Invalidator (`NetSimInvalidator`). Each barrier is associated with one or more NetSimInvalidators. An invalidator can mark itself (and, by extension, the barrier it belongs to) as unstable. This happens when a component, such as a node, receives new messages or events that require processing.

Figure 4.3 shows a visual representation of a simple small barrier. The barrier is organized as a tree, with sub-barriers as nodes. The *Root Barrier* (A), is linked to a single *Invalidator* (B), which determines the global stability of the network. Each *Child Barrier* (C) is connected to a parent invalidator, enabling it to propagate instability upwards in the tree. Sub-barriers are themselves linked to multiple invalidators (D), one of which may be assigned to downstream sub-barriers. The remaining invalidators are assigned to *Invalidatable Components* (E), such as nodes. These components can independently mark themselves as unstable, for instance, when a node receives a new message to process.

This hierarchical structure is designed to minimize contention on a central or global invalidation counter. Because sub-barriers operate independently, invalidators across different parts of the tree can be validated or invalidated concurrently. As a result, the system scales more effectively with increasing network size and complexity. The barrier can dynamically grow at runtime to accommodate these changes. Now that we have explained what the network stability barrier is, we can revise again the definition of network stability, from the stability barrier perspective.

Network Stability (final). The network is considered *stable* if and only if all `NetSimInvalidators` associated with the `NetSimBarrier` are in a validated state (the components that own them are stable). This, in turn, guarantees that the root invalidator is in a validated state.

```
○ 0.0.0.0/0 (41)
├─○ 0.0.0.0/26 (40)
│ ├─○ 0.0.0.0/27 (36)
│ │ ├─○ 0.0.0.0/28 (18)
│ │ │ ├─● 0.0.0.0/29 (9) // Pod 1
│ │ │ ├─○ 0.0.0.0/30 (4) // Pod 1 switches
│ │ │ │ ├─○ 0.0.0.0/31 (2)
│ │ │ │ │ ├─● 0.0.0.0 (1)
│ │ │ │ │ └─● 0.0.0.1 (1)
│ │ │ │ └─○ 0.0.0.2/31 (2)
│ │ │ │   ├─● 0.0.0.2 (1)
│ │ │ │   └─● 0.0.0.3 (1)
│ │ │ └─○ 0.0.0.4/30 (4) // Pod 1 terminals
│ │ │   ├─○ 0.0.0.4/31 (2)
│ │ │   │ ├─● 0.0.0.4 (1)
│ │ │   │ └─● 0.0.0.5 (1)
// Some lines are omitted.
```

```
// Some lines were omitted
│ │   └─● 0.0.0.24/29 (9) // Pod 4
│ │     ├─○ 0.0.0.24/30 (4) // Pod 4 switches
│ │     │ ├─○ 0.0.0.24/31 (2)
│ │     │ │ ├─● 0.0.0.24 (1)
│ │     │ │ └─● 0.0.0.25 (1)
│ │     │ └─○ 0.0.0.26/31 (2)
│ │     │   ├─● 0.0.0.26 (1)
│ │     │   └─● 0.0.0.27 (1)
│ │     └─○ 0.0.0.28/30 (4) // Pod 4 terminals
│ │       ├─○ 0.0.0.28/31 (2)
│ │       │ ├─● 0.0.0.28 (1)
│ │       │ └─● 0.0.0.29 (1)
│ │       └─○ 0.0.0.30/31 (2)
│ │         ├─● 0.0.0.30 (1)
│ │         └─● 0.0.0.31 (1)
// Core switches are omitted.
```

●= created subnet or assigned ip, ○= representation grouping

Figure 4.4: Prefix-trie of FatTree depicted in Figure 2.3, displayed through REPL environment.

4.2.3 Flyweight Pool

The `FWPool`, allows optimization through reuse of objects instead of initialization, according to flyweight design pattern [22]. It is used primarily for `Msg` and `Evnt` objects, but also for some internal implementation objects. The `FWPool` is organized into type-specific pools, each of which is further divided into multiple sub-pools. This structure allows multiple coroutines to acquire objects of the same type concurrently via separate sub-pools. Each coroutine is assigned a single sub-pool, determined by computing the modulus of its id with the total number of sub-pools. Sub-pools are implemented as `kotlin.coroutines.Channel` instances: acquiring a flyweight object corresponds to receiving from the channel, while disposing of it corresponds to sending it back. `FWPool` sizes and runtime checks can be configured in the simulation scope JSON representation, explained in Section 4.2.5, although these specific implementation configurations are omitted.

4.2.4 Address Manager and Subnets

OpenDCN includes explicit support for the creation and management of IP subnets. This functionality serves multiple purposes: it enables structured topology definitions, facilitates experiments involving subnet-dependent routing, and optimizes routing state storage for large-scale simulations. In particular, when simulating protocols across tens of thousands of nodes, routing tables can grow prohibitively large. By organizing nodes into subnets, the system can significantly reduce the size of routing tables through hierarchical summarization of routing information. Routing data is populated dynamically at runtime using distance-vector protocols; therefore, any reduction in the number of unique routing entries directly contributes to space efficiency.

Subnet management and IP address assignment are handled by the *Address Manager* (`AddrMngr`) component. This module is responsible for maintaining the mapping between node identifiers and IP addresses, managing subnet hierarchies, and dispensing IPs in a controlled manner. The `AddrMngr` supports arbitrarily deep nesting of subnets, constrained only by the availability of address space. This hierarchical approach enables efficient routing information sharing and enhances scalability.

Figure 4.4 illustrates the prefix-trie structure generated by the `AddrMngr` for the FatTree topology depicted in Figure 2.3. This trie reflects the logical subdivision of the address space and supports runtime visualization via the REPL interface and during experiment initialization.

```
{
 // Section 4.2.1
  "netPath": "network/path",
  "config": {
    "stabilityMode": "checked",
    "routingPolicy": "ecmp",
    "seed": 0,
    "exportConfig": {
      "networkExportColumns": [
        // Section 4.3
      ],
      "nodeExportColumns": [
        // Section 4.3
      ],
      "outputFolder": "path/to/file",
      "exportInterval": "5min"
    },
    "nodeConfig": {
      "defaultPortSpeed": "100Mbps" ,
      "defaultNPorts": null,
      "defaultEnergyModel": null,
      "terminalConfig": {
        "defaultPortSpeed": null,
        "defaultNPorts": null
      },
      "switchConfig": {
        "defaultPortSpeed": null,
        "defaultNPorts": null
      }
    },
    "netSimDevConfig": {
      // Omitted
      // for brevity
    }
  }
}
```

Figure 4.5: JSON representation of a `NetSimScope`.

4.2.5 JSON Representation and Deserialization

Figure 4.5 illustrates the JSON representation of a `NetSimScope`. It contains main configuration regarding export, including selection of export columns, of which the most important are explained and listed in Section 4.3, and export interval. It allows to define default node, switch and terminal (host) configurations, to be applied whenever not overridden. The routing protocol to be used during the simulation is also included. Note that some configuration are omitted for brevity or lesser importance, while other can be overriden by the compute module in case of a combined network-compute simulation (Section 4.7).

Name	Unit	Description
network.nodes.count	-	Number of nodes currently part of the network.
network.nodes.hosts[all]	-	Number of hosts currently part of the network
network.nodes.hosts[active]	-	Number of active hosts currently part of the network
network.flows.count	-	Number of active network flows.
network.flows.throughput[tot]	Mbps	The sum of the throughput of all active flows.
network.flows.throughput[%]	%	The total throughput percentage of all active flows.
network.flows.throughput[avg%]	%	The average flow throughput percentage.
node.flows.uptime[average]	ms	The average uptime of active flows in the network.
network.energy.power	W	The current power draw of the network.
network.energy.energy	J	The energy consumed by the network up until now.
node.flows[incoming]	-	Number of flows incoming from adjacent nodes.
node.flows[outgoing]	-	Number of flows outgoing to adjacent nodes.
node.flows[generating]	-	Number of flows being generated by this node.
node.flows[consuming]	-	Number of flows being consumed by this node.
node.flows.throughput[tot]	Mbps	The total throughput on the node.
node.flows.throughput[%]	%	Total throughput percentage in the node.
node.flows.throughput[avg%]	%	Average throughput percentage of flows traversing the node.
node.flows.throughput[min%]	%	Min throughput percentage among flows traversing the node.
node.flows.throughput[max%]	%	Max throughput percentage among flows traversing the node.
node.flows.uptime[avg]	ms	The average uptime of active flows traversing the node.
nodes.flows.energy.power	W	The current power draw of the node.
nodes.flows.energy.energy	J	The energy consumed by the node up until now.
job.flows[generating]	-	Number of flows being generated by this job.
job.flows[consuming]	-	Number of flows being consumed by this job.
job.flows.throughput[tot]	Mbps	The total throughput on the job network interface.
job.flows.throughput[%]	%	The total throughput percentage on the job network interface.
job.flows.throughput[min%]	%	Min throughput percentage among flows generated by the job.
job.flows.throughput[max%]	%	Max throughput percentage among flows generated by the job.
job.flows.uptime[average]	ms	The average uptime of active flows generated by the job.

Table 4.3: Metrics exposed by the OpenDCN Simulator.

4.3 Exportable Metrics

We expose telemetry data amounting to 30+ different metrics. The most relevant are highlighted in Table 4.3, we list the representative [20, 50] metrics relevant for this work. These output metrics are consistent among all OpenDCN simulation modes.

4.4 Simulating Network Workloads with OpenDCN

This section details both implementation and setup of trace-driven network workload simulation in OpenDCN. In this simulation mode, a static network trace is converted into a sequence of runtime network events that drive the simulation, enabling the collection of detailed performance and behavioral metrics.

In Section 4.4.1 we present the OpenDCN trace format. This format is designed to support both inter- and intra-datacenter traffic patterns, accommodate multiple flows between endpoints, and provide flexibility through optional duration fields. Finally, in Section 4.4.2, we detail how to configure a trace-based simulation experiment using a JSON specification.

Field	Type	Meaning
`timestamp`	`int64`	The milliseconds elapsed from epoch
`transmitter_id*`	`int64`	The id of the transmitter node, if `null` assumed internet
`destination_id*`	`int64`	The id of the destination node, if `null` assumed internet
`net_tx`	`double`	The data-rate of the flow in `Kbps`
`flow_id†`	`int64`	The id of the flow, if `null` only 1 flow is possible between 2 nodes
`duration*`	`int64`	Flow duration in `ms`; if `null`, flow does not change until another event affects it

* = nullable field, † = nullable column (all rows or none)

Table 4.4: OpenDCN network trace format.

4.4.1 Trace Format

The trace format shown in Table 4.4 has been designed to facilitate building network workloads from any conceivable trace, supporting total control of the flow through updates, as well as through flow durations. It enables inter- and intra-datacenter communications, allowing for two flows between each host (one per direction) as well as multiple flows between two machines when a `flow_id` is specified. This flexibility ensures the generation of precise network workloads from diverse non-standardized input data. As additional constraints, `transmitter_id` and `destination_id` cannot be equal (including `null`, and either the whole `flow_id` column is specified, or it is discarded. Duration does not need to be fully defined for each row, however it is favorable to either convert the non-standardized trace to a workload with the whole duration column defined or not.

```
{
   "netSimScopeSpecPath": "path/to/json/file.json", // Section 4.2.5
   "wlPath": "/path/to/parquet/trace/file.parquet", // Section 4.4.1
   "virtualMapping": true
}
```

Figure 4.6: OpenDCN network trace simulation configuration.

4.4.2 Configuring a Network Workload Experiment in OpenDCN

The entry point for conducting network trace simulations in OpenDCN is a JSON experiment configuration file, as illustrated in Figure 4.6. The `netSimScopeSpecPath` key should be assigned the file path pointing to the network simulation scope specification, detailed in Section 4.2.5. Similarly, the `wlPath` key must reference the Parquet trace file conforming to the OpenDCN network trace specification, described in Section 4.4.1.

An additional field, `virtualMapping`, determines how node IPs from the trace are mapped to the network topology. When set to true, the trace is not tied to a specific physical topology. In such cases, each host IP in the trace is virtually mapped to a randomly selected terminal node IP within the target topology. This feature is valuable for rapid evaluation scenario using the same trace dataset. When the trace is associated with a known physical topology, `virtualMapping` should be set to false to preserve accurate mapping.

Command	Arguments/Options	Description
	Scope Control	
`import`	-	Imports a new `NetSimScope/Network/config` from the given path.
`scope`	`<path>`	-
`network`	`<path>`	-
`config`	`<path>`	
`export`	-	Exports the current active `NetSimScope/Network` to JSON format.
`scope`	`<path>`	
`network`	`<path>`	-
	Network Construction	
`node`	-	-
`mk`	`[--ip <ip>] [--ports <num>] [--speed <dr>]`	Builds a new node with the given parameters.
`switch`	-	-
`gswitch`	-	Builds a gateway switch.
`terminal`	-	-
`rm`	`<ip>`	Removal/failure of a node.
`link`	-	-
`mk`	`<ip 1> <ip 2> [--badwidth <dr>]`	Connects nodes with `ip 1` and `ip 2` with optional link capacity.
`rm`	`<ip 1> <ip 2>`	Removal/failure of link.
	Simulation Control	
`avance-time`	`<time>`	Advances the virtual simulation time by the given value.
`flow`	-	-
`mk`	`[--demand <dr>] [--src <ip>]* [--dest <ip>]*`	Starts a new flow with the given parameters.
`rm`	`<flow-id>`	Stops the flow with the given id.
`update`	`<flow-id> [--demand <dr>]*`	Updates the demand of the flow with the given id with the given data rate.
`pattern`	`<pattern name> <injection rate>`	Section 4.6.
`node`	-	-
`off`	`<ip>`	Turns off node with the given ip (no power draw).
`on`	`<ip>`	Turns on node.
	Monitoring and Export	
`energy-report`	-	Logs formatted energy related metrics.
`export metrics`	`<folder-path> [--create]`	Export metrics to the given path in parquet format.
`network`	-	-
`info`	-	-
`snapshot`	`[--select <flag-array>]`	
`prefix-trie`	`[--root <ip>]`	
`node`	-	-
`info`	`<ip>`	Logs node configuration.
`snapshot`	`[--select <flag-array>]`	Logs a snapshot of the given metrics.
`flow`	-	-
`snapshot`	`[--id <flow-id>]`	Logs the metrics related to the flow with the given id.

* = required option; `dr` denotes a data rate (e.g., `1Gbps`); `ip` denotes an IP address (e.g., `0.0.0.1` or its integer representation `1`).

Table 4.5: Overview of OpenDCN REPL main commands with arguments, options, and usage.

4.5 Interactive Simulation with OpenDCN REPL

This section introfuces interactive network simulation mode in OpenDCN. Interactive simulation is enabled via a commandline REPL interface, through which users have full control over the construction of network topologies and the manual injection of simulation events, as well as export of components and metrics. The most essential commands available in the REPL environment are summarized in Table 4.5.

The REPL interface is implemented using the Clikt library[2], which provides a modular, extensible, and structured command-line environment. The REPL is designed with user-friendliness in mind, with documentation and error messages alike.

4.6 Traffic Pattern Simulation in OpenDCN

OpenDCN supports the simulation of synthetic traffic patterns (Section 2.4), through a dedicated REPL command that allows users to inject and monitor traffic flows in a controlled and repeatable environment. The core command for simulating traffic patterns is:

```
flow traffic-pattern <pattern name> <injection-rate>
```

This command triggers the injection of a large set of flows between nodes, based on the specified traffic pattern and the configured injection rate, which can be specified both as a percentage (e.g., `80%`) and as a data rate (e.g., `100Mbps`). In the latter case, the specified data rate will be injected by each terminal in the network.

Name	Argument Name	Target Topology	Description
Uniform Random	`random`	All	Each source selects a dest. uniformely at random. More in Section 2.4.
Random Permutation	`random-perm`	All	Same as Uniform Random but with 1:1 mapping. More in Section 2.4.
Bit Complement	`bit-complement`	All	Each source selects its bitwise complement as dest. More in Section 2.4.
Bit Reversal	`bit-reversal`	All	Each source select its bitwise reversal as dest. More in Section 2.4
Bit Shuffle	`bit-shuffle`	All	-
Bit Transpose	`bit-transpose`	All	-
Tornado[56]	`tornado`	Mesh	Source sends to a dest. ”half way” across network.
Pod Permutation	`ftree-adv`	FatTree	1:1 pod mapping.
Group Permutation	`df-adv`	DragonFly	1:1 group mapping.

Table 4.6: Traffic Pattern currently supported by OpenDCN.

Table 4.6 summarizes the supported traffic patterns in OpenDCN, while an example of output following a synthetic traffic pattern simulation is shown below:

```
Executing Synthetic WL...  100% [===============] 1055/1055 (0:00:00 / 0:00:00)
| Synthetic workload executed successfully in 365.647098ms
 ==== Overview ====
 | avg-tput [%]   min-tput [%]   max-tput [%]   tot-demand     tot-tput       tot-tput [%]   ··· //
    ↪ output cut
 | 47.225%        4.713%         100%           1055.000 Gbps  498.222 Gbps   47.225%        ··· //
    ↪ output cut
```

[2] *Clikt: Command Line Interface for Kotlin.* Available at: `https://ajalt.github.io/clikt/`.

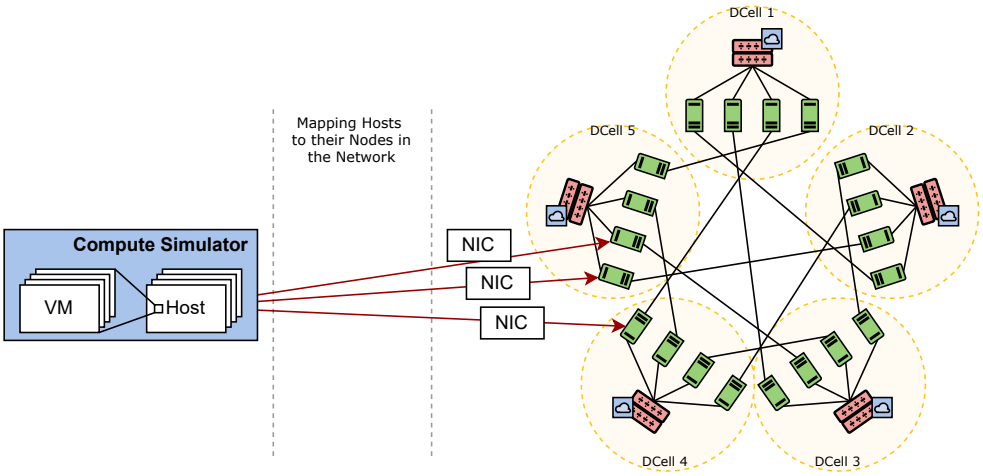


Figure 4.7: Representation of integration interaction between compute and network in OpenDC.

Since the simulation is run interactively, users can issue any REPL monitoring command to inspect network behavior and export all metrics listed in Table 4.3. The figure highlights a few key performance metrics, with many columns omitted for brevity. The example includes metrics such as `avg-tput [%]`, which indicates the average throughput achieved across all flows involved in the pattern. `min-tput [%]` shows the minimum throughput (i.e., worst-case flow), while `max-tput [%]` indicates the best-case flow. The `tot-tput [%]` metric is computed as the total throughput divided by the total demand. When all flows have the same demand, `avg-tput [%]` and `tot-tput [%]` are equal. These throughput metrics can also be interpreted in terms of *injection rate* and *acceptance rate*, widely used in cycle accurate simulations. The injection rate corresponds to the value passed to the traffic command (when expressed as a percentage), while the acceptance rate is calculated by multiplying the injection rate by `tot-tput [%]`. For instance, in the following command:

```
flow traffic-pattern <pattern-name> 100%
```

the injection rate is 100%, and the acceptance rate is equal to the measured `tot-tput [%]`. This interpretation of results will be used extensively in Section 5.2, where we compare the accuracy of our traffic pattern simulations against those of BookSim [33], a widely used cycle accurate network simulator in research.

4.7 Combined Compute-Network Simulation in OpenDC

In this section we outline the principles behind the integration of ODCN in OpenDC. We start by providing a visual representation of the interaction between network and compute modules during simulations in Section 4.7.1. Then we explain how to configure a compute-network co-simulation through JSON representation of a joint simulation experiment in Section 4.7.2.

4.7.1 Integration Overview

Network topology and machine network capacities are available for provisioning and scheduling purposes allowing network aware scheduling. Figure 4.7 shows how the compute and network modules interact during simulation of workloads. On the left there is a simplified representation

```
{
  "runs": 5,
  "outputFolder": "path/to/folder",
  "topologies": [{
    // Figure 4.8b
  }],
  "networkScopes": [{
    // Section 4.2.5
  }],
  "workloads": [{
    "pathToFile": "path/to/file",
    "type": "ComputeNetworkWorkload",
    "scalingPolicy": "Perfect"
  }],
  "allocationPolicies": [{
    // Policies
  }]
}
```

(a) Combined compute-network experiment JSON representation.

```
{
  "clusters":
  [{
    "name": "C01",
    "hosts" :
    [{
      "count": 49,
      "cpu":
        {
          "coreCount": 8,
          "coreSpeed": "3.2 Ghz"
        },
      "memory": {
        "memorySize": "128e3 MiB",
        "memorySpeed": "1 Mhz"
      },
      "powerModel": {
        "modelType": "linear",
        "power": "400 Watts",
        "maxPower": "1 KW",
        "idlePower": "0.4W"
      }
    }]
  }]
}
```

(b) Compute topology JSON representation.

Figure 4.8: Defining a compute-network co-simulation experiment in OpenDC

of the compute state at runtime, with multiple VMs per host and multiple hosts. Each host is mapped to a NIC that handles the interactions between the two modules. Each VM can perform its network workload by using the NIC of the host it is running on. Compute and network portions of the workloads are dependent one one another: the VM can adjust its network flows in response to congestion or specific VM level policies.

4.7.2 Configuring a Compute-Network Co-Simulation in OpenDCN

Figure 4.8 shows how to define a combined compute-network co-simulation experiment in OpenDC, through a JSON representation. Figure 4.8a defines the experiment, while Figure 4.8b defines the compute cluster used during the simulation. Note that in this example specific ids for machines in the compute topology are missing, therefore *virtual mapping* will be applied as discussed in Section 4.2.5. The `networkScopes` in the experiment definition are the only contact point between the 2 modules and are completely optional; if omitted compute-only simulation will be performed.

5

Experiments

This chapter presents the experimental evaluation of OpenDCN. We assess simulator's *correctness*, *scalability/performance*, and *practical utility* following three guiding questions:

- Does OpenDCN reproduce expected network behavior under controlled inputs?
- How does OpenDCN perform as problem size and concurrency increase (scalability/runtime)?
- What system-level insights become visible when the network is modeled alongside compute (vs. compute-only)?

We answer these questions through experiments with three different simulation modes: trace-based, traffic-pattern-based, and network-compute co-simulation.

Focus	Section	Sim. Mode	Description
Accuracy	Section 5.2.1	Traffic Pattern	Compare traffic pattern simulation accuracy against BookSim [33], a peer-reviewed cycle-accurate network simulator.
Performance, Scalability	Section 5.2.2	Traffic Pattern	Compare traffic pattern simulation performance against BookSim.
Topology, Routing, Energy, QoS	Section 5.3.1	Trace Based	Evaluating impact of topology and routing protocol on energy efficiency and QoS.
Performance, Scalability	Section 5.3.2	Trace Based	Evaluating performance and scalability of trace-based simulation relative to topology and parallelism.
En. Attribution	Section 5.4.1	Combined	Example use case for evaluating energy impact of network relative to that of compute/ the rest of the IT infrastructure.
Task QoS, Network Sensitivity	Section 5.4.2	Combined	Investigating task scheduling, execution, and completion performance dependency on network.

Table 5.1: Overview of the experiments discussed in this section.

5.1 Experiments Overview

This section evaluates OpenDCN in all three main simulation modes (excluding interactive). The experiments are summarized in Table 5.1. We validate traffic-pattern simulation against BookSim's baseline comparing runtime scalability with varying topology sizes and injection rates (Section 5.2). We investigate how topology and routing protocols affect QoS and energy efficiency through trace-based simulation in Section 5.3. Additionally, we also assess trace-based scalability for increasing network topologies. In Section 5.4 we make use of the integrated OpenDC performing compute-network co-simulation, attributing energy consumption to the two modules. Additionally, we perform task delay and slowdown analysis relative to a compute-only baseline.

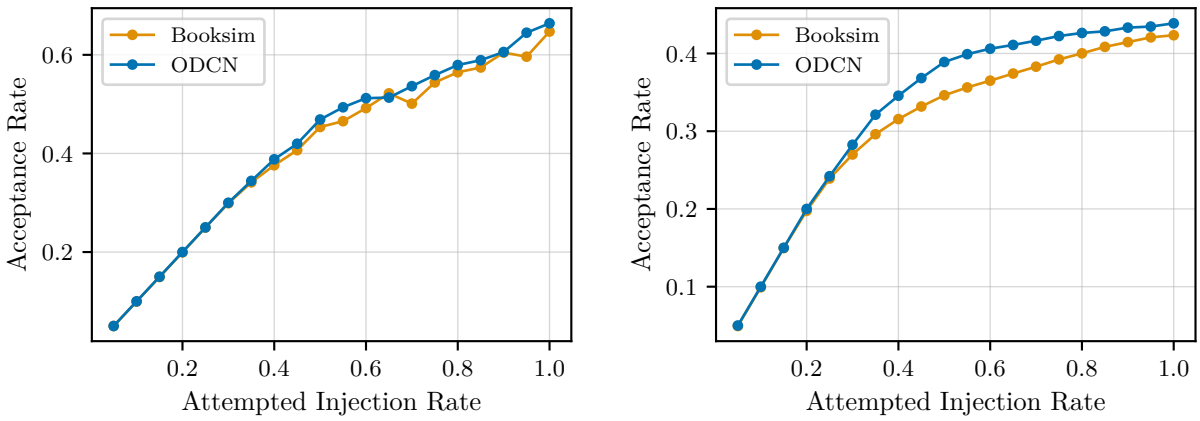
5.2 Traffic-Pattern Simulation: Validation and Performance vs. BookSim

In this experiment, we evaluate the ODCN traffic pattern simulator by comparing its performance against Booksim [33], a widely adopted, peer-reviewed, cycle-accurate simulator frequently used in network-on-chip research. In Section 5.2.1, we assess the accuracy of ODCN by comparing its results to Booksim's in terms of acceptance rate as a function of increasing offered load, across two different network topologies. In Section 5.2.2, we evaluate the runtime performance of both simulators as network complexity and injection rate increase.

5.2.1 Accuracy vs. BookSim (Accepted Rate under Offered Load)

To enable a meaningful comparison, we adopt the performance metrics reported by Booksim and convert the corresponding ODCN outputs accordingly. Specifically, we align ODCN's output with Booksim's average accepted flit rate by mapping it to ODCN's average DSR, as defined in Equation (2.12), scaled by the injection rate.

Assuming all terminals in ODCN operate at the same nominal speed s and inject traffic at a demand



(a) Flatfly(n=2,k=4,c=4)

(b) Dragonfly(a=8,h=4,p=4,g=33)

Figure 5.1: Comparison of traffic pattern simulation results under incremental injection rates. Throughput is maximized when the accepted rate matches the offered load.

level $d = p \cdot s$, where $p \in [0, 1]$ is the normalized offered load, the accepted rate (denoted acc) can be expressed as:

$$\mathsf{acc} = \overline{DSR} \cdot p \tag{5.1}$$

The attempted injection rate in Booksim corresponds directly to p.

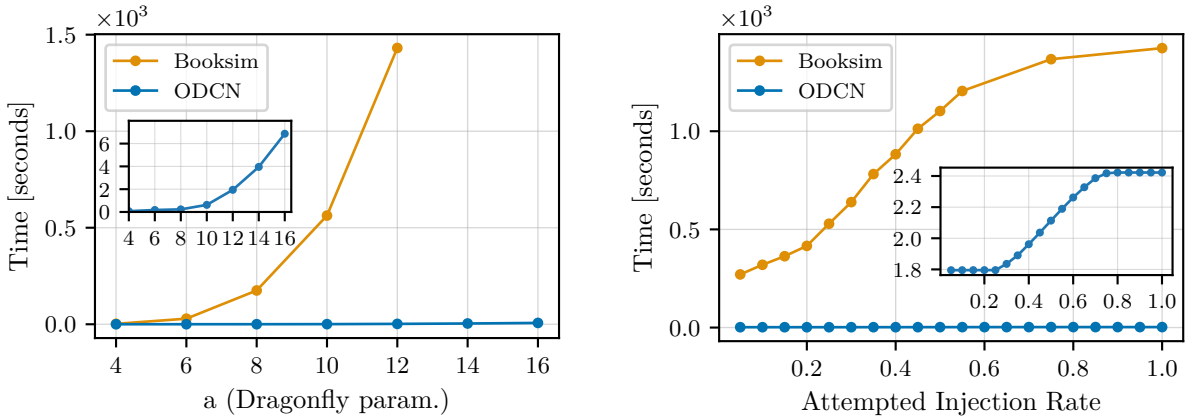
Experiment Setup: We evaluate performance on two different topologies: *Flatfly* and *Dragonfly*. The Flatfly topology is configured with parameters $n = 2$, $k = 4$, and $c = 4$, resulting in a total of 80 nodes, of which 64 are terminals, and 112 links. The Dragonfly topology uses parameters $a = 8$, $p = 4$, $h = 4$, and $g = 33$, yielding a significantly larger network with 1320 nodes (1056 terminal) and 2508 links.

Due to simulation time limitations in Booksim when using Flatfly with randomized routing, the Flatfly network configuration is kept smaller. Routing is configured as *Random Minimal (RAND MIN)* for Flatfly and *Deterministic Minimal (MIN)* for Dragonfly, as described in Section 2.3. In both cases, the injected traffic follows a *random permutation* pattern (Section 2.4).

Booksim is configured with 2 to 8 virtual channels, buffer sizes ranging from 32 to 256 flits, packet sizes ranging from 1 to 32 flits, and the default 3000 cycles warm up phase. ODCN is configured with subnets for each Flatfly dimension and Dragonfly group. Simulations are conducted using injection rate intervals of 0.05, with 20 runs per injection rate for Flatfly and 5 runs for Dragonfly. The reduced number of runs for Dragonfly is justified by its larger scale and the correspondingly lower variance in results under random permutation traffic.

Experiment Results: The results are presented in Figure 5.1. Overall, the two simulators produce comparable outcomes. Notably, ODCN shows a slightly higher acceptance rate in the Dragonfly topology, reflected in a more pronounced curvature.

We attribute this deviation to the longer average path lengths in Dragonfly compared to Flatfly.



(a) Runtime as a function of increasing Dragonfly network size (fixed injection rate at 100%).

(b) Runtime as a function of increasing injection rate (fixed Dragonfly topology with $a = 12$).

Figure 5.2: Comparison of traffic pattern simulation runtimes with increasing network size and injection rate.

These amplify the negative effects of simulated flow control in Booksim, resulting in decreased throughput. An additional contributing factor may be Booksim's modeling of virtual channels, which introduces further overhead not accounted for in ODCN.

MF1	ODCN is capable of accurately simulating network traffic patterns while operating at a significantly higher level of abstraction compared to many existing simulators

5.2.2 Runtime Scalability vs. BookSim (Topology Size and Injection Rate)

To showcase the advantages of higher-level network simulation, we compare the runtime performance of ODCN and Booksim under varying network configurations. Specifically, we aim to highlight the scalability of each simulator with respect to both increasing network size and increasing injection rate.

Experiment Setup. We evaluate scalability using the Dragonfly topology. Booksim is configured with 8 virtual channels, virtual channel buffer sizes of 256 flits, packet size of 32 flits, and the default warm-up period of 3000 cycles. ODCN is configured using dedicated subnets for each Dragonfly group.

In the first part of the experiment, we fix the injection rate at 100% and scale the network size. Specifically, we evaluate Dragonfly configurations with $a \in [4, 12]$ in steps of 2, corresponding to node counts ranging from 108 to 6132 and link counts from 162 to 12702. Additionally, due to its significantly faster execution, ODCN was also evaluated for larger configurations with $a = 14$ and $a = 16$, scaling up to 18,576 nodes and 40,248 links. For each configuration, we perform and measure the runtime of 5 simulation runs.

In the second part of the experiment, we fix the topology to Dragonfly with $a = 12$ and evaluate runtime performance under increasing injection rates from 0.05 to 1.0 in steps of 0.05.

Top	Abbrv.	Params	V	N	R	E
FatTree	FT	$k = 16$	1344	1024	320	3072
Flattened ButterFly	FF	$n = 2, k = 10, c = 10$	1100	1000	100	1900
Dragonfly*	DF	$a = 8, p = 4, h = 4, g = 33$	1320	1056	264	2500

* = these parameters make it a *balanced* dragonfly.

Table 5.2: Summary of the topologies used in the experiment, and their main characteristics.

Experiment Results. As shown in Figure 5.2a, ODCN performs better than Booksim by several orders of magnitude in all tested scenarios. At $a = 12$, Booksim requires nearly 1500 seconds to complete the simulation, while ODCN completes the same task in approximately 2 seconds, with a speedup factor of over 700×. Although both simulators exhibit increasing runtime as network size scales, their growth patterns differ. In the main plot, Booksim's exponential growth is clearly visible, whereas ODCN's runtime appears almost flat due to its much smaller scale. The inset plot, with a refined y-axis, reveals that ODCN also scales exponentially but with a less pronounced exponential curve, making it much more suitable for large-scale simulation.

Figure 5.2b illustrates the scalability of the two simulators as a function of the attempted injection rate. Once again, the simulators operate on vastly different time scales, with ODCN's runtime appearing nearly constant in the main plot. The inset plot provides a clearer view of ODCN's behavior using an appropriately scaled y-axis, revealing that its runtime only increases slightly as network congestion begins to manifest, before stabilizing at higher injection rates. Quantitatively, as the injection rate increases from 5% to 100%, Booksim's runtime rises by approximately 1153 seconds, representing a +425.9% increase. In contrast, ODCN's runtime grows by only 0.627 seconds over the same range, an increase of just +34.96%. This highlights a higher capability to scale on higher injection rates of ODCN compared to Booksim.

MF2 OpenDCN's higher level of abstraction enables it to simulate network traffic patterns on increasingly large topologies with significantly lower runtimes than Booksim, achieving speedup factors exceeding 700×.

MF3 OpenDCN's flow-level modeling enables efficient simulation across injection rates from 0.05% to 100%, with minimal runtime variation even on large-scale topologies.

5.3 Trace-Driven Evaluation of Datacenter Networks

In this section we perform trace-based network simulation in OpenDCN. Section 5.3.1 compares QoS and energy efficiency of three different widely used topologies while Section 5.3.2 investigates the simulator performance and scalability, compared to real world equivalent baseline.

5.3.1 Performance–Energy Trade-offs Across FatTree, Flattened Butterfly, and Dragonfly with Different Routing Protocols

To demonstrate the applicability of OpenDCN trace simulation for evaluating network topologies, we present a comparative study of three representative architectures: FatTree (FT), Flattened Butterfly (FF), and Dragonfly (DF). We evaluate these topologies in two main metrics: DSR and NPE (Section 2.5).

Experiment Setup. We evaluate FatTree (FT), Flattened Butterfly (FF) and Dragonfly (DF) topologies. These topologies are structurally distinct, yet they were configured to support a comparable number of terminals in order to ensure a fair evaluation. A summary of the chosen configuration parameters, along with the resulting number of vertices, terminals, and edges, is provided in Table 5.2. The selected designs vary by at most 5.6% in N. For routing, we adopt the RAND-MIN strategy, while all nodes, both switches and hosts, are configured with 1Gbps port speed. For measuring energy consumption, we adopt the datacenter network energy model presented by *Wang et al.* [63], which defines the power draw of the network as:.

$$\sum_{i=1}^{R} P_i^{switch} \tag{5.2}$$

where P_i^{switch} is the power draw of switch i and R is the total number of switches in the network.

$$P_i^{switch} = P_i^{chassis} + P_i^{cards} + \sum_{j=1}^{p} (P_{i,j}^{port_s} + P_{i,j}^{port_d}) \tag{5.3}$$

where $P_i chassis$ is the static power consumption of switch i (not including ports), p is the number of ports on switch i, and $P_{i,j}^{port_s}$ and $P_{i,j}^{port_d}$ are the static and dynamic power draw of port j on switch i respectively.

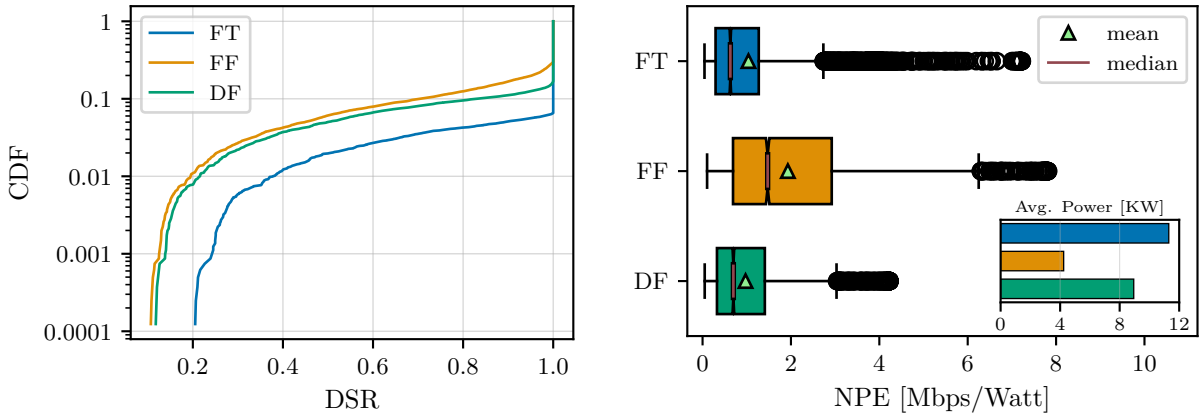
$$P_{i,j}^{port_d} = P_{i,j}^{port_f} * \frac{rx_{i,j} + tx_{i,j}}{bw_{i,j}^{link}} \tag{5.4}$$

where $rx_{i,j}$ and $tx_{i,j}$ are receiving and transmitting data rate of port j on switch i respectively, $bw_{i,j}^{link}$ is the max bandwidth of the link connected to port j on switch i, and $P_{i,j}^{port_f}$ is the dynamic power consumption the port in full link capacity.

The workload trace employed in this study is derived from the Bitbrains dataset[1] [55], which consists of 1250 VMs over a two-month period. Since the Bitbrains dataset does not expose host-VM mappings, we assume an average of 46 VMs per host, consistent with density distributions reported in industry studies [19]. We then synthetically generate a two-month trace for a system of 1000 hosts, which matches the minimum host count across the evaluated topologies.

To assess the impact of routing, we repeat the same experiment using the deterministic MIN strategy, which excludes multipath routing. Because results under MIN may vary depending on the specific choice of shortest paths at each node, we execute five independent runs per topology and report the averaged outcomes.

[1] *GWA-T-12 BitBrains dataset.* Available at: `https://atlarge-research.com/gwa-t-12/`.



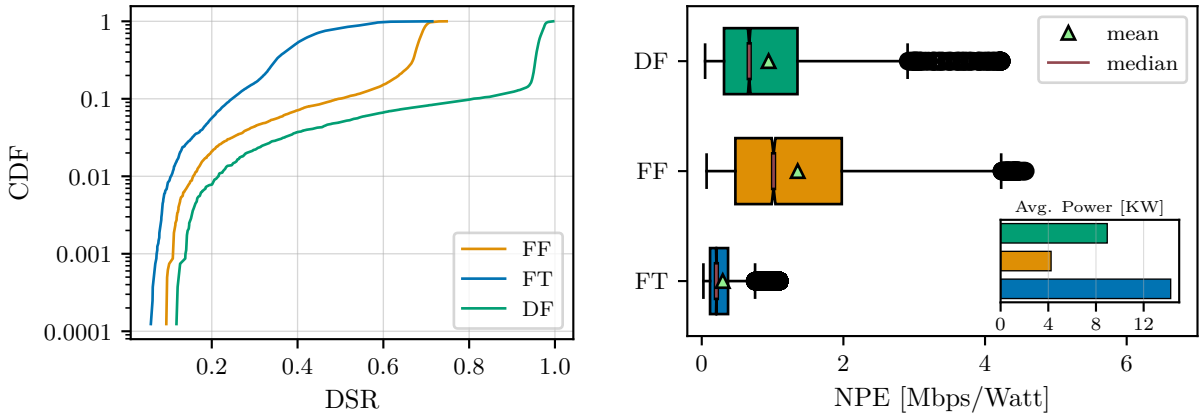
(a) DSR Cumulative Distribution Function (DSR). (b) Average NPE across simulation virtual time.

Figure 5.3: Performance comparison of different topologies under RANDMIN routing.

Experiment Results. Figure 5.3 reports the performance comparison of the three network topologies with RANDMIN.

Specifically, Figure 5.4a illustrates the complementary cumulative tail distribution of DSR. Under high congestion, FT topology demonstrates better performance, achieving nearly a 7% average DSR improvement compared to the other two topologies. In contrast, the FF and Dragonfly DF have highly similar distributions, with DF marginally outperforming FF by an average of 1.7%. Figure 5.4b presents the NPE of the three topologies measured at 5-minute intervals throughout the simulation. In terms of energy efficiency, the FF topology clearly dominates, achieving higher NPE values compared to both DF and FT. On average, FF provides a 108% improvement in mean NPE relative to DF, and a 219% improvement relative to FT.

These results exemplifies the trade-off between energy efficiency and performance: while FT employs over twice as many switches as FF, enabling it to sustain higher congestion levels, this design choice substantially increases the overall energy consumption of the network. Note that the measured power draw values are consistent with the topologies switch numbers and switch ports utilization, which were also captured during simulation (Section 4.3).



(a) DSR Cumulative Distribution Function (DSR). (b) Average NPE across simulation virtual time.

Figure 5.4: Performance comparison of different topologies under deterministic MIN routing.

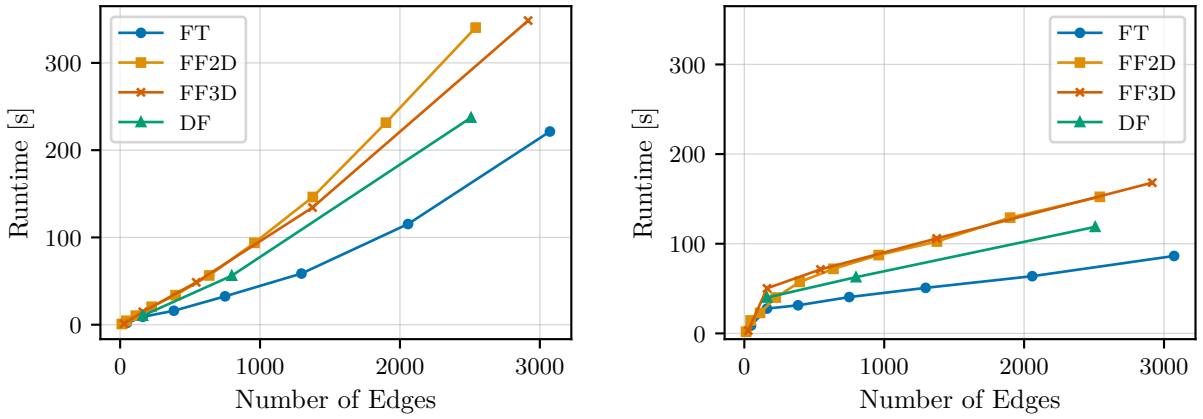
Figure 5.4 reports results with deterministic MIN routing. As expected, MIN increases link contention and reduces DSR. Compared to MIN, RANDMIN yields substantial average DSR gain: 148% for FT, 45% for FF, and 4.6% for DF, along with NPE gains of 207%, 42% and 3.1% respectively. Despite reduced DSR, FF retains the highest NPE average due to its low switch count, while DF approaches FF in efficiency under MIN due to its lower sensitivity to non-multipath routing.

MF4	FT sustains higher DSR under congestion, while FF achieves the highest energy efficiency. DF offers a balanced middle ground.
MF5	RANDMIN substantially improves both DSR and NPE compared to MIN, with FT benefiting the most from multipath routing and DF the least, showing that routing strategy strongly impacts topology performance.
MF6	OpenDCN enables comparison of topologies with realistic traces, capturing how topology and routing determine the trade-off between network performance and its energy consumption.

5.3.2 Simulator Performance and Scalability with Increasing Topology Size (Parallel vs. Single-Threaded

In this experiment we are going to evaluate the scalability of ODCN under a trace workload.

Experiment Setup. We evaluate scalability patterns across three topologies: FT with $k \in 4, 6, \ldots, 16$, FF with $n \in 2, 3$ and $k \in 2, 3, \ldots, 11$, and DF with $a \in 4, 6, 8$. The resulting properties N, V, R, and E, can be derived from Table 4.2. For FF, we distinguish between the two-dimensional and three-dimensional variants, denoted FF2D and FF3D, respectively. While ODCN supports arbitrary dimensionality for FF, this study restricts evaluation to a maximum of three dimensions. To further stress-test the simulator, we additionally include large-scale cases with FT with $k = 32$ and DF with $a = 14$, yielding $N = 8192$ and 9702, and $E = 24{,}576$ and 23,562, respectively. Workload traces are synthetically generated following the methodology of Section 5.3.1, producing two-month traces at 5-minute granularity. Unlike the earlier experiments, however, the number of hosts varies



(a) Scalability without parallelism.

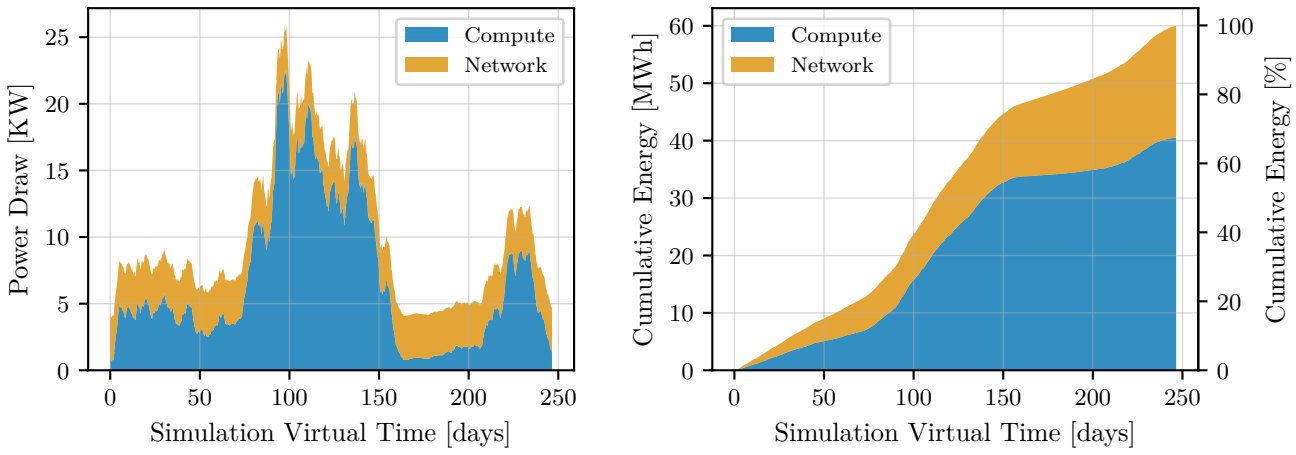
(b) Scalability with parallelism.

Figure 5.5: Trace-based OpenDCN network simulation scalability analysis.

with each topology's N, ranging from 8 to 1331, while the number of nodes V spans from 12 to 3060. Each topology is evaluated both with simulation parallelism enabled and with it disabled, except for the large-scale configurations, which are executed only with parallelism. We use E as the most indicative measure of network complexity. For every configuration, we perform five independent runs and report the average runtime.

Experiment Results. Figure 5.5 summarizes the performance results. Figure 5.5a presents the case without parallelism, where runtime exhibits exponential growth with respect to the number of edges, although with a relatively small exponent. Among the evaluated topologies, FF is the most costly to simulate, likely due to its low diameter combined with high concentration. Figure 5.5b shows results using parallelism. The results show a high overhead at low network sizes, while an almost sublinear scaling. Moreover, large scale FT with $k = 32$ was simulated in under 16 minutes, and DF with $a = 14$ in under 18 minutes. These results demonstrate that ODCN's parallelism capabilities improve scalability for large-scale topologies.

MF7	OpenDCN can simulate 2 months of workloads on topologies with ≈2000 nodes and 4000 links at 5 minute granularity in under 3 minutes, which corresponds to under 0.004% the real world equivalent.
MF8	OpenDCN can simulate 2 months of workloads on large scale topologies, with ≈10,000 nodes and 25,000 links at 5 minutes granularity in ≈17 minutes, which corresponds to less than 0.02% the real world equivalent
MF9	OpenDCN's parallelism capabilities enable scalability to large topologies, while the option to disable parallelism offers improved efficiency for small-scale networks.



(a) Power draw.

(b) Cumulative energy consumption.

Figure 5.6: Comparison of traffic pattern simulation results under incremental injection rates. Throughput is maximized when the accepted rate matches the offered load.

5.4 Accounting for the Network: Energy Attribution and QoS under Co-Simulation

In this section, we investigate combined network–compute simulations. We show additional information and analysis that are enabled by integrating networkign in OpenDC, in addition to all those things theat were already possible with OpenDC. Specifically, Section 5.4.1 presents a representative use case that quantifies and analyzes the correlation between computational demand, network utilization, and the associated energy consumption. Section 5.4.2 examines a scenario where network dependencies directly influence task execution dynamics, thereby affecting completion times and overall system performance. Together, these experiments provide insight into how tightly coupled compute and network behaviors shape both efficiency and scalability in distributed environments.

5.4.1 Energy Attribution in Co-Simulation (Network vs. Compute)

We evaluate network energy consumption compared to compute

Experiment Setup. This study uses one month of the Materna trace[2], which captures a cluster consisting of 49 hosts with a total of 69 physical CPU cores. Since the trace does not provide information about the underlying network topology, we model the infrastructure using a fat-tree topology with $k = 6$, supporting up to 54 hosts. We employ RANMIN routing and record results across five independent simulation runs, reporting the averages.

Experiment Results. Figure 5.6 summarizes the results. Figure 5.6a presents the temporal evolution of power consumption, showing the relative contributions of compute and network resources. As expected, compute power exhibits considerably higher variability, driven by workload dynamics, while network power remains mostly static. This follows from the network energy model: in the absence of energy-saving policies (e.g., switch power-down), switch power draw is dominated by

[2]GWA-T-13: https://atlarge-research.com/gwa-t-13/

Name	Symbol	Description
Submission time	t_{sub}	Time when the task is submitted.
Schedule time (comp)	t_{sch}^{comp}	Schedule time with compute-only sim.
Schedule time (comp-net)	t_{sch}^{net}	Schedule time with compute-network sim.
Finish time (comp)	t_{fin}^{comp}	Finish time with compute-only sim.
Finish time (comp-net)	t_{fin}^{net}	Finish time with compute-network sim.
Scheduling dur. (comp)	d_{sch}^{comp}	$t_{sch}^{comp} - t_{sch}$
Scheduling dur. (comp-net)	d_{sch}	$t_s^{net}ch - t_{sch}$
Execution dur. (comp)	d_{exe}^{comp}	$t_{fin}^{comp} - t_{sch}^{comp}$
Execution dur. (comp-net)	d_{exe}^{net}	$t_{fin}^{net} - t_{sch}^{net}$
Completion dur. (comp)	d_{tot}^{comp}	$t_{fin}^{comp} - t_{sub}$
Completion dur. (comp-net)	d_{tot}^{net}	$t_{fin}^{net} - t_{sub}$
Scheduling delay	Δd_{sch}	$d_{sch}^{net} - d_{sch}^{comp}$
Execution delay	Δd_{exe}	$d_{exe}^{net} - d_{exe}^{comp}$
Completion delay	Δd_{tot}	$d_{tot}^{net} - d_{tot}^{comp}$
Scheduling slowdown	s_{sch}	$d_{sch}^{net} \,/\, d_{sch}^{comp}$
Execution slowdown	s_{exe}	$d_{exe}^{net} \,/\, d_{exe}^{comp}$
Completion slowdown	s_{tot}	$d_{tot}^{net} \,/\, d_{tot}^{comp}$

t = timestamps, d = durations, Δd = delay of net-comp vs comp-only, s = slowdown net-comp vs comp-only.

Table 5.3: Notation for task time variables and derived durations, delays, and slowdowns under compute-only and compute-network scheduling.

static consumption, with utilization-dependent dynamic power contributing at most 5%.

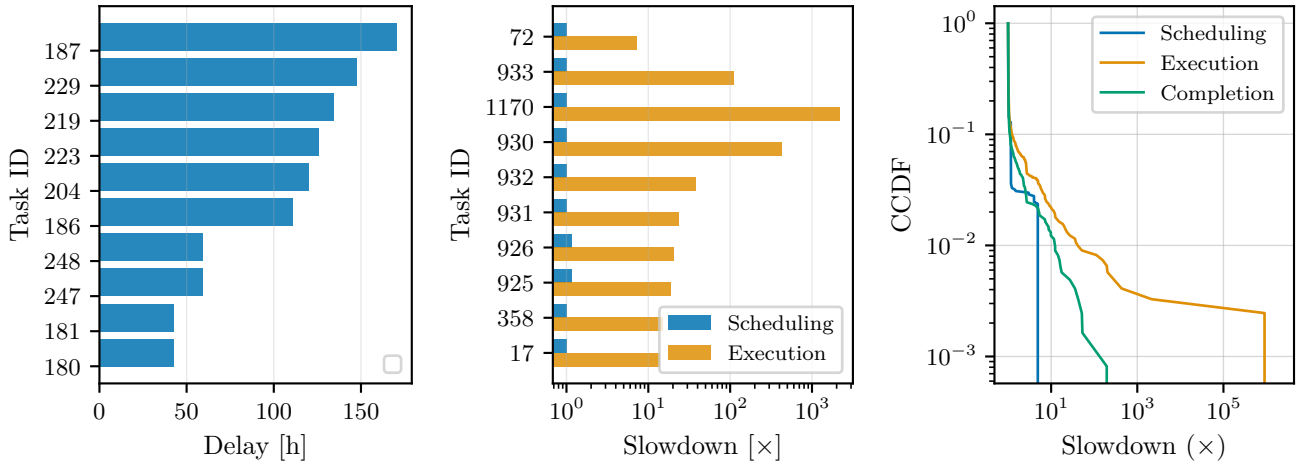
Figure 5.6b illustrates the cumulative energy consumption. Here, compute dominates overall consumption, although the persistent baseline of network power results in a non-negligible share of total energy use. Across the simulated period, the average network NPUE was 3.07, indicating that for every watt consumed by the network, roughly three watts were consumed by all IT resources.

MF10 In the Materna trace with a Fat-Tree topology ($k = 6$), compute power shows strong temporal variability, while network power remains mostly static due to the absence of energy-saving mechanisms.

MF11 In this scenario, compute dominates energy use, but the static baseline of network power amounts to an average NPUE of 3.07, about three watts of IT power for every watt consumed by the network.

5.4.2 Task QoS under Co-Simulation (Delays and Slowdowns)

In this experiment, we investigate the impact of jointly modeling compute and network resources on task performance. We employ the integrated capabilities of OpenDCN and OpenDC to simulate a scenario where task scheduling decisions are influenced not only by host-level compute constraints but also by network-level contention. Our objective is to quantify how incorporating network effects



(a) Top-10 tasks by scheduling delay, highlighting outliers.

(b) Slowdowns of the top-10 tasks by completion delay.

(c) CCDF of scheduling, execution, and completion slowdowns.

Figure 5.7: Delays and slowdowns of tasks in compute–network simulation relative to compute-only simulation

alters task scheduling latency, execution time, and overall completion time compared to a compute-only baseline. Table 5.3 summarizes the notation used throughout this experiment.

Experiment Setup.

We evaluate our approach using the Bitbrains fat-storage workload, which consists of approximately 1,250 virtual machines (tasks) submitted over the course of one month. This workload is simulated in OpenDC on a cluster of 128 hosts, under two conditions: (i) compute-only simulation, and (ii) joint compute–network simulation. For the network-enabled experiments, we employ a Fat-Tree topology with $k = 10$, resulting in $N = 128$ hosts. The network fabric is configured with 1 Gbps switches and minimal routing (MIN). We deliberately constrain the link speed to 1 Gbps to match the relatively low network demand of the Bitbrains workload (on average ≈ 40 KB/s per task) [54], enabling us to study a setting in which system performance may become network-constrained. Stakeholders with access to more network-intensive traces can readily reproduce the experiments with higher link speeds to study bandwidth-sensitive scenarios.

To evaluate the performance impact of incorporating network in the simulation model, we record timestamps (t) relative to the start of the virtual simulation time, such as submission, scheduling, and finish times, and derive durations (d) for scheduling, execution, and completion under both compute-only (d^{comp}) and joint compute–network (d^{net}) simulation, as defined in Table 5.3. Based on these durations, we compute relative performance metrics: delays (Δd), which quantify the difference between compute-only and compute–network runs, and slowdowns (s), which express their ratio, each for scheduling, execution, and completion.

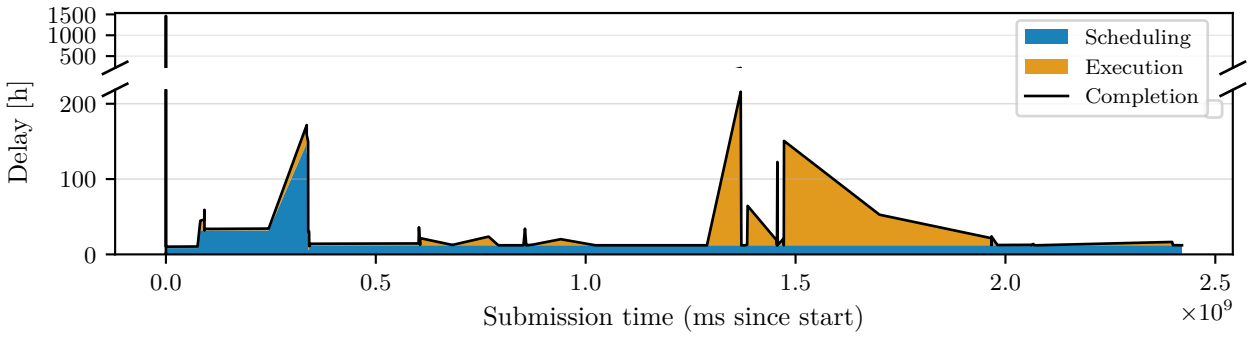


Figure 5.8: Delays distribution by task submission time.

Experiment Results. The results in Figures 5.7 and 5.8 show that incorporating network effects can substantially influence task performance.

We observe that a subset of tasks experiences severe scheduling delays, with the top-10 delayed tasks waiting over 150 hours before execution. When accounting for network contention, these tasks also exhibit pronounced slowdowns in both execution and completion time, with some slowdowns exceeding three orders of magnitude. The CCDF reveals that although most tasks are only slightly affected, a non-negligible part suffers huge penalties once network constraints are modeled.

Figure 5.8 shows the temporal distribution of delays relative to submission time. The results indicate bursts of congestion where tasks accumulate both scheduling and execution delays. Such bursts are hidden in compute-only simulations, but become visible when network behavior is incorporated.

These findings highlight that neglecting network effects can lead to overly optimistic estimates of task performance. In particular, while average behavior appears similar across compute-only and compute–network simulations, the tail behavior (i.e., the worst-affected tasks) diverges. This distinction is critical for datacenter operators and system designers: understanding tail performance is essential for ensuring QoS and capacity planning.

MF12	In the Bitbrains fast-storage workload on a Fat-Tree topology ($k = 10$, 1 Gbps), network contention has execution and completion slowdowns compared to a compute-only model by up to three orders of magnitude for a subset of tasks.
MF13	Under the same scenario, a small fraction of tasks experiences extreme scheduling and completion delays (over 150 hours), which remain hidden in compute-only simulations.
MF14	Omitting network effects leads to overly optimistic performance estimates. While averages appear similar, the extremes diverge, making network modeling essential to capture tail behavior.

6

Conclusions and Future Work

6.1 Conclusion

In this work we investigated how to model network resources at a mid-to-high level of abstraction, enabling discrete event simulation of network and compute-network workloads with OpenDC(N), while providing evidence of the importance of network in datacenter simulation through a series of experiments (MRQ). In Chapter 1 we described the societal relevance of datacenter and the ability to perform accurate simulations. We identified the lack of existing tools that fully incorporate network models in datacenter simulation, while supporting independent simulation of both compute and network, and the importance of interactivity in simulation. In Chapter 2 we provided relevant background fro understanding the complexity of datacenter networks and how to evaluate them. In Chapter 3 we illustrated the high level architecture of the simulator and how we modeled network resources. In Chapter 4 we briefly explained the implementation of OpenDCN and all the supported simulations. In Chapter 5 we conducted experiments, validating our network model and providing a series of use cases and valuable insights on the importance of network simulation in datacenters. We now answer each research question fully.

RQ1 How to model network resources at a mid-level-abstraction across datacenters and generic network topologies, enabling trace-based, traffic-pattern-based and interactive network simulation?

In Chapter 3 we derived design requirements (Section 3.1) which guided our design process. We presented a high level architecture together with our flow-level network traffic model. We specified how congestion is detected and handled during simulation and enumerated the discrete events that drive the model, while introducing the concept of network stability for possibly asynchronous injection of network events. We later described the implementation and configuration of the three network simulation modes: trace-based, traffic-pattern, and interactive.

RQ2 How can a compute-centric simulator be extended with a network layer to support both joint compute–network co-simulation and standalone network and compute analysis?

In Chapter 4 we illustrate how OpenDCN is integrated in OpenDC, following state-of-the-art software engineering practices, with well define touch points. This preserves the ability to run independent simulation of both compute and network while enabling combined compute-

Project	Env.	Stakeholders	Compute	Network	Interactive	Traffic Pattern	Co-Sim.
CloudSim [13]	Cloud, Edge, Fog	Research	✓	✓(coarse)	✗	✗	✓
SimGrid [14]	Grid, Cloud, P2P	Research, Edu.	✓	✓(coarse)	✗	✗	✗
DGSim [31]	Grid	Research	✓	✗	✗	✗	✗
GreenCloud [37]	Cloud, DCN	Research	✓	✓	✗	✓	✓
ICanCloud [47]	Cloud	Research, Edu.	✓	✓(coarse)	✗	✗	✗
Edge-CloudSim [58]	Edge, Cloud	Research	✓	✓(coarse)	✗	✗	✓
NS-3 [28]	Network	Research	✗	✓	✗	✓	✗
OMNET++ [61]	Network	Research	✗	✓	✓	✓	✗
OpenDC(N) (this work)	Cloud, DCN, Network	Research, Edu.	✓	✓	✓	✓	✓

Network column: ✓= network simulation, including explicit topology, node, routing, and congestion modeling; ✓(coarse) = analytical bandwidth–delay abstraction, no flows; ✗= no network support.

Table 6.1: Comparison of selected datacenter and network simulators.

network simulation. We provided information on how to configure such combined experiments, and later on shown some real world applications.

RQ3 How does datacenter networking affect the performance and overall system behavior on datacenter workloads?

In Chapter 5 we investigate the benefits of including network in datacenter simulation. First we validated our model against a peer reviewed network simulator. Then, we analyzed the impact that topologies and routing protocols have on energy efficiency and QoS in datacenter networks, confronting three widely used topologies under a realistic datacenter network workload. Finally, we performed compute-network co-simulation, demonstrating how the inclusion of network in a more comprehensive scenario drastically changes system behavior, energy consumption and QoS, especially in the tail relative to a compute only baseline.

We position OpenDC(N) within the existing landscape. Table 6.1 compares selected datacenter and network simulator based on target environment, stakeholders, network support, interactivity, traffic pattern capabilities and compute-network co-simulation. As summarized, existing tools tend to be compute-centric with missing or coarse network support, network centric with missing compute support, or lacking important network simulation capabilities. OpenDC(N) combines mid-to-high level network modeling, interactivity, traffic pattern support, and integration with OpenDC for co-simulation, thereby bridging this gap.

6.2 Future Work

We envision 2 main areas of future research and development, building upon our contributions from this work.

1. *Expand the variety of built in components:* We plan to expand OpenDCN's catalog of built-in components, adding additional DCN topologies (e.g., Clos variants, Leaf–Spine, BCube/DCell, Dragonfly), routing families (UGAL, adaptive), traffic generators (permutations, hotspots, incast), energy models (switch sleep states, rate adaptation, calibrated power curves), and QoS policies (priority, rate limiting). Each addition would further enhance OpenDCN ease of adoption, especially for datacenter operators and educational purposes.

2. *Educational activities and workshops:* The REPL environment of OpenDCN facilitates *interactive* learning and exploration: live topology construction, fault/traffic injection, step-through execution, and metric visualization (e.g., paths, link utilization). We envision OpenDCN as a tool used in education environment with REPL-driven labs, assignment and instructor demos.

Bibliography

[1] ACM artifact review and badging. `https://www.acm.org/publications/policies/artifact-review-and-badging-current`.

[2] GNS3: Graphical Network Simulator-3. `https://www.gns3.com/`. Accessed: 2025-07-20.

[3] Nokia the global network traffic report. `https://onestore.nokia.com/asset/21366`.

[4] Statista volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025. `https://www.statista.com/statistics/871513/worldwide-data-created`.

[5] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 51–62, 2010.

[6] M. A. Al-Fares. *A scalable, adaptive, and extensible data center network architecture*. University of California, San Diego, 2012.

[7] M. Alizadeh and T. Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st Annual Symposium on High-Performance Interconnects*, pages 71–74, 2013.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[9] C. L. Belady and C. G. Malone. Metrics and an infrastructure model to evaluate data center efficiency. In *International Electronic Packaging Technical Conference and Exhibition*, volume 42770, pages 751–755, 2007.

[10] F. Benamrane, R. Benaini, et al. An east-west interface for distributed sdn control plane: Implementation and evaluation. *Computers & Electrical Engineering*, 57:162–175, 2017.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[12] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *SC'14: proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 348–359. IEEE, 2014.

[13] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

[14] H. Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437. IEEE, 2001.

[15] T. Daim, J. Justice, M. Krampits, M. Letts, G. Subramanian, and M. Thirumalai. Data center metrics: An energy efficiency model for information technology managers. *Management of Environmental Quality: An International Journal*, 20(6):712–731, 2009.

[16] W. J. Dally and B. P. Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[17] M. Dayarathna, Y. Wen, and R. Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys Tutorials*, 18(1):732–794, 2016.

[18] P. Dechamps. The iea world energy outlook 2022 – a brief analysis and implications. *The European Energy and Climate Journal*, 11(3):100 – 103, 2023.

[19] F. Denneman. Insights into vm density. `https://frankdenneman.nl/2016/02/15/insights-into-vm-density/`, 2016.

[20] C. Fiandrino, D. Kliazovich, P. Bouvry, and A. Y. Zomaya. Performance metrics for data center communication systems. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 98–105, 2015.

[21] C. Fiandrino, D. Kliazovich, P. Bouvry, and A. Y. Zomaya. Performance and energy efficiency metrics for communication systems of cloud computing data centers. *IEEE Transactions on Cloud Computing*, 5(4):738–750, 2017.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[23] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review*, 39:51–62, 01 2011.

[24] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 63–74, 2009.

[25] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 75–86, 2008.

[26] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[27] A. Hammadi and L. Mhamdi. A survey on architectures and energy efficiency in data center networks. *Computer Communications*, 40:1–21, 2014.

[28] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 14(14):527, 2008.

[29] A. Iosup. Massivizing computer systems. Keynote Presentation, 2021. Available online at: `https://atlarge-research.com/pdfs/pres-20210204-aiosup-massivizing.pdf`.

[30] A. Iosup, G. Andreadis, V. Van Beek, M. Bijman, E. Van Eyk, M. Neacsu, L. Overweel, S. Talluri, L. Versluis, and M. Visser. The opendc vision: Towards collaborative datacenter simulation and exploration for everybody. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 85–94. IEEE, 2017.

[31] A. Iosup, O. Sonmez, and D. Epema. Dgsim: Comparing grid resource management architectures through trace-based simulation. In *European Conference on Parallel Processing*, pages 13–25. Springer, 2008.

[32] T. Y. James. Applying mininet for network education.

[33] N. Jiang, G. Michelogiannakis, D. Becker, B. Towles, and W. J. Dally. Booksim 2.0 user's guide. *Standford University*, page q1, 2010.

[34] K. Kant. Data center evolution: A tutorial on state of the art, issues, and challenges. *Computer Networks*, 53(17):2939–2965, 2009.

[35] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.

[36] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.

[37] D. Kliazovich, P. Bouvry, and S. U. Khan. Greencloud: a packet-level simulator of energy-aware cloud computing data centers. *The Journal of Supercomputing*, 62(3):1263–1283, 2012.

[38] K. Lakhotia, M. Besta, L. Monroe, K. Isham, P. Iff, T. Hoefler, and F. Petrini. Polarfly: a cost-effective and flexible low-diameter topology. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[39] B. Lantz, B. Heller, and N. McKeown. Mininet: An instant virtual network on your laptop. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6. ACM, 2010.

[40] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[41] D. Li, C. Guo, H. Wu, K. Tan, Y. Zhang, and S. Lu. Ficonn: Using backup port for server interconnection in data centers. In *IEEE INFOCOM 2009*, pages 2276–2285. IEEE, 2009.

[42] Y. Liu, X. Wei, J. Xiao, Z. Liu, Y. Xu, and Y. Tian. Energy consumption and emission mitigation prediction based on data center traffic and pue for global data centers. *Global Energy Interconnection*, 3(3):272–282, 2020.

[43] C. Malone and C. Belady. Metrics to characterize data center & it equipment energy use. In *Proceedings of the Digital Power Forum, Richardson, TX*, volume 35, 2006.

[44] F. Mastenbroek, G. Andreadis, S. Jounaid, W. Lai, J. Burley, J. Bosch, E. Van Eyk, L. Versluis, V. Van Beek, and A. Iosup. Opendc 2.0: Convenient modeling and simulation of emerging tech-

nologies in cloud datacenters. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 455–464. IEEE, 2021.

[45] F. Mastenbroek, T. D. Matteis, V. van Beek, and A. Iosup. Radice: A risk analysis framework for datacenters. *IEEE Transactions on Cloud Computing*, 2023.

[46] M. Noormohammadpour and C. S. Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525, 2017.

[47] A. Núñez, J. L. Vázquez-Poletti, A. C. Caminero, G. G. Castañé, J. Carretero, and I. M. Llorente. icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[48] G. Ostrouchov. Parallel computing on a hypercube: an overview of the architecture and some applications. In *Computer Science and Statistics, Proceedings of the 19th Symposium on the Interface*, pages 27–32, 1987.

[49] O. Popoola and B. Pranggono. On energy consumption of switch-centric data center networks. *J. Supercomput.*, 74(1):334–369, jan 2018.

[50] V. D. Reddy, B. Setz, G. S. V. R. K. Rao, G. R. Gangadharan, and M. Aiello. Metrics for sustainable data centers. *IEEE Transactions on Sustainable Computing*, 2(3):290–303, 2017.

[51] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The evolution of data to life-critical. don't focus on big data. *2*, 2017.

[52] Y. Shang, D. Li, J. Zhu, and M. Xu. On the network power effectiveness of data center architectures. *IEEE Transactions on Computers*, 64(11):3237–3248, 2015.

[53] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, pages 1–13, 2016.

[54] S. Shen, V. van Beek, and A. Iosup. Workload characterization of cloud datacenter of bitbrains. Technical report pds-2014-001, Delft University of Technology, Parallel and Distributed Systems Section, Delft, The Netherlands, Feb. 2014.

[55] S. Shen, V. Van Beek, and A. Iosup. Statistical characterization of business-critical workloads hosted in cloud datacenters. In *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*, pages 465–474. IEEE, 2015.

[56] A. Singh, W. J. Dally, B. Towles, and A. K. Gupta. Locality-preserving randomized oblivious routing on torus networks. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 9–13, 2002.

[57] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[58] C. Sonmez, A. Ozgovde, and C. Ersoy. Edgecloudsim: An environment for performance evaluation of edge computing systems. *Transactions on Emerging Telecommunications Technologies*, 29(11):e3493, 2018.

[59] Statista. Data center average annual pue worldwide, 2023. Accessed: 2023-10-01.

[60] L. G. Valiant. A scheme for fast parallel communication. *SIAM journal on computing*, 11(2):350–361, 1982.

[61] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–10, 2008.

[62] T. Wang, Z. Su, Y. Xia, B. Qin, and M. Hamdi. Novacube: A low latency torus-based network architecture for data centers. In *2014 IEEE Global Communications Conference*, pages 2252–2257. IEEE, 2014.

[63] X. Wang, Y. Yao, X. Wang, K. Lu, and Q. Cao. Carpo: Correlation-aware power optimization in data center networks. In *2012 Proceedings IEEE INFOCOM*, pages 1125–1133. IEEE, 2012.