University of Amsterdam

Vrije University Amsterdam

Master Thesis

Future Network Services with Next-Generation Network Digital Twins: Architectural Framework and Tool Evaluation

---

Author: Zhiheng Yang (UvA: 14483262 VU: 2773738)

1st supervisor: Dr. Adam S.Z. Belloum
2nd reader: Dr. Chrysa Papagianni

A thesis submitted in fulfillment of the requirements for

the joint UvA-VU Master of Science degree in Computer Science

August 30, 2024

"With great power comes great responsibility",
by Uncle Ben

Abstract

In the context of 6G, Digital Twin technology has potential to play a crucial role by offering a sophisticated, dynamic model that mirrors the physical world in real-time. Integrating digital twins into 6G enables precise orchestration and optimization of network resources, meeting the diverse and stringent demands of next-generation applications. Furthermore, digital twins provide a platform for continuous learning and AI-driven insights. Research on the architecture of Network Digital Twins (NDTs) for 5G/6G is still limited, often focusing on partial implementations rather than comprehensive, full-stack approaches. This paper proposes a high-level architectural framework for 6G NDTs, structured across three layers: the Physical Twin Layer, the Digital Twin Layer, and the Application and Service Layer. We discuss the challenges of deploying these systems and the importance of selecting appropriate tools, presenting an experimental case study focusing on network simulators and emulators, including NS-3, OMNet++, GNS3, and Mininet. Our comparative analysis demonstrates clear differences in performance overhead, with some tools showing markedly better efficiency in resource utilization than others, even times, to achieve the same goal. Our findings also reveal that different network simulators and emulators exhibit varying levels of sensitivity to network changes and have different applicability.

Acknowledgements

This research was conducted under the auspices and with the generous support of the Future Network Services.

I would like to extend my sincere thanks to Dr. Adam S.Z. Belloum for his continuous guidance and mentorship, which were crucial in laying the groundwork for this study. I am also deeply grateful to Dr. Chrysa Papagianni for her insightful advice and unwavering support, even during her busy schedule and holidays. I am genuinely excited that we can continue our collaboration in the future. I have also met many friends and scholars in the Multiscale Networked Systems research group. I am deeply thankful for the opportunity to interact with such a vibrant community of outstanding researchers. This environment has provided me with valuable experience beyond the scope of academic research. The conducive atmosphere and environment at Science Park greatly contributed to my work, making it an ideal place for research. My family, and some old friends, they all mentally supported me in this progress. I would even like to express my appreciation for the weather in Amsterdam — remarkably, it didn't rain every day.

I am grateful to all those who have contributed to this journey and made this unforgettable chapter of my future career possible.

Contents

List of Figures

LIST OF FIGURES

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

1.1 Context

The concept of Digital Twins (DTs) has seen significant growth in recent years, evolving from its initial introduction at a product lifecycle management conference to becoming a critical technology across various industries, particularly in manufacturing and aerospace. Despite its widespread adoption in these areas, the application and evaluation of DTs within network systems remain relatively underdeveloped. As the telecommunications industry gears up for 6G networks—offering unprecedented improvements in speed, capacity, and latency—the relevance of Network Digital Twins (NDTs) comes into focus (1).

NDTs serve as sophisticated virtual replicas of network setups and operational strategies, pivotal in enhancing real-time monitoring, predictive analytics, and pre-implementation simulations of network changes. Research on DTs in the network has just begun, and its application is still in the infancy stage (2), particularly in the context of 6G networks.

Many of the cases and studies that claim to introduce NDTs in networks, such as 5G, focus on partial twining, for example, focusing on the RF part or the network management part (3). Therefore, there is a lack of full-stack NDTs and its overall architecture, especially for the next-generation network. To address these gaps, we propose a high-level architectural framework for the 6G Network Digital Twins (6G NDTs) system, consisting of three key layers: the Physical Twin Layer, the Digital Twin Layer, and the Application and Service Layer. We also explore potential strategies for deploying 6G NDTs system and emphasize the importance of selecting appropriate tools for their implementation.

Network simulation and emulation tools can enable the accurate modeling, testing, and validation of network behaviors and configurations in a controlled and repeatable environment, which is potentially useful in the research environment. To recognize the critical

applicability of these tools in our NDT system, we also benchmarked and evaluated several network simulators and emulators. This benchmark study assesses various performance metrics of NS-3, OMNeT++, Mininet, and GNS3. In performance overhead evaluation, it covers CPU usage and memory requirements; in applicability evaluation, the extensibility, scalability, and modularity are also included. By meticulously evaluating these tools, the study identifies each tool's specific strengths and weaknesses in different aspects, providing possible solutions or suggestions for developing 6G NDT system components.

1.2 Research Questions

Below are our key research questions guiding the development and evaluation of our 6G network digital twin framework:

- **RQ1:** What high-level architecture should be designed for a 6G network digital twin system to better support its intended functions?
- **RQ2:** Which network simulation and emulation tools best suit this 6G NDTs system architecture:
  - in the aspect of performance overhead?
  - in the aspect of fidelity and extensibility?
  - in the aspect of flexibility and rapid scalability for dynamic deployment?

1.3 Main Contributions

This thesis makes several contributions to the field of digital twins within the context of 6G networks, which are detailed as follows:

- A propose of a high-Level 6G NDTs system architecture:
  - Physical Twin Layer: covers the physical twins that can abstracted and modeled.
  - Digital Twin Layer: implements computational models that facilitate advanced simulations and predictions to enhance network management and operations, and maintain synchronized real-time data between the physical network components and their digital counterparts.
  - Application and Service Layer: deploys network services that optimize operations using insights from the digital twins.

- An evaluation of Network Simulation and Emulation Tools:
  - Principles and Suitability: Delivers an examination of the underlying principles and mechanisms of network simulators and emulators to determine their advantage in the NDTs architecture.
  - Overhead benchmark: Detailed performance analysis of NS-3, OMNeT++, Mininet, and GNS3 with a focus on CPU usage and memory requirements.
  - Performance Results: Provides comparative insights that assist in selecting the most suitable simulation and emulation tools for specific network scenarios.

1.4 Thesis Structure

The structure of this thesis is as follows::

Chapter 1 sets the stage by discussing the relevance of NDTs and outlines the research questions that guide the study. Chapter 2 provides the necessary theoretical foundation, detailing the evolution of digital twin technology and next-generation networks, as well as an overview of some existing network simulation and emulation tools. Chapter 3 introduces a proposed high-level architecture for 6G NDTs and discusses the tool requirements essential for this architecture. Chapter 4 details the experimental setups, including network topology configurations and the performance metrics used for evaluation. Chapter 5 elaborates on the experimental results, offering an understanding of the performance, efficiency, and characteristics of various simulation tools that are possible to be integrated with the NDTs architecture. Chapter 6 critically analyze the findings and potential biases in the study , and finally, Chapter 7 encapsulates the study's contributions and potential directions for future research.

2

Background and Related Work

2.1 Digital Twins

The concept of the digital twin was first introduced by Michael Grieves during a Product Lifecycle Management conference in 2002 (4). He proposed it as a virtual counterpart to the physical world, presenting an innovative approach to model systems and processes across their lifecycle (5, 6). The aerospace sector took the lead to implement this concept in practical applications. The U.S. Air Force and NASA have invested in exploring how DT can be utilized to manage space assets and training flights (7). In fact, ideas similar to DT had already appeared in NASA before this concept was proposed (8). Now the technology has expanded to include manufacturing (9, 10), healthcare (11, 12), the automotive industry(13), urban planning (14), IoT systems (15), aerospace (9), and the development of 5G/6G networks. Each sector benefits from the tailored application of digital twins to optimize operations and enhance predictive analytics.

2.1.1 Network Digital Twins

As the telecommunications industry anticipates the rollout of 5G/6G, which promises substantial improvements in speed, capacity, and latency, the significance of Network Digital Twins (NDTs) comes into sharper focus (1). NDTs are virtual replicas of physical network systems, used to simulate and optimize network operations in real-time. These NDTs are sophisticated virtual replicas of network setups and operational strategies, pivotal in enhancing real-time monitoring, predictive analytics, and pre-implementation simulations of network changes. The enhancement of these capabilities is crucial for meeting the demands of modern telecommunications, as shown by studies from (16) and (17). Research on DTs in the network has just begun, and its application is still in the infancy stage(2). Wireless

communication can be the next scenario where DTs can be applied to fully realize their functions (18). As a new paradigm, 6G holds significant research and application value when combined with DTs(19).

2.2 Next-generation Network

Next-generation networks refer to the evolving telecommunications infrastructures that offer enhanced capabilities, such as higher data rates, lower latency, increased reliability, and greater scalability compared to previous generations. These networks can now support a wide range of advanced applications, including IoT, autonomous vehicles, and immersive experiences. This evolution is exemplified by the advancements from 5G to 6G.

2.2.1 5G Network

5G networks represent the fifth generation of mobile network technology, designed to provide significantly higher data rates, reduced latency, greater capacity, and enhanced reliability compared to previous generations. Technically, 5G operates across three frequency bands: low-band, mid-band, and high-band (millimeter-wave). Each band offers different trade-offs in terms of coverage, speed, and capacity.

Low-band 5G operates in frequencies below 1 GHz, providing broad coverage and penetration through obstacles but with relatively lower data speeds, comparable to enhanced 4G LTE. Mid-band 5G, operating between 1 GHz and 6 GHz, balances coverage and speed, offering faster data rates while maintaining decent coverage. High-band 5G, also known as mmWave, operates at frequencies above 24 GHz, delivering extremely high data rates (up to several Gbps) with very low latency, but it has limited range and struggles with physical obstructions.

5G networks employ advanced technologies such as Massive MIMO (Multiple Input, Multiple Output), which uses a large number of antennas to increase capacity and coverage, and beamforming, which directs signals more precisely towards users to improve performance and efficiency (16). Network slicing is another key feature, allowing operators to create multiple virtual networks within a single physical 5G infrastructure (20), tailored to different use cases such as IoT, autonomous vehicles, and enhanced mobile broadband (21, 22).

2.2.2 6G Network

6G networks represent the sixth generation of mobile network technology, designed to surpass 5G by delivering even higher data rates, ultra-low latency, greater capacity, and enhanced reliability with more advanced features. Technically, 6G is expected to operate across a broader range of frequency bands, including sub-terahertz and terahertz frequencies, pushing the boundaries of speed and connectivity (23). 6G will incorporate cutting-edge technologies such as Reconfigurable Intelligent Surfaces , which can manipulate electromagnetic waves to enhance signal strength and coverage, and AI-driven network management, allowing real-time optimization of network resources and services (23, 24). Quantum communication is also a potential feature, promising ultra-secure data transmission.

Building on 5G's network slicing, 6G is possible to offer even more granular and dynamic slicing capabilities, enabling the creation of highly specialized virtual networks for diverse applications like holographic communication, immersive extended reality, and advanced IoT (25).

2.3 Discrete Event Simulation

Discrete Event Simulation (DES) is a modeling technique used to represent and analyze systems where changes occur at specific points in time. Unlike continuous simulations, where changes happen continuously over time, DES focuses on the occurrence of distinct events that cause the system's state to change (26).

In DES, the system is modeled as a sequence of events, each of which occurs at a particular time and triggers changes in the state of the system. These events are discrete, meaning they happen at specific moments rather than continuously. The simulation clock advances in discrete steps from one event to the next, rather than continuously. It also incorporates stochastic elements, allowing it to model the randomness inherent in many real-world systems. This makes DES particularly effective for simulating processes like queuing systems, manufacturing lines, or network traffic, where events such as arrivals, departures, or failures occur at random intervals.

This contrasts with emulators, where the goal is to reproduce the exact timing and behavior of a system as closely as possible, often requiring real-time processing. The key difference lies in their approaches: DES is used for scenarios where the focus is on understanding the system's behavior over time and analyzing the impact of different events on the overall system. Emulators, on the other hand, are typically used when the goal is

to replicate the exact functioning of a system, such as in testing software or hardware in a controlled environment. However, DES approach is also considered as a useful approach to model digital twins, especially in industry field, for example, Logistics (27).

The process of the discrete event simulation is shown in Figure 2.1.

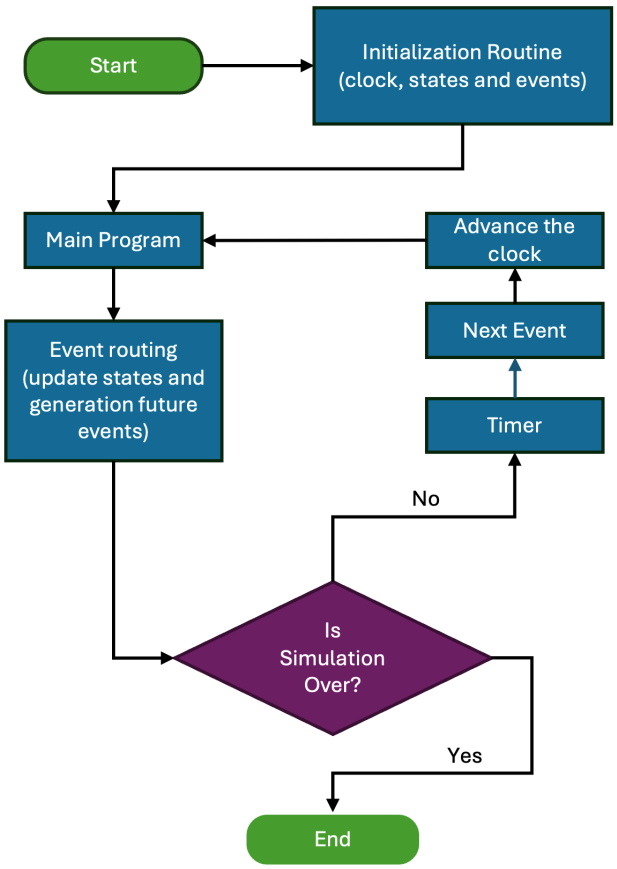

Figure 2.1: Discrete event simulation flow diagram

2.4 Network Namespace

Network namespaces in Linux provide a way to isolate network resources at the process level, enabling the creation of separate and independent network stacks within the same system. Each namespace has its own network interfaces, IP addresses, routing tables, firewall rules, and ports, which function independently from those in other namespaces. This means that processes in one namespace can operate with a completely different network configuration from those in another, without any interference (28).

Network namespaces are commonly used in containerization technologies like Docker and Kubernetes to provide isolated networking environments for containers. This isolation allows different containers to have their own network interfaces and IP addresses, creating a secure and scalable multi-tenant environment.

2.5 Virtual Network Interface

A Virtual Network Interface (VNI, or VNIC) is a software-based abstraction of a physical network interface card (NIC). It functions like a traditional network interface but is implemented in software, which allows it to be used in virtualized environments, such as virtual machines (VMs), containers, or other network emulation tools.

VNIs are essential in scenarios where physical network interfaces are either insufficient or unnecessary. They enable the creation of multiple isolated networking environments on a single physical machine, with each VNI behaving as though it is a separate physical NIC. This allows for flexible networking configurations, such as assigning different IP addresses, routing rules, and network policies to different virtual interfaces within the same host.

VNIs are commonly used in virtual networking setups, including virtual private networks (VPNs), containerized applications (e.g., Docker), and network emulators.

2.6 Network Simulator

A Network Simulator is a software tool designed to replicate the behavior of network systems through the use of mathematical models. It allows researchers and developers to design, evaluate, and analyze the performance of networks in a controlled and repeatable environment. The key feature of a network simulator is its ability to mimic the behavior of network devices and protocols without requiring physical hardware. Network simulators can model a wide range of network scenarios, including traffic flow, protocol interactions, and the impact of different network topologies. The use of a network simulator is particularly valuable in situations where real-world experimentation would be too costly, time-consuming, or impractical. It operates in a virtual space, where time can be manipulated to speed up or slow down the simulation process, thus providing insights into network performance under various conditions.

2.6.1 NS-3

Network Simulator 3, NS-3, is an open-source discrete-event network simulator widely used in academic settings to simulate the behavior research environment of wired networks (29).

It is written in C++ with optional Python bindings, allowing for high-fidelity simulation with flexibility in scripting (30).

It also has serval extensions for wireless network systems. It can provide a comprehensive set of tools to model, simulate, and analyze a wide range of network protocols and configurations.

NS-3 is designed to be modular, making it possible to extend and customize simulations by incorporating new models or modifying existing ones (31).

NS-3's architecture is composed of several layers that reflect the structure of real-world networking. The core components include:

- **Node:** Represents a network device, such as a computer or router, that contains network stacks and applications.
- **NetDevice:** Simulates network interfaces, such as Ethernet or Wi-Fi adapters.
- **Channel:** Models communication mediums between nodes, supporting both wired and wireless channels.
- **Application:** Represents user-level processes that generate and consume network traffic, such as HTTP clients and servers.

One of NS-3's significant features is its support for *real-time simulation*, where NS-3 can interact with live network interfaces and real-world traffic. This capability enables hybrid simulations that combine simulated and real-world network components. To integrate with a real network stack and emit/consume packets, we can use a real-time scheduler to lock the simulation clock with the hardware clock Furthermore, NS-3 includes a visualization tool called *NetAnim*, which provides a graphical representation of network topology and packet flows during simulation (32).

2.6.2 OMNet++

OMNeT++ is simulier to NS-3, which is a versatile, modular, and component-based C++ simulation framework primarily used for simulating networks, including wired, wireless, and mobile networks. It is also a discrete-event simulator (33). It is an open-source, extensible, and highly customizable tool widely adopted in academic and industrial research

for developing and analyzing network protocols, communication systems, and distributed systems (29).

OMNeT++ is distinguished by its modular architecture, which allows users to design complex systems by assembling reusable components. These components, known as modules, can be either simple (written in C++) or composite (constructed from other modules). The framework's flexibility is further enhanced by its support for hierarchical modeling, enabling users to build models with varying levels of abstraction (29, 34).

OMNeT++ is built around several core components:

- **Simulation Kernel:** This is the core engine of OMNeT++, responsible for event scheduling, execution of simulation runs, and managing the interactions between modules. The kernel ensures that events are processed in the correct order and that the state of the simulation is consistent throughout the run.

- **Network Description Language (NED):** NED is a specialized language used in OMNeT++ for defining the structure of the network models. It describes the connections between modules and the parameters that control their behavior. NED supports both graphical and textual editing, allowing for intuitive model creation and modification.

- **Simulation Models (NetDevice):** These are the user-defined modules and networks constructed using the NED language and C++. Simulation models can represent various network entities, such as routers, switches, and protocols, enabling detailed and accurate representation of communication systems.

- **INET Framework:** INET is a widely used model library that extends OMNeT++ with models for Internet protocols, wired and wireless networks, mobility, and other communication-related components. It serves as a foundational building block for many OMNeT++ simulations, providing ready-to-use implementations of commonly used protocols and network components. However, it is an extension of OMNet++ and requires additional installation and configuration.

- **Visulization and Analysis Tools:** OMNet++ provides IDE for simplifying the configuration and visualizing simulation results. It also offers integrated support for plotting, charting, and statistical analysis.

2.7 Network Emulator

A Network Emulator, in contrast to a network simulator, provides a real-time environment that replicates the conditions of an actual network. It allows real network hardware and software to interact with the emulated network, offering a high degree of realism. Unlike simulators, which operate in a purely virtual environment, emulators interface with physical devices, making them suitable for testing the performance of systems in realistic scenarios where hardware interactions are critical.

2.7.1 GNS3

GNS3 (Graphical Network Simulator-3) is an open-source network software emulator that allows users to design, test, and troubleshoot complex networks with a high degree of flexibility and realism (35). Besides, GNS3 is distinguished by its ability to emulate real network hardware, enabling users to run actual Cisco IOS images and other network operating systems within a virtual environment. This capability offers a more authentic simulation experience compared to traditional packet-based simulators, allowing for the testing of real-world scenarios, including the behavior of routing protocols, network configurations, and device performance under different conditions. Additionally, GNS3 supports a wide range of vendors beyond Cisco, including Juniper, Arista, and others, through its integration with QEMU and VirtualBox.

GNS3 is composed of several interconnected components:

- **GNS3 GUI**: This is the graphical user interface that users interact with. It exposes ports and allows the design and management of network topologies by providing a drag-and-drop environment for adding devices, configuring connections, and controlling simulations.
- **GNS3 Server**: The server acts as the backend, responsible for handling the emulation of devices and network traffic. It can be run locally on a user's machine or deployed on remote servers, facilitating distributed simulations that leverage the computational resources of multiple systems.
- **Dynamips**: This is the core emulator that allows GNS3 to run Cisco IOS images. It provides low-level emulation, simulating the hardware of Cisco routers and switches.
- **QEMU and VirtualBox**: These virtualization technologies are integrated into GNS3 to enable the emulation of non-Cisco devices. QEMU is particularly versatile,

allowing the emulation of a wide variety of hardware platforms, while VirtualBox is often used to run full operating systems as part of the network topology (36).

- **Docker Integration**: GNS3 also supports Docker, which enables lightweight and scalable deployment of network functions in a containerized environment (37). It can also collaborate with containers (38). This feature is particularly beneficial for simulating Network Function Virtualization (NFV) scenarios.

2.7.2 Mininet

Mininet is a network emulation platform that allows the creation and testing of realistic virtual networks on a single machine. It can provide a lightweight and flexible environment to emulate complex network topologies and conduct experiments in software-defined networking (SDN). Mininet is built on Linux networking tools, leveraging lightweight container-based virtualization to simulate the behavior of a complete network of hosts, switches, and links.

Mininet's primary strength lies in its ability to emulate large-scale networks with relatively minimal resource requirements. Each node in the network, including hosts, switches, and routers, is represented by a process within the Linux kernel, ensuring that the emulated network behaves similarly to a real-world network. Additionally, Mininet supports the OpenFlow protocol, making it much ideal for SDN research.

3

Design

3.1 High-level 6G NDTs architecture

At present, the industry has not only a variety of views and research on the definition of DTs, but also some concepts about DTs are relatively complicated, and they are mixed in different articles. Some focus on modeling, while some emphasize simulation and control. Readers are often easily confused about concepts between different articles. We summarize the background and the different emphases of the research on DTs, and we propose our definition of DTNs as follows:

A dynamic, virtual replica of a physical network's infrastructure, operations, and lifecycle. It mirrors, simulates, and interacts with the physical network in real-time, enabling continuous synchronization between the digital and physical environments. (16, 39). It uses data-driven computational models to maintain real-time conditions while forecasting future states (3). An NDT features bidirectional interfaces that not only update the digital twin based on changes in the physical network but also allow the digital twin to issue commands to alter the physical network.

Although there are some existing Network Digital Twin architectures, research in this area remains relatively sparse. Moreover, these architectures exhibit several issues, making it still needed to propose an NDTs system architecture, especially in the context of 6G network:

- They are often too broad and coarse, with insufficient detail and clarity in their explanations, resulting in a lack of depth.
- The explanations related to the network aspect are vague, making it seem like these architectures were not specifically designed for networks, but rather resemble

a generic digital twin architecture.

- Some are still focused on traditional networks and are not tailored for next-generation wireless networks. They often rely on conventional network approaches without incorporating NFV and SDN.
- There is a lack of connection between application and theory, making these approaches somewhat detached from practical realities, as they fail to demonstrate how theoretical concepts can be applied effectively.

In this section, we proposed a three-layer general 6G NDTs system structure, including the Physical Twin Layer, Digital Twin Layer, and Application and Service Layer, as shown in Figure 3.1.

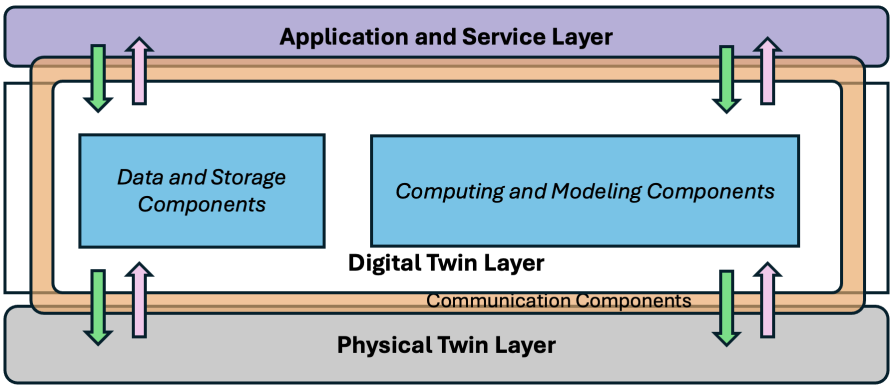

Figure 3.1: 6G network digital twins high-level architecture, including there layers: physical twin layer, digital twin layer, and application and service layer

3.1.1 Physical Twin Layer

The physical twin layer is primarily responsible for gathering and initially processing data from the physical environment. we deliberately chose not to adopt the term “Physical Layer” from certain sources(40) to avoid confusion, and the term "twin" means objectification and modularity.

In terms of composition requirements, the principle is that all objects in the physical world that can be modeled as digital models should be involved (40). Network-related hardware devices, such as User Equipment (UE), Next Generation NodeB (gNodeB), and other hardware within the Radio Access Network (RAN), will inevitably be included at this layer. However, to achieve a more comprehensive full-stack digital twins system,

environmental information must also be integrated, which is often ignored. For instance, location data is essential for positioning, along with other factors such as physical surfaces, as they affect signal attenuation.

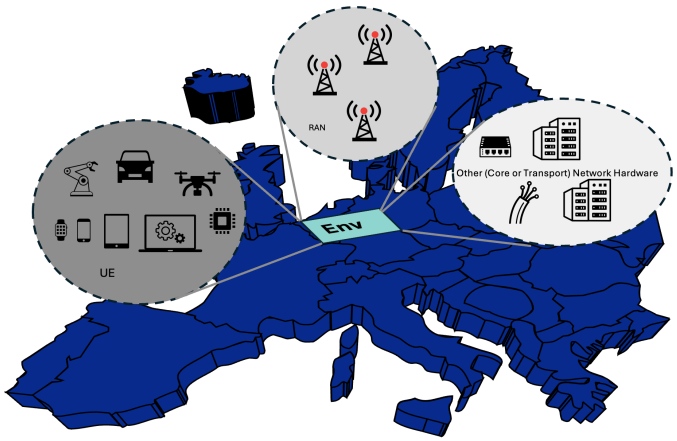

Figure 3.2: Physical twin layer, composed of network devices and key environment

In terms of performance requirements, the physical twin layer of a 6G NDTs system must have hardware capable of supporting high-frequency, efficient, and reliable data synchronization to ensure seamless updates and command execution between physical and digital components. We refer to the interfaces required for such communication as Southbound Interfaces (SBIs), which primarily handle physical-to-virtual and virtual-to-physical communication. The devices within the physical twin layer should also support physical-to-physical communication. Clearly, the design of 6G physical networks must meet fundamental requirements to support ultra-high data rates, low latency, and massive connectivity. Moreover, energy efficiency, security, and seamless interoperability with existing networks are also critical components.

3.1.2 Digital Twin Layer

The Digital Twin Layer includes digital counterparts of physical objects but is not limited to these representations. In functional components, we propose that this layer should minimally include three core components: Communication Components, Data Storage Components, and Computing and Modeling Components. Data storage and computing can both be distributed and centralized. Security should be considered in every function. To simplify, we will not abstract the security components separately for now.

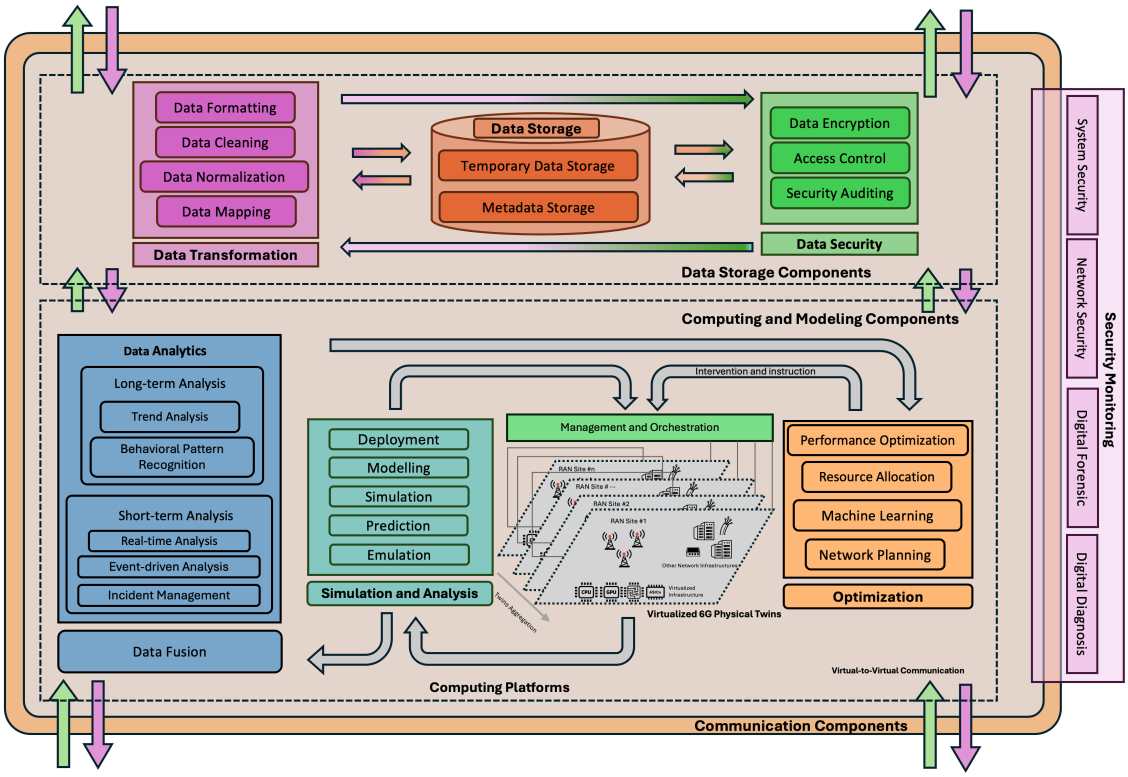

Figure 3.3: Digital twin layer, with communication components, computing and modeling components, and data storage components. (the composition is omitted in the figure)

3.1.2.1 Communication Components

They act as the essential link between physical and virtual objects and virtual-to-virtual communication. They manage the uninterrupted transmission and transformation of data, including instructions and influences to manipulate the physical network. This module converts raw data into structured formats that the Virtual Twins Layer can readily use, optimizing compatibility and usability across the system. It also integrates advanced data processing functions such as mapping to align real-world elements with their digital counterparts.

While there are some existing concepts of a four-layer architecture. In the four-layer architecture, an independent communication layer is introduced to manage this interaction. However, in our approach, we have integrated these communication functions directly into the digital twin layer. This difference in architectural design reflects divergent focal points. Abstracting the communication function as a separate layer underscores the role of "middleware" in facilitating interaction between the physical and digital domains. Conversely, by embedding this function within the digital twin layer, our approach aligns more closely

with the concept of an "interface," where communication is seamlessly integrated within the digital twin itself.

3.1.2.2 Data Storage Components

The Data Storage Components are for effective data management, encompassing temporary storage for quick data processing, metadata storage for efficient organization and retrieval, and robust data security measures. Data can come from various sources. For example, real-world data can be obtained from the physical world in real-time, while simulated data can also be cleaned, reorganized, stored, and packaged into datasets to support future analysis and use or training AI models. Together, these components should also be under a comprehensive system that safeguards data integrity while facilitating smooth and secure data operations. Storage can ideally be distributed, to fit the 6G architecture like D6G, where each network device can run a simplified 6G stack (41). While centralized storage is also an option, it may lead to data ownership and security issues, despite being more straightforward for data modeling and computational tasks.

3.1.2.3 Computing and Modeling Components

In terms of composition, this module is mainly composed of two parts:

- Virtualized Hardware: This part involves the virtualized hardware. The choice between simulation and emulation is based on specific requirements. If the network device is required to run a 6G stack internally, emulation should be used as much as possible.

- Containerized software: This part involves the implementation of service and network functions as software modules. With the introduction of SDN, software components enable dynamic and flexible network management by separating the control plane from the data plane. This allows centralized control of network traffic and resources, and can be programmed to adjust in real-time to meet specific requirements, such as load balancing or Quality of Service (QoS) optimization. The management component is not abstracted separately since it should also be included in the software part.

In terms of function, The Computing and Modeling Components should be capable of abstracting and aggregating computing and modeling units of various complexities and

granularity to meet specific requirements. This includes capabilities for modeling, simulation, and prediction, as well as conducting 'what-if' analyses. Additionally, it should incorporate a comprehensive fault detection system capable of swiftly identifying and diagnosing operational issues alongside robust system optimization tools that fine-tune performance across diverse network scenarios. Moreover, its predictive maintenance algorithms proactively suggest repairs and upgrades, significantly reducing downtime and extending the lifespan of network components. To be noted, these advanced functionalities often rely on deep analytics and machine learning algorithms, which substantially support the upper Application and Service Layer. These functions should ideally operate as stateless services, generating relevant storage and logs, which can be automatically saved to external storage. This data can then be combined with data generated by the network itself, such as telemetry data, to contribute to model training. AI models can replace traditional data modeling methods and be applied to network management, such as resource allocation, congestion control, and network slicing (42). These algorithms can continuously learn from ongoing operations and simulations, thereby enhancing their accuracy and efficacy.

3.1.3 Application and Service Layer

The Application and Service Layer is the uppermost tier of the 6G NDTs system architecture, where it directly orchestrates services and delivers enhanced experiences for end-users.

This layer incorporates various applications spanning augmented and virtual reality, autonomous vehicles, smart city technologies, and tele-medicine, all designed to seamlessly integrate with daily human activities and infrastructural operations. This layer can communicate with the digital twins layer, which we call interfaces between them Northbound Interfaces (NBI). Moreover, the Applications and Service Layer should be characterized by its Service-oriented Architecture (SOA) which enables it to offer modularized and reusable services. This modularity allows for services to be independently deployed, managed, and scaled, meeting specific user demands without affecting the overall system. Furthermore, it supports isolated operations where individual services can function in a standalone mode, enhancing fault tolerance and reducing dependencies. This isolation is essential for ensuring that any disruptions in one service do not cascade to others, thereby maintaining the robustness and continuity of user services. Besides, each service should be both resilient and precisely tailored to meet evolving user needs and environmental contexts.

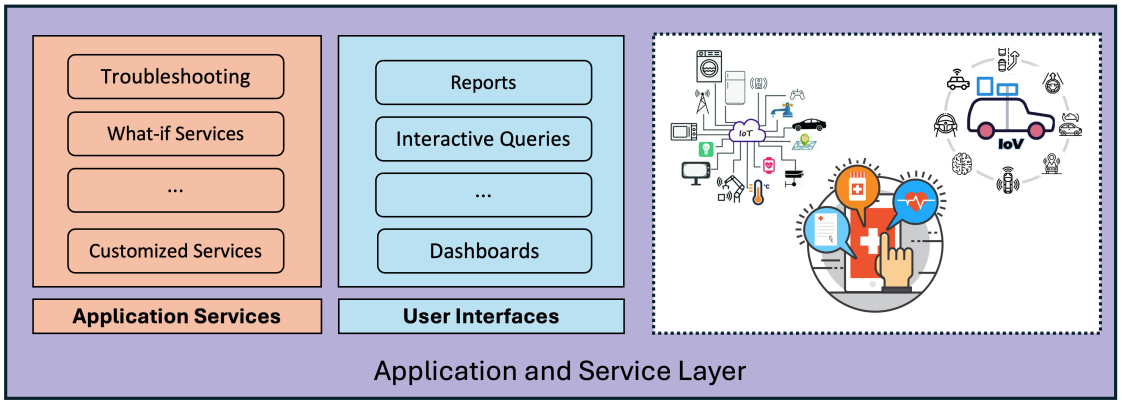

Figure 3.4: Application and service layer, focus on upper-tier interfaces and applications

3.2 Why Simulation or Emulation Tools?

After the architecture is proposed, a natural but broad question arises: **Where should we focus our efforts first when starting the deployment?** This involves a broad range of considerations, starting with the selection of tools. Before investigating this question, the priority is finding the research focus on network digital twining.

Considering the focus of our study, we will concentrate on the digital twin layer rather than the physical infrastructure or the application and service layers. The physical twin layer primarily consists of actual network infrastructure, and the application and service layer functions as interfaces exposed to higher layers. Therefore, specific requirements related to hardware infrastructure and service interfaces are not our main concern and priority. Our proposed digital twin layer consists of the digital replication of network infrastructure. This infrastructure should encompass the Radio Access Network (RAN), the core network (as termed in 5G), the transport network, and the physical environment. The most significant simulation challenges in this context lie within the RAN and core networks, while the transport network requires relatively fewer updates.

Given the central role of the digital twin layer in our design, the digital network forms the core of this layer. It is crucial to prioritize the capability to digitally replicate and operate the physical network. Currently, some basic prototypes and implementations of network digital twins require modeling the network communication components. While some approaches (43, 44, 45) use static statistical models, a more suitable method involves utilizing different network simulators (46, 47, 48). This requirement places specific demands on the selection and use of network simulators and emulators in developing NDTs.

3. DESIGN

The criteria for selecting the appropriate network simulators and emulators are often ambiguous, diverse, and inherently complex. It is essential to determine which types of simulators or emulators are best suited to our current architecture. While a network simulator or emulator can provide a strong foundation for creating NDTs, it must be integrated with real-time data to continuously update the model and validate its accuracy against the actual network. Additionally, these tools must offer sufficient scalability and balance between performance overhead and simulation/emulation fidelity.

Scalability and extensibility are challenging to quantify, particularly in the context of 6G, where a standardized architecture has yet to be established. Therefore, current tools need to be highly extensible to serve as a foundation for ongoing development. The complexity and novelty of 6G networks demand simulators and emulators capable of accurately replicating both physical and logical behaviors, while also being easily extended and updated as 6G standards evolve. In contrast, the performance overhead associated with common network components can be relatively measured to provide a general estimate. This suggests that an experimental approach can be adopted to evaluate these aspects.

To further explore whether different network simulators and emulators indeed exhibit differences , and if such differences exist, how significant they are, we conducted a series of experiments. The implementation and results of these experiments are discussed in the following sections.

4

Implementation

4.1 Network Simulation and Emulation Platforms

We selected four different network simulators and emulators to implement these topologies. Each platform was configured to ensure consistent functionality, allowing for a fair comparison of performance metrics.

We selected NS-3, OMNeT++, Mininet, and GNS3 for our comparative study due to their distinct characteristics and the specific demands of our research objectives. These platforms were chosen based on their widespread use in academic and industry settings, as well as their ability to handle a range of network configurations and scales.

4.2 Network Application Scenario

In our experimental setup across different simulators and emulators, we configured network devices to perform the most basic yet essential functions, ensuring that these devices were analogous in their capabilities across the various platforms. Each network device, configured with similar fundamental functionalities, was designed to operate in a comparable manner, allowing for a consistent testing environment. Specifically, we simulated a scenario where each pair of network devices engaged in mutual UDP packet exchange within the same network environment. For each PC node, we deployed a UDP application with standardized parameters, as previously discussed. The common configuration for each node is shown in Figure 4.1. These default parameters included a transmission rate of 1 Mbps, a packet size of 1024 bytes, and a communication duration of 10 seconds, with reporting intervals set from 0.1 to 1 second. By maintaining uniformity in the configuration of these basic functionalities across all network devices, we ensured that the network behaviors

observed were attributable to the underlying simulator or emulator rather than disparities in device setup. This approach allowed us to focus on analyzing the performance and behavior of the network under consistent conditions, facilitating a more accurate comparison of the different simulation and emulation platforms.

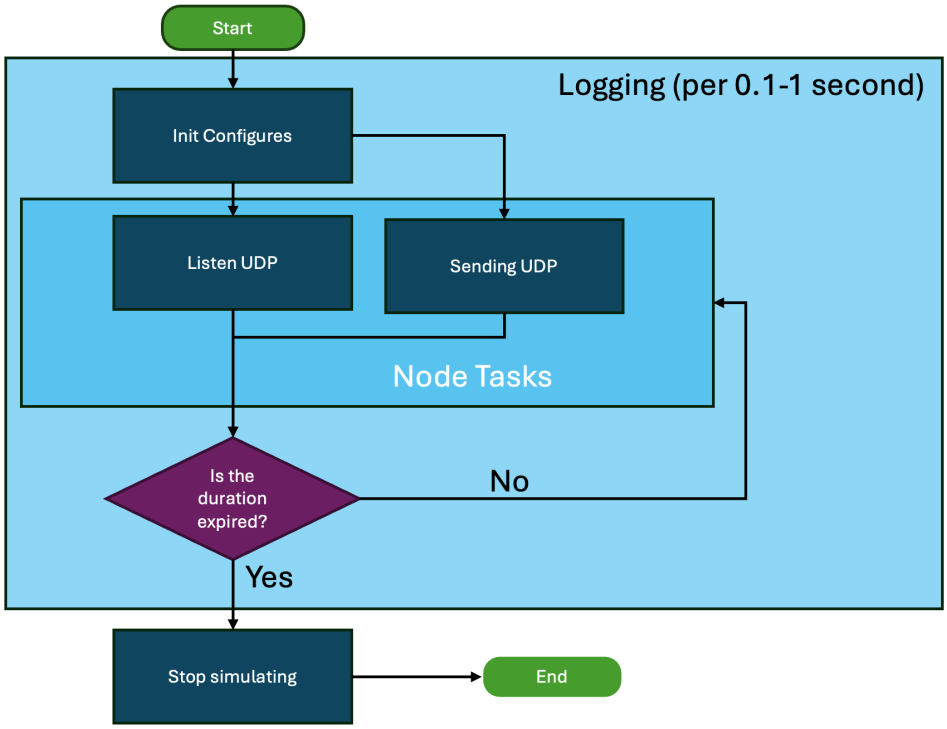

Figure 4.1: UDP application experimental settings flow chart

4.3 Network Topology

In this study, we designed and implemented four distinct network topologies: Star topology, Mesh topology, Tree topology, and Ring topology. Each topology is structured with switches interconnected in different configurations, with each switch connected to a PC node. These topologies, while not always representative of contemporary real-world networks, were chosen to facilitate a diverse set of performance evaluations. Each topology has unique characteristics, such as the potential for network loops in mesh topology, and we anticipate that simulators and emulators will behave differently, particularly in discrete event simulations where more complex topologies generate a higher number of events and greater complexity. This diversity enables a more comprehensive analysis of memory usage and CPU time across various network simulator and emulator platforms.

- **Star Topology**: The star topology features a central switch that connects to all other nodes. This configuration is straightforward and easy to manage, making it efficient for small-scale networks. However, it is highly susceptible to the failure of the central switch, which can lead to complete network downtime.

- **Mesh Topology**: In the mesh topology, each switch is connected to every other switch, providing high redundancy and reliability. This topology is resilient to individual link failures, ensuring robust network performance. However, it is resource-intensive, requiring a large number of connections, which can lead to high memory and CPU usage, especially as the network scales.

- **Tree Topology**: The tree topology organizes switches in a hierarchical manner, similar to a binary tree structure. This topology is efficient for managing larger networks and is scalable. However, it may suffer from bottlenecks as the root switch or upper-level switches can become overloaded. This is a relatively more widely-used topology.

- **Ring Topology**: The ring topology connects switches in a circular manner, where each switch is connected to two others. This configuration provides a simple and cost-effective network setup. However, it is prone to performance issues if a single link fails, as it may disrupt the entire network.

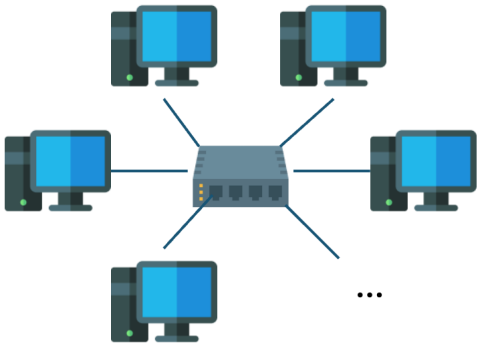

Figure 4.2: Star Topology, where multiple nodes are connected to the central switch

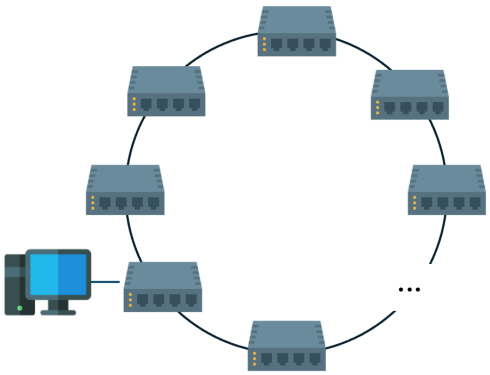

Figure 4.3: Ring Topology, where each switch is connected to form a ring

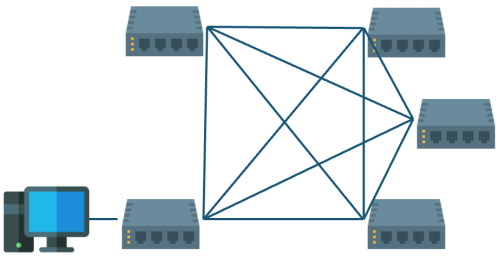

Figure 4.4: Mesh Topology, where each switch is interconnected with every other switch

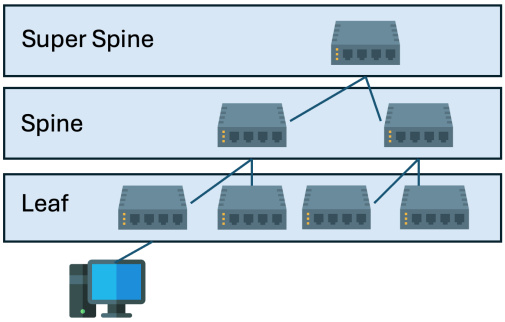

Figure 4.5: Tree Topology, where switches are organized in hierarchical layers (usually represented as a binary tree)

4.3.1 Scalable Configuration

When configuring different network topologies in our experiments, we encountered limitations imposed by certain emulators, for example, GNS3, where the most basic switch models are constrained to only 8 ports. This limitation poses challenges, particularly when implementing star and mesh topologies with more nodes, as the available switch interfaces may become insufficient. To address this issue without modifying the configuration files, which could potentially introduce unintended variables into our benchmarking tasks, we adopted a straightforward solution. In short, when a switch has only one port remaining, we connect this port to another switch. This approach effectively extends the network by providing additional interfaces (e.g., six or seven more ports), thereby enabling the continuation of our intended topology configuration without altering the original setup or compromising the integrity of our benchmarks. As long as the topology is the same, each simulator or emulator uses the same algorithm, thus ensuring consistency.

4.4 Performance Metrics

For evaluating the performance of the different network topologies, we focused on two primary metrics: memory usage and CPU time. These metrics are important for understanding the resource demands of each network configuration, particularly when scaled from 2 to 40 nodes. Memory usage was measured by monitoring the Virtual Memory Resident Set Size (VMRSS), while CPU time was tracked to evaluate the processing efficiency of each simulator/emulator. The details can be found in the evolution section, Section 5.2

Table 4.1: Basic comparison of NS-3 and GNS3

Feature	**NS-3**	**GNS3**
Type	Simulator	Emulator
Primary Use	Network protocol simulation	Network device emulation
GUI Support	Yes	Yes
Level of Abstraction	Packet-level	Device-level
Real-time Execution	No	Yes
Language Support	C++, Python	Python, Various
Virtualization	No	Yes
Customizability	High (custom models)	High (supports various images)
Scalability	High	Limited by physical resources
Hardware Integration	Yes	Yes (with physical devices)
Network Stack Simu	Yes	No
Discrete Event Simu	Yes	No
Example Supported	TCP, UDP, IPv4/IPv6, Wi-Fi	Any real network protocol
Docker Support	Yes	Yes
Documentation	Extensive	Extensive
Learning Curve	Steep	Moderate

NS-3 and OMNeT++ were chosen as they are both well-established discrete-event network simulators, yet they offer different approaches to simulating network protocols. NS-3, written in C++, is widely recognized for its comprehensive support of IP-based networks, making it a robust choice for detailed protocol analysis. In contrast, OMNeT++ is more modular and flexible, often used for both networking and non-networking simulations, which allows for a broader range of experimental scenarios. This distinction is crucial as it enables us to analyze how different underlying architectures of simulators affect performance under varying network conditions.

On the other hand, Mininet and GNS3 represent network emulation tools, but they diverge significantly in their operational approaches. Mininet, developed in Python, utilizes network namespaces and virtual Ethernet interfaces to create lightweight, scalable network topologies that closely resemble real-world environments. This makes Mininet particularly suitable for scenarios where rapid prototyping and testing of network topologies are required with minimal resource overhead. Conversely, GNS3, also Python-based, relies on virtualization technologies like QEMU or VirtualBox to run actual network operating systems, offering a higher fidelity emulation at the cost of increased computational resources.

Table 4.2: Basic comparison of OMNet++ and Mininet

Feature	OMNet++	Mininet
Type	Simulator	Emulator
Primary Use	Discrete event simulation	Network topology emulation
GUI Support	Yes	No
Level of Abstraction	Event-level	Network-level
Real-time Execution	No	Yes
Language Support	C++, Python	Python
Virtualization	No	Yes
Customizability	High	Medium (more on virtual networks)
Scalability	High	Limited by physical resources
Hardware Integration	No	Yes (with virtual machines)
Network Stack Simu	Yes	Partial
Discrete Event Simu	Yes	No
Example Supported	TCP, UDP, MANET, LTE	TCP, UDP, SDN
Docker Support	Limited	Yes
Documentation	Extensive	Extensive
Learning Curve	Steep	Moderate

Their specific characteristics and comparison can be seen in Table 4.1 and 4.2.

4.5 Performance Logging Principle

Our method for recording performance overhead is targeted. When monitoring system performance, it is important to note that resource consumption is not always limited to a single process. In many cases, multiple processes are involved, particularly when a main process spawns several child processes. These child processes, while being initiated by the main process, operate independently and consume their own share of CPU, memory, and other resources. Therefore, to accurately assess the overall performance impact of an application or service, we need to monitor not just the main process but also all its associated child processes.

4.5.1 Main Process Logging

We focus exclusively on tracking the performance metrics of the specific simulation or emulation process, identified by its PID, with data collected directly from the `/proc/[pid]/status`

file. Each benchmark task is executed within a container, ensuring an isolated environment, while the logging process operates as a separate, independent process.

The logging begins automatically as soon as the relevant simulation or emulation process is detected, with a recording frequency that varies between 0.1 and 1 second. This frequency is dynamically adjusted based on the scale of the simulation or emulation to ensure that sufficient data is collected for meaningful analysis. Additionally, to minimize any potential interference, all GUI-related components are disabled or excluded, allowing the simulation or emulation to run with only the essential resources required. This approach ensures that our performance data accurately reflects the true overhead of the process under study, excluding extraneous factors to some extent.

4.5.2 Child Processes Logging

Table 4.3: GNS3 Server process and related Dynamips sub-processes

Process Type	Process ID (PID)	Parent Process ID (PPID)	Port
GNS3 Server	1617	378	N/A
Dynamips Instance 1	903103	1617	40431
Dynamips Instance 2	903206	1617	60023
Dynamips Instance 3	903303	1617	48837
Dynamips Instance 4	903393	1617	50871
Dynamips Instance 5	903532	1617	58065
Dynamips Instance 6	903685	1617	45603
Dynamips Instance 7	903759	1617	33011
Dynamips Instance 8	903924	1617	40349
Dynamips Instance 9	904126	1617	37089
...	...	...	...
Dynamips Instance n	904292	1617	56331

When working with network emulators, it is important to note that, in addition to the main server process, there are often many other processes that depend on or are directly spawned by this process. For example, in GNS3, when network device instances are created, several additional processes are often started. In our experiments, especially when the network scale is large, many child processes are initiated, such as Dynamips, Nodes, and UBridge.

- **Dynamips:** is an emulator used primarily to simulate Cisco router hardware, allowing for the emulation of actual Cisco IOS images.

- **Node:** normally, the VPCS (Virtual PC Simulator) is used; it is a lightweight tool that simulates multiple virtual PCs within the GNS3 environment, enabling basic network tests like ping and traceroute across the simulated network.
- **UBridge:** acts as a network bridge, connecting GNS3 to various emulation and simulation tools, facilitating packet forwarding between different network devices and protocols.

Table 4.4: GNS3 Server process and related VPCS sub-processes

Process Type	Process ID (PID)	Parent Process ID (PPID)	Port
GNS3 Server	1617	378	N/A
Node Instance 1	945581	1617	5002
Node Instance 2	945684	1617	5005
Node Instance 3	945775	1617	5008
Node Instance 4	945872	1617	5011
Node Instance 5	946007	1617	5014
Node Instance 6	946158	1617	5017
Node Instance 7	946268	1617	5020
Node Instance 8	946393	1617	5023
Node Instance 9	946580	1617	5026
...	...	...	...
Node Instance n	946776	1617	5029

In Mininet, various processes are associated with network simulation, including processes for each virtual host and switch. Additionally, the `ovs-vswitchd` process plays a significant role as it is responsible for managing the Open vSwitch data plane, handling tasks such as packet forwarding. Below is a table summarizing the key Mininet-related processes identified in our experiments.

In our performance logging, we included all related processes, such as the `ovs-vswitchd` and the individual Mininet hosts and switches, to ensure a comprehensive assessment of resource utilization.

Although the various related processes of different emulators are more complicated and organized in different ways. In our testing, we ensure that all related child processes of each simulator or emulator are included in the performance analysis.

Table 4.5: GNS3 Server process and related UBridge sub-processes

Process Type	Process ID (PID)	Parent Process ID (PPID)	Port
GNS3 Server	1617	378	N/A
UBridge Instance 1	945589	1617	43207
UBridge Instance 2	945692	1617	35983
UBridge Instance 3	945783	1617	33485
UBridge Instance 4	945880	1617	40011
UBridge Instance 5	946015	1617	35917
UBridge Instance 6	946166	1617	44859
UBridge Instance 7	946276	1617	33897
UBridge Instance 8	946401	1617	42111
UBridge Instance 9	946588	1617	38735
...	...	...	...
UBridge Instance 10	946784	1617	46555

Table 4.6: Mininet processes and related processes, including Mininet Host, and Mininet Switch

Process Type	Process ID (PID)	Parent Process ID (PPID)
ovs-vswitchd	65	64
Controller	545424	-
Mininet Host (h1)	539862	-
Mininet Host (h2)	539864	-
Mininet Host (h3)	539871	-
Mininet Switch (s1)	539992	-
Mininet Switch (s2)	540000	-
Mininet Switch (s3)	540003	-
...	...	...

5

Evaluation

5.1 Experiment Setup

5.1.1 Environments

Unlike model training or scientific computation, where the algorithms are often the primary focus, benchmarking is relatively more sensitive to the underlying infrastructure and even small variations can significantly impact the results.

For the experiment environment, there are several considerations:

- **Environment Consistency:** VMs provide a consistent and controlled environment, which is essential for accurately benchmarking the performance in our task. In contrast, HPC platforms are typically optimized for parallel computing tasks and may introduce variability in resource allocation and performance due to shared infrastructure and job scheduling systems. This variability could lead to inconsistent results, making it difficult to attribute performance differences solely to the simulators or emulators being tested.

- **Controllable Resource Isolation:** VMs offer better isolation of resources such as CPU and memory, ensuring that each simulation or emulation instance runs in a dedicated environment without interference from other processes. On an HPC platform, resources are often shared among multiple users and jobs, which can introduce noise and affect the reliability of performance measurements, particularly when testing at different scales.

- **Ease of Deployment with Containers:** By placing our benchmark code within containers, we further enhance consistency and reproducibility across different VM

environments. Containers can be easily deployed and managed within VMs. This level of control is harder to achieve on HPC platforms, where the underlying system and resource management tools might differ from one node to another.

5.1.1.1 Hardware

We have multiple options for hardware selection, such as local machines, DAS-6 distributed supercomputer clusters, and virtual machines provided by Surf.

When the network size is too large, the local machine will crash, and there will also be a port shortage. Therefore, we ruled out this option at the beginning. For DAS-6, its performance is the best and most suitable for high-load operations. However, it is a multi-user and large-scale system, and its distributed computing paradigms, job scheduling, and resource management are also relatively complex and hard to control. DAS-6 is a shared academic supercomputing platform on which users are generally not granted superuser (root) privileges. This is a common security measure in multi-user environments to prevent unauthorized access to critical system resources and maintain the platform's overall integrity and stability. However, the lack of this privilege often affects operations such as virtualization. It also does not support Docker, where Docker daemon specifically requires root access. Podman can replace Docker to a certain extent, but the inability to use Docker is just a microcosm. Without the sudo command, installing dependencies will still be difficult and obstructions.

For the above reasons, we chose the VMs of the System and Network Engineering Lab. Two VM instances with exactly the same configuration are used. The hardware details are listed below, including CPU information in Table 5.1, cache information in Table 5.2, and NUMA configuration in Table 5.3.

5.1.1.2 Software

In addition to the hardware configuration, our experimental setup is divided into two main software components: the system environment and the container environment. The system environment includes the underlying operating system and its associated tools, which provide the necessary foundation for running and managing the experiments. The second component is the container environment, which we use to containerize our benchmarking tools and applications. By using containers, we can isolate the software dependencies and configurations for each experiment, ensuring that they run in a controlled and repeatable manner.

Table 5.1: Hardware: VMs CPU Information

Property	Details
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Address sizes	40 bits physical, 48 bits virtual
Byte Order	Little Endian
CPU(s)	8
On-line CPU(s) list	0-7
Vendor ID	GenuineIntel
Model name	Intel Xeon (Cascadelake)
CPU family	6
Model	85
Thread(s) per core	1
Core(s) per socket	1
Socket(s)	8
Stepping	6
BogoMIPS	4190.15
Virtualization	VT-x
Hypervisor vendor	KVM
Virtualization type	Full

Table 5.2: Hardware: VMs Cache Information

Cache	Size
L1d	256 KiB (8 instances)
L1i	256 KiB (8 instances)
L2	32 MiB (8 instances)
L3	128 MiB (8 instances)

Table 5.3: Hardware: VMs NUMA Configuration

Property	Details
NUMA node(s)	1
NUMA node0 CPU(s)	0-7

In order to conduct our experiments in a controlled and consistent environment, we have chosen to use a standardized setup for our operation system configuration. This ensures that the results we obtain are not influenced by variations in hardware or software across different test runs. Table 5.4 provides a detailed overview of the system specifications and virtualization environment of the system used throughout our study.

Docker is a popular containerization tool that allows users to run applications in iso-

Table 5.4: Software: VMs System Information

Property	Details
Distributor ID	Debian
Description	Debian GNU/Linux 12 (bookworm)
Release	12
Codename	bookworm
Kernel Version	6.1.0-23-amd64
Virtualization	kvm
Operating System	Debian GNU/Linux 12 (bookworm)
Architecture	x86-64
Hardware Vendor	OpenStack Foundation
Hardware Model	OpenStack Nova
Firmware Version	1.14.0-2

lated environments. However, Docker typically requires root access to manage containers. Podman is an alternative to Docker, which is designed to work in environments without root access. Podman can run containers in a rootless mode, meaning that users can manage containers as non-privileged users without compromising system security. This makes Podman a suitable option for containerized workflows on platforms like DAS-6. The basic system information for images is listed in Table 5.5, for

We conducted some preliminary experiments on the DAS-6 platform using Podman, while all of our final experiments were completed in virtual machines (VMs) using Docker. Importantly, we utilized the same images across both environments to ensure consistency and comparability of the results.

To establish a consistent foundation for our experiments on four different simulators and emulators, we utilized a base container image configured with specific system properties, as shown in Table 5.5. Based on this base image, we configure the environment for each simulator and emulator and build container images.

To be noted, although OMNeT++ and GNS3 both offer graphical user interfaces (GUIs) for visualization and ease of use, we chose to not install or disable these interfaces in our experiments. This decision was made to focus solely on measuring the performance of the core simulation and emulation engines without the additional overhead or influence of the GUI components.

Table 5.5: Software: containers base image system and resource information

Property	Details
Distributor ID	Ubuntu
Description	Ubuntu 20.04.6 LTS
Release	20.04
Codename	focal
Kernel Version	6.1.0-23-amd64
Architecture	x86-64
CPU(s)	8
Thread(s) per core	1
Core(s) per socket	1
Socket(s)	8
NUMA node(s)	1
Model name	Intel Xeon Processor (Cascadelake)
Virtualization	VT-x
Hypervisor vendor	KVM
Virtualization type	Full
L1d cache	256 KiB
L1i cache	256 KiB
L2 cache	32 MiB
L3 cache	128 MiB

Table 5.6: Software: containers software and tools version information

Software/Tool	Version
NS-3	3.30
GNS3 Server	2.2.49
OMNeT++	6.0
INET Framework	4.4
QEMU Emulator	4.2.1
Mininet	2.3.0
Python	3.8.10
GCC	9.4.0

5.2 Evaluation Matrics

5.2.1 VMRSS

`VMRSS` (Virtual Memory Resident Set Size) represents the amount of physical memory used by a process in a Linux system. Specifically, it indicates the size of the page frames of the process that reside in physical memory. This part of the memory includes the actual physical memory occupied by the process, excluding portions swapped out to disk.

`VMRSS` is calculated by multiplying the number of resident pages (usually in units of 4KB) by the page size. The formula is as follows:

$$\texttt{VMRSS} = \texttt{Number of Resident Pages} \times \texttt{Page Size} \tag{5.1}$$

where:

- **Number of Resident Pages**: The number of pages that are resident in physical memory.
- **Page Size**: The size of a memory page, typically 4KB.

`VMRSS` can be viewed through the `/proc/[pid]/status` file under the `VMRSS` field, displayed in KB.

5.2.2 CPU Time

`CPU Time` refers to the actual time a process spends running on the CPU, which can be divided into user mode time (User CPU Time) and kernel mode time (System CPU Time).

- **User CPU Time**: The time the process spends executing code in user mode.
- **System CPU Time**: The time the process spends in kernel mode executing system calls and other kernel operations.

`CPU Time` is calculated as the product of the number of system clock ticks and the clock tick time. Specifically, it can be calculated using the following formula:

$$\texttt{CPU Time} = (\texttt{User Mode Clock Ticks} + \texttt{Kernel Mode Clock Ticks}) \times \texttt{Clock Tick Time} \tag{5.2}$$

where:

- **User Mode Clock Ticks**: The number of clock ticks spent executing in user mode.
- **Kernel Mode Clock Ticks**: The number of clock ticks spent executing in kernel mode.
- **Clock Tick Time**: The duration of each clock tick, typically $\frac{1}{\texttt{System Clock Frequency}}$.

5.2.3 Retrieving CPU Time

In Linux, `utime` and `stime` fields in the `/proc/[pid]/stat` file provide the required data:

- `utime`: The amount of CPU time spent in user mode, measured in clock ticks.
- `stime`: The amount of CPU time spent in kernel mode, measured in clock ticks.

To convert these clock ticks to seconds, we can use the system clock frequency, which can be obtained using the `sysconf(_SC_CLK_TCK)` command . The conversion formula is:

$$\texttt{User CPU Time (seconds)} = \frac{\texttt{utime}}{\texttt{Clock Ticks Per Second}} \tag{5.3}$$

$$\texttt{System CPU Time (seconds)} = \frac{\texttt{stime}}{\texttt{Clock Ticks Per Second}} \tag{5.4}$$

5.3 Results

5.3.1 Data Preprocess

For the main research question, we configured the pipeline for each simulator/emulator and ran each configuration approximately five times, storing the results in externally mounted storage. This process yielded raw data, but the data was somewhat disorganized and required further preprocessing, such as data cleaning. In the following sections, we describe some of the methods we employed. We have omitted explanations related to data reformatting and instead focused on other key aspects of our preprocessing approach.

5.3.1.1 Multiple Trials

In our experiments, we employed a rigorous data logging methodology to ensure the reliability and accuracy of the performance measurements. Each test scenario was executed multiple times, with a minimum of five trials for each configuration. This approach was adopted to account for variability in the system's performance due to factors such as background processes, network conditions, and other environmental variables.

During the data collection process, we closely monitored the results for any instances of extreme performance fluctuations or results that were clearly unreasonable and significantly different from the rest. Such anomalies could be indicative of transient issues, such as unexpected system load or network disturbances, that do not reflect the typical performance of the system under study. When such unusual runs were detected, they were directly excluded from the final analysis to maintain the integrity of our results.

5.3.1.2 Outlier Removal

Outliers, which are data points that deviate significantly from the rest of the dataset, can often distort the results and lead to misleading conclusions. To enhance the accuracy of our analysis, we implemented two outlier detection methods and removal procedures on the collected data. One is basic method, another one is machine learning based method.

The first method we used is the Interquartile Range (IQR) method to detect outliers. The `IQR` is calculated as the difference between the third quartile (Q3) and the first quartile (Q1) of the dataset:

$$IQR = Q3 - Q1 \tag{5.5}$$

Using the `IQR`, we define a threshold to determine outliers. Any data point that lies below $Q1 - 1.5 \times IQR$ or above $Q3 + 1.5 \times IQR$ is considered an outlier and was subsequently removed from our dataset:

$$LowerBound = Q1 - 1.5 \times IQR \tag{5.6}$$

$$UpperBound = Q3 + 1.5 \times IQR \tag{5.7}$$

Another advanced method we employed for outlier detection is the Isolation Forest algorithm. It has been successfully implemented in GNS3 and mininet, since the traditional method does not come up with better results in them. Isolation Forest is based on the principle that outliers are fewer and different from the rest of the data, making them easier to isolate.

The algorithm works by recursively partitioning the data using randomly selected features and split values, creating an ensemble of isolation trees. Outliers are identified based on the path length required to isolate them in these trees. Points that are isolated closer to the root of the tree are more likely to be outliers.

The anomaly score $s(x, n)$ for a data point x is calculated as:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}} \tag{5.8}$$

where:

- $E(h(x))$ is the average path length from the root node to the terminating node for point x in an isolation tree.
- $c(n)$ is the average path length of unsuccessful searches in a binary tree, defined as:

$$c(n) = 2H(n-1) - \frac{2(n-1)}{n} \tag{5.9}$$

  where $H(i)$ is the harmonic number, which can be approximated by $\ln(i)+0.5772156649$ (Euler-Mascheroni constant).

- n is the number of instances in the dataset.

Data points with an anomaly score close to 1 are considered outliers and are removed from the dataset.

5.3.2 Comparison of Network Size and Type

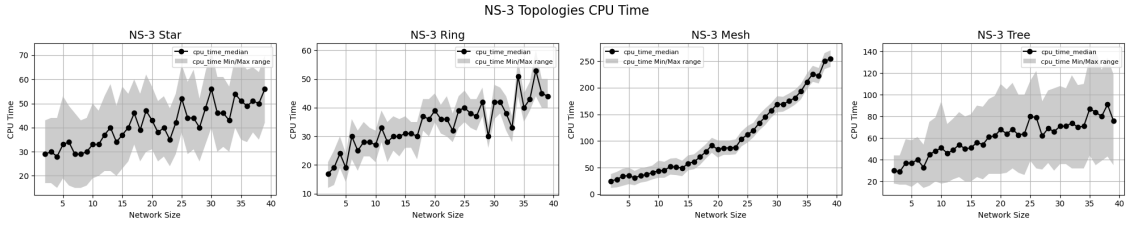

Figure 5.1: NS-3's CPU usage variation with network scale across different topologies: star, sing, mesh, and tree

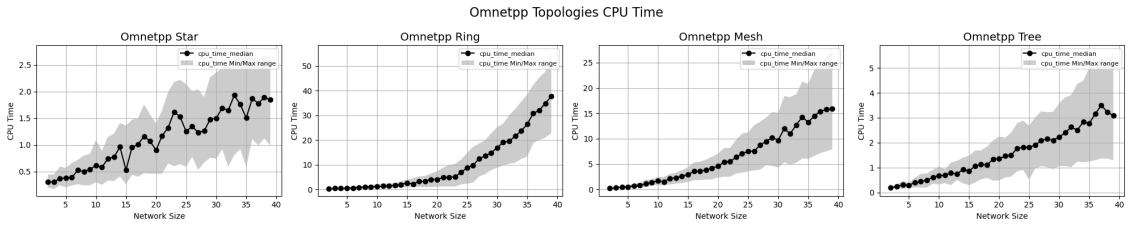

Figure 5.2: Omnetpp's CPU usage variation with network scale across different topologies: star, sing, mesh, and tree

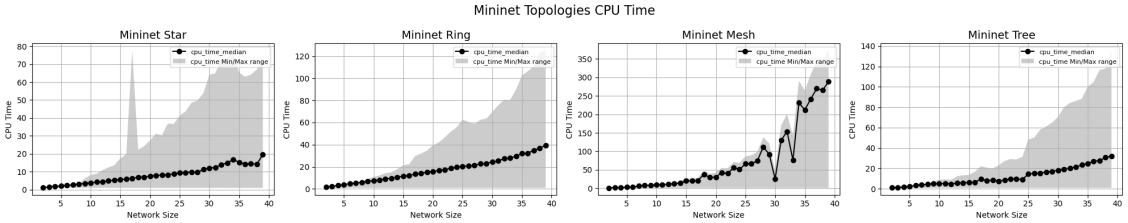

Figure 5.3: Mininet's CPU usage variation with network scale across different topologies: star, sing, mesh, and tree

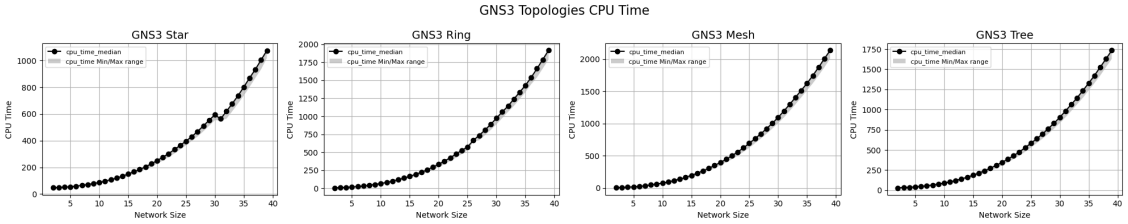

Figure 5.4: GNS3's CPU usage variation with network scale across different topologies: star, sing, mesh, and tree

The Figures 5.1, 5.2, 5.3 and 5.4 present the CPU time utilization measured about four different simulators/emulators: Mininet, NS-3, OMNet++, and GNS3, each evaluated under four distinct network topologies: Star, Ring, Mesh, and Tree. The direct comparison can be shown in Figures 5.9 , 5.11, 5.13 and 5.15. In the Mininet simulator, the CPU time increases consistently across all topologies as the network size grows, with the Mesh topology demonstrating the steepest increase, indicating higher computational demands in more interconnected network structures. In contrast, the NS-3 simulator shows a more varied pattern where the Mesh topology again requires significantly more CPU time, followed by the Ring and Tree topologies, while the Star topology exhibits the lowest CPU time. GNS3, on the other hand, displays a similar trend to NS-3, with the Mesh topology consuming the most CPU time, particularly as the network size expands, whereas the Star topology consistently requires the least CPU resources. The situation of OMNet++ is very similar to that of NS-3. It also conforms to the rule that the more complex and larger the network is, the more CPU resources are occupied. Moreover, its principle is similar to that of NS-3, which seems to explain their similarities to a certain extent. These observations suggest a clear correlation between network complexity and CPU utilization across different simulation environments.

The Figures 5.5, 5.6, 5.7, and 5.8 illustrate the VMRSS (Virtual Memory Resident Set Size) usage across the same set of simulators/emulators and network topologies. More

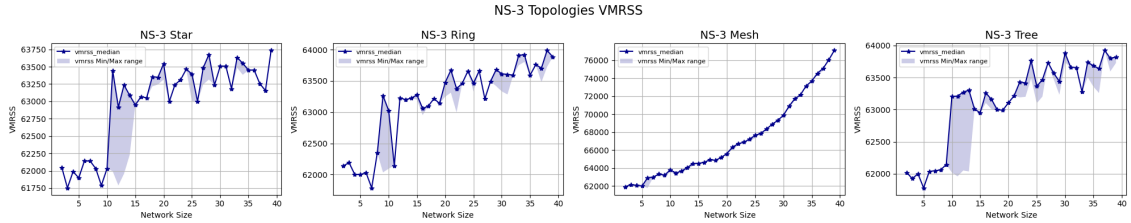

Figure 5.5: NS-3's memory usage variation with network scale across different topologies: star, sing, mesh, and tree

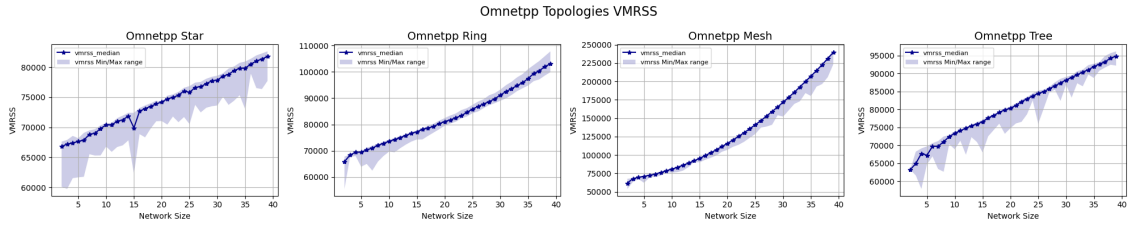

Figure 5.6: Omnetpp's memory usage variation with network scale across different topologies: star, sing, mesh, and tree

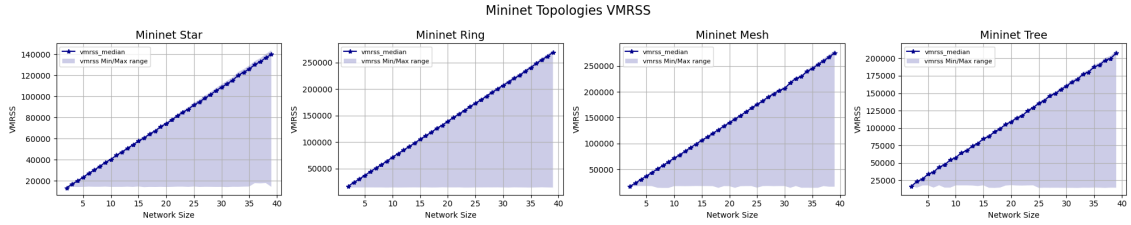

Figure 5.7: Mininet's memory usage variation with network scale across different topologies: star, sing, mesh, and tree

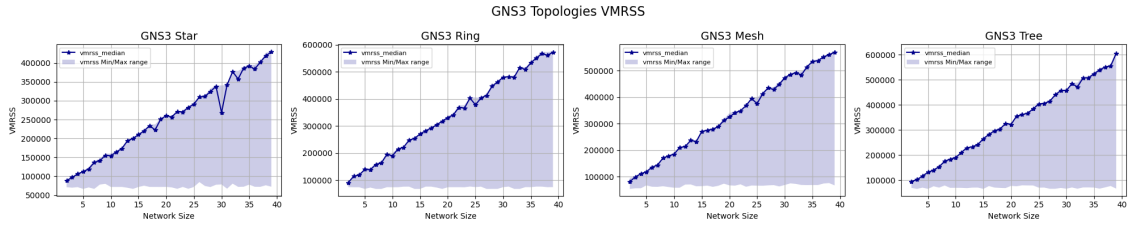

Figure 5.8: GNS3's memory usage variation with network scale across different topologies: star, sing, mesh, and tree

direct comparisons can be shown in Figures 5.10 , 5.12, 5.14 and 5.16. In Mininet, the VMRSS usage follows an upward trajectory with the increase in the number of nodes, with the Mesh topology consistently using the most memory. The Ring and Tree topologies demonstrate moderate memory usage, while the Star topology remains the most memory-

efficient. In the NS-3 simulator, the Mesh topology again shows the highest VMRSS consumption, especially as the network grows, followed by the Tree and Ring topologies, with the Star topology using the least memory. GNS3 reflects a similar pattern where the Mesh topology demands the most memory resources, and the Star topology exhibits the lowest VMRSS usage. One notable observation in GNS3 is the significant difference between the lower bound and the median of VMRSS utilization. This discrepancy arises primarily during the initialization phase of the emulation. At the start of the emulation, various virtual network devices and channels are gradually brought online. The process of creating and initializing these components must be fully completed before the actual emulation can begin. This initialization phase is both critical and time-consuming. The initially low VMRSS utilization reflects the incremental resource allocation, which increases substantially once all devices and channels are operational and will be stable in the later stage, For OMNet++, as the network size increases, the increment of VMRSS occupancy becomes higher and more stable. These results indicate that more complex topologies like Mesh tend to impose greater memory requirements across all simulators/emulators, with simpler topologies like Star being more memory-efficient.

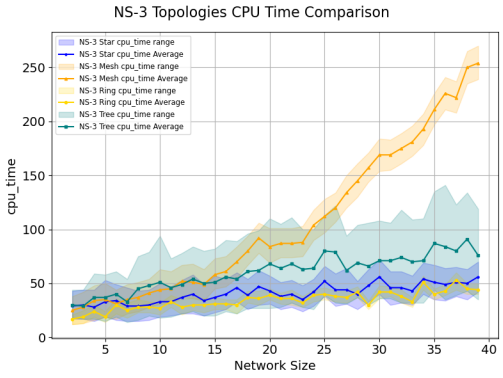

Figure 5.9: Comparison of NS-3's CPU usage variation with network scale across different topologies

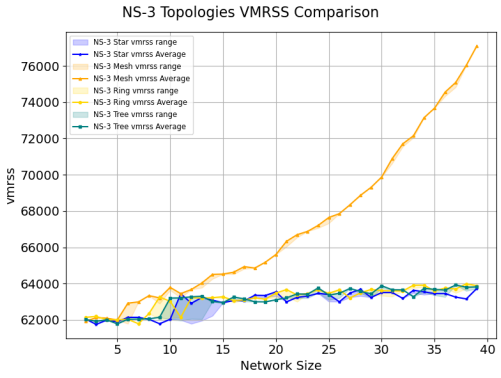

Figure 5.10: Comparison of NS-3's memory usage variation with network scale across different topologies

5.3.3 Comparison Across Network Simulators and Emulators

For network simulators like NS-3 and OMNet++. They are designed to model and analyze network behavior in a highly controlled and abstracted environment. These tools simulate the network protocols, events, and interactions between nodes without requiring real-time or real-world interaction. This means that they should normally have lower performance overhead intuitively, and our experiments also proved this.

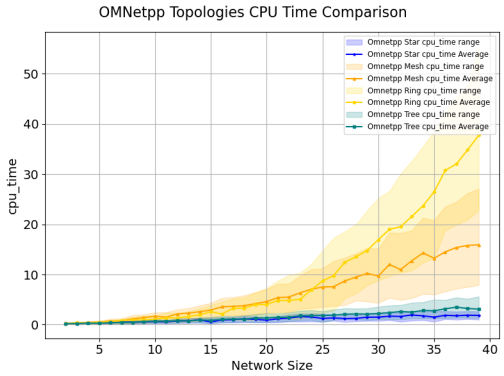

Figure 5.11: Comparison of OMNetpp's CPU usage variation with network scale across different topologies

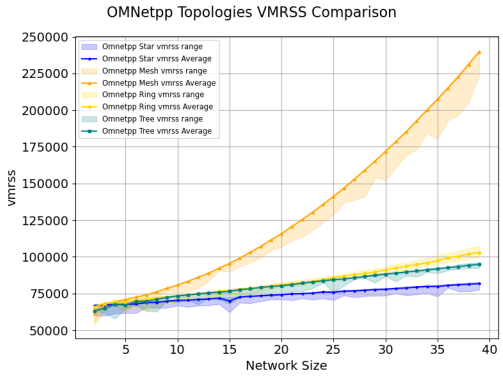

Figure 5.12: Comparison of OMNetpp's memory usage variation with network scale across different topologies

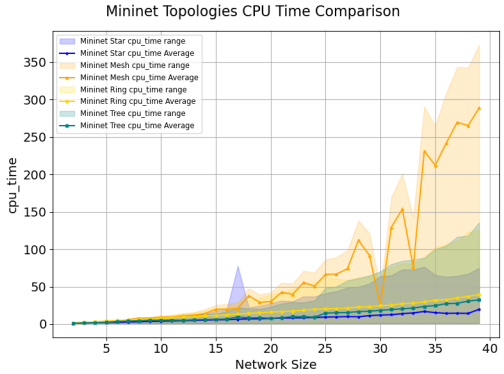

Figure 5.13: Comparison of Mininet's CPU usage variation with network scale across different topologies

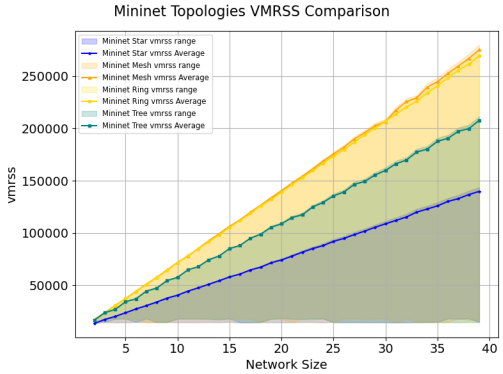

Figure 5.14: Comparison of Mininet's memory usage variation with network scale across different topologies

For NS-3, it manages a large queue of events, such as packet transmissions, receptions, and processing, which need to be scheduled and executed at specific times. The CPU time in NS-3 is largely consumed by the event-driven architecture that requires careful scheduling and execution of potentially millions of events, especially in large or highly interconnected topologies like Mesh. Additionally, NS-3 maintains detailed data structures such as routing tables, packet buffers, and protocol state machines, which consume relatively more memory. The modularity and extensibility of NS-3, while powerful, add layers of abstraction that increase the overhead in both time and memory usage as more complex topologies or more detailed protocol models are employed.

OMNeT++ is another discrete-event simulator with a focus on component-based modular simulation. It is particularly adept at simulating large-scale network systems and

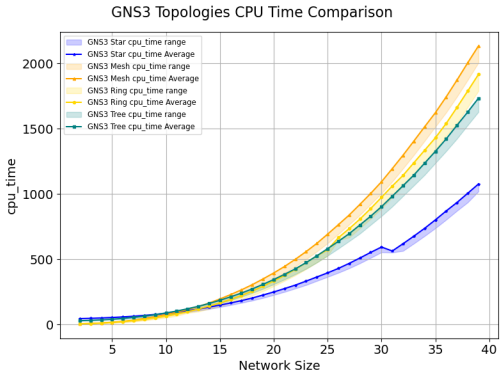

Figure 5.15: Comparison of GNS3's CPU usage variation with network scale across different topologies

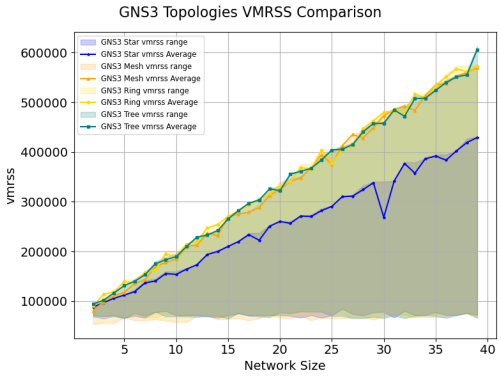

Figure 5.16: Comparison of GNS3's memory usage variation with network scale across different topologies

supports hierarchical topologies. OMNeT++ tends to be more memory-intensive compared to NS-3 due to its graphical and hierarchical modeling capabilities. The framework allows for complex interconnections and large-scale simulations, which require extensive memory to maintain the state of each module and more CPU resources to manage the interaction between these modules over time. And, OMNeT++ is optimized for scenarios where detailed analysis of network performance and protocol behavior is required, which justifies its higher resource consumption. We tend to think that this behavior might be attributed to the differences in how the two simulation environments manage memory and CPU resources. OMNet++ may allocate and reserve more memory upfront, leading to higher VMRSS readings, but it might be more efficient in managing computational tasks, thus resulting in lower CPU usage. In contrast, NS-3 could be using less memory overall but may require more CPU processing power to handle similar tasks, especially if it is more CPU-intensive in its execution of network simulations.

Emulators, on the other hand, like Mininet and GNS3, attempt to replicate real-world network conditions by running actual networking code on virtual or physical machines. This real-time execution makes them much closer to a production environment, leading to different, specific, more performance characteristics compared to simulators.

In short, Mininet is not always "mini". It emulates networks using lightweight virtualization to create virtual hosts, switches, and links that can run actual networking software. Because Mininet operates in real-time and interfaces directly with the underlying operating system, it tends to have lower CPU overhead compared to GNS3 or discrete-event simulators in a more smaller network scale like NS-3 and OMNeT++ . However, the trade-off is in memory usage, where the overhead of managing multiple containers, virtual interfaces,

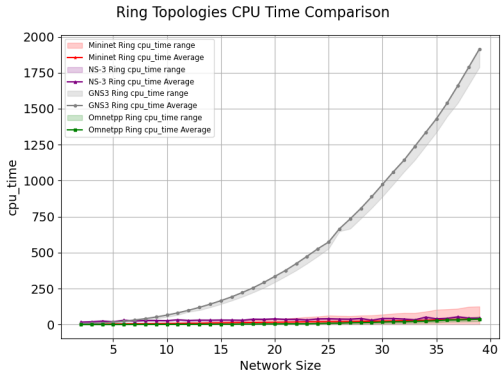

Figure 5.17: Comparison of ring topology CPU usage variation with network scale across different simulators/emulators

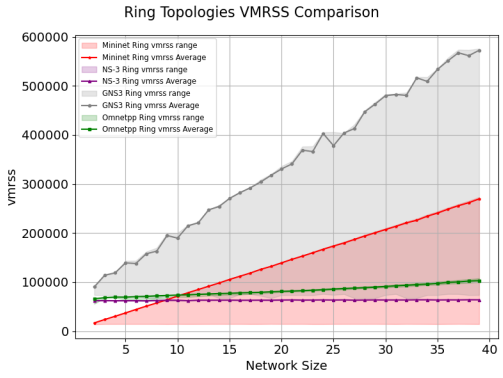

Figure 5.18: Comparison of ring topology memory usage variation with network scale across different simulators/emulators

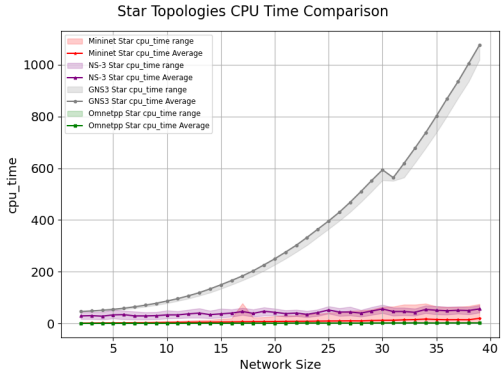

Figure 5.19: Comparison of star topology CPU usage variation with network scale across different simulators/emulators

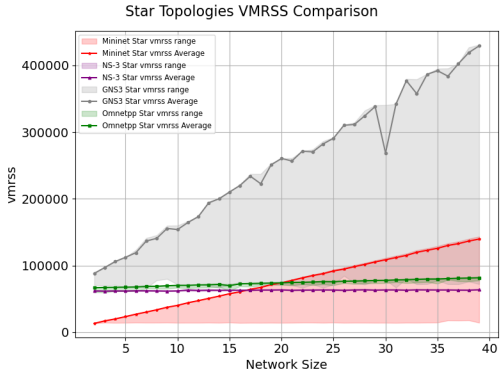

Figure 5.20: Comparison of star topology memory usage variation with network scale across different simulators/emulators

and network namespaces can become significant, particularly in complex topologies like Mesh. Additionally, Mininet's performance is highly dependent on the host system's resources, and its ability to accurately emulate network delays, bandwidth limitations, and packet losses relies heavily on the fidelity of the underlying virtualized environment. Each Mininet node, whether a host or a switch, corresponds to an independent process. When creating a large network topology, the system must manage these individual processes, which naturally leads to higher VMRSS and CPU usage.

GNS3 goes a step further by allowing the emulation of entire operating systems and real networking devices (such as old Cisco routers and switches) using virtual machines

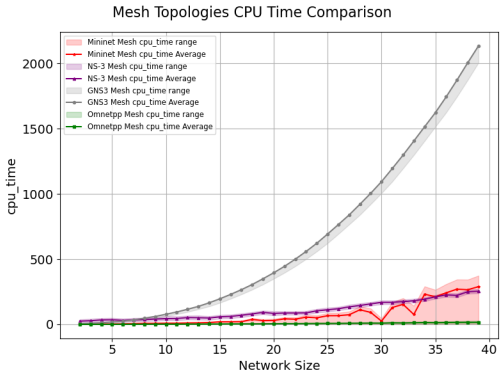

Figure 5.21: Comparison of mesh topology CPU usage variation with network scale across different simulators/emulators

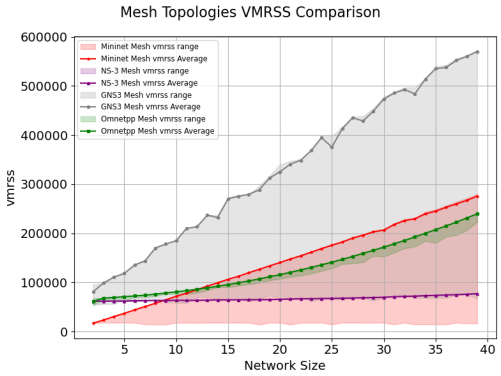

Figure 5.22: Comparison of mesh topology memory usage variation with network scale across different simulators/emulators

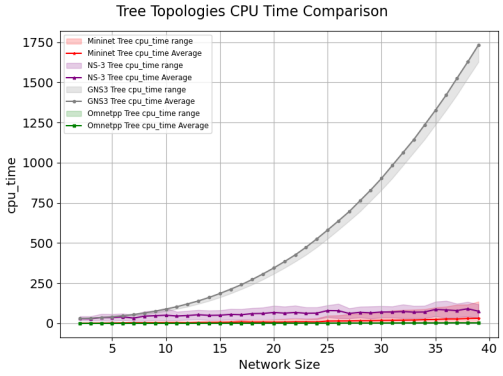

Figure 5.23: Comparison of tree topology CPU usage variation with network scale across different simulators/emulators

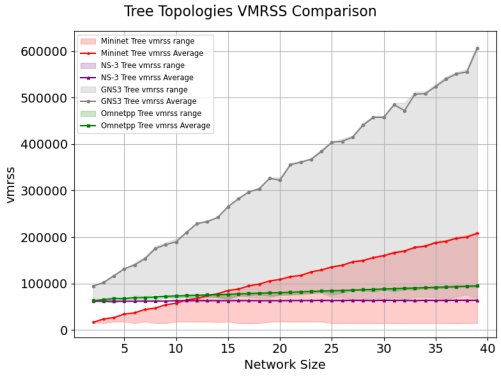

Figure 5.24: Comparison of tree topology memory usage variation with network scale across different simulators/emulators

and software like QEMU or VMware. This approach provides a higher degree of realism since the actual device software is being run, but it comes at a substantial cost in terms of CPU and memory usage. Emulating a complex topology in GNS3 can lead to high CPU utilization as each virtual device needs to process real network packets, run full operating systems, and manage interfaces in real-time. Memory consumption is also significant since each virtual device has its own memory requirements, including the operating system, application processes, and packet buffers. This makes GNS3 particularly resource-hungry compared to both Mininet and traditional simulators like NS-3 and OMNeT++.

The performance overhead differences between simulators like NS-3 and OMNeT++,

and emulators like Mininet and GNS3, can be summarized as follows to answer RQ2-1:

- **CPU Utilization**: Simulators generally require more CPU resources to manage and process simulated events, especially in complex or large-scale simulations. Emulators, while also CPU-intensive, tend to consume CPU in a more predictable and real-time manner, reflecting the execution of real networking processes and operating systems.
- **Memory Usage**: Emulators tend to use more memory due to the need to allocate resources for running virtual machines, containers, or other forms of virtualization. This includes the memory overhead of the operating system, virtual network interfaces, and other processes associated with emulating a real network environment. Simulators, on the other hand, typically have lower memory requirements as they abstract away many of the low-level details of network operation, focusing instead on the logical representation of network behavior.

6

Discussion

6.1 Flexibility and Scalability

As we mentioned, flexibility andquirements for developing or implementing a successful 6G NDTs system. Our research question also raises doubts about this scalability can be important. Although our experiment cannot directly quantify this part, our experimental setting includes dynamic deployment, so we can indirectly see the flexibility and scalability of these tools.

NS-3 is particularly well-suited to meet the dynamic deployment needs of NDTs due to its flexible discrete-event simulation architecture. NS-3 allows for network configurations and topologies to be adjusted dynamically during runtime, which is crucial for 6G networks. This capability supports the frequent updates and reconfigurations required in NDTs, offering rapid scalability and adaptability without the need to restart the entire simulation. Additionally, NS-3's ability to interface with real networks allows for hybrid scenarios where simulated and real components interact seamlessly, further enhancing its flexibility in dynamic deployments. Combined with some other RAN extensions, NS-3 can actually handle both wired and wireless parts to a certain extent (49, 50).

OMNet++, on the other hand, is not as suited for dynamic deployment in NDTs. The network topology in OMNet++ is defined using Network Description files before the simulation begins, and hard to be altered during execution. This pre-defined structure limits the ability to adapt to changes on-the-fly, which is a significant drawback in the context of 6G NDTs where network elements need to be reconfigured frequently even it als has wireless, especially mmWave extension (51). GNS3 offers high-fidelity simulations using real network device images but lacks the dynamic scaling and orchestration capabilities of tools like Kubernetes, making it less effective for scenarios requiring rapid deployment and

reconfiguration. Mininet, while providing a lightweight and scalable network emulation environment, does not offer the same depth of dynamic reconfiguration as NS-3, limiting its applicability in highly dynamic NDTs.

Mininet's flexibility is somewhat specialized to SDN environments, and the physical resources of the host machine ultimately limit its scalability. As the network complexity increases, Mininet's performance may degrade, making it less ideal for very large or highly detailed simulations. GNS3, on the other hand, provides significant flexibility by allowing the emulation of real network devices using actual hardware images. This makes GNS3 ideal for scenarios that require high-fidelity simulations closely resembling real-world network environments. However, this flexibility comes at a cost. The performance overhead of emulating real devices is substantial, consuming significant CPU and memory resources, especially as the number of devices increases. This high resource demand limits GNS3's scalability and can make large-scale simulations impractical without considerable hardware support.

Given the performance demands of GNS3, it raises the question of whether it's worth using such a resource-intensive tool for emulation when container orchestration platforms like Kubernetes offer a more scalable and efficient approach. Kubernetes can manage large numbers of containers with far less overhead, allowing for rapid deployment, scaling, and dynamic reconfiguration. While GNS3 provides unparalleled realism at the hardware level, for many use cases in 6G Network Digital Twins, the flexibility and scalability provided by container orchestration might outweigh the benefits of GNS3's high-fidelity.

6.2 Modularity and Extensibility

When it comes to simulation fidelity, GNS3 stands out for its ability to emulate networks using actual device images, providing hardware-level realism that is unmatched by purely software-based simulators. This makes GNS3 particularly valuable for NDTs that require precise, high-fidelity modeling of network hardware and behavior. However, GNS3's focus on hardware-centric emulation means it may lack the modularity and extensibility necessary for integrating new features or adapting to evolving 6G standards, especially compared to more software-oriented tools.

NS-3 also offers high fidelity through its detailed simulation of network protocols and behaviors. While it may not match GNS3 in terms of hardware-level realism, its support for modularity and extensibility is exceptional. NS-3 allows users to extend the simulator by adding new modules or integrating with external systems, making it highly adaptable to

the evolving needs of 6G NDTs. This is particularly useful for developing custom protocols or testing new network configurations. Furthermore, NS-3's capability to rapidly generate large-scale datasets without real-time constraints is beneficial for creating training data for machine learning applications, which are often a component of NDTs.

In contrast, OMNet++, while modular and extensible in terms of adding new components or protocols, is less flexible once the simulation has started due to its reliance on static network definitions. This limitation hinders its ability to support the dynamic and evolving nature of NDTs.

7

Conclusion

This thesis presented a high-level architectural framework for Network Digital Twins (NDTs) tailored for emerging 6G networks alongside a quantitative comparative analysis of various network simulators and emulators with respect to their performance overheads across different configurations and operational scenarios. The proposed 6G NDTs system architecture, comprising the Physical Twin Layer, Digital Twin Layer, and Application and Service Layer, thereby enabling more dynamic and resilient network management and future vision.

Our experimental evaluation across platforms such as NS-3, OMNeT++, Mininet, and GNS3 demonstrated notable differences in terms of CPU and memory utilization. Mininet was found to be relatively resource-efficient in emulators, whereas GNS3, despite its higher emulation fidelity, required significantly more resources, highlighting the trade-offs between simulation depth and resource consumption. In contrast, network simulators like NS-3 and OMNet++ are more resource-efficient. However, there are some differences. The performance of OMNet++ seems to be relatively more sensitive to the network structure and size, although the CPU usage is relatively low. While, NS-3 is more stable and has an overall lower memory usage.

The integration of NDTs within these simulation environments revealed that the choice of simulation and emulation tools heavily influences the performance of digital twins. In the experiment, we also found that their extensibility, scalability, flexibility and modularity vary.

However, the study's scope was constrained by the limited range of network topologies and scenarios tested, suggesting the need for further research to explore the applicability of the proposed NDT framework across more diverse network conditions. Future work should

also consider the rapid evolution of network technologies and the other components to be modeled, which could significantly influence the effectiveness of NDTs implementations.

In conclusion, this research contributes to the foundational understanding of how NDTs can be effectively implemented within next-generation 6G networks, providing an idea for future developments in network management and optimization. The result of the comparative tool analysis can also assist in making informed decisions regarding tool selection in different scenario.

References

[1] Shuping Dang, Osama Amin, Basem Shihada, and Mohamed-Slim Alouini. **What should 6G be?** *Nature Electronics*, **3**(1):20–29, 2020. 1, 5

[2] Yu Zheng, Sen Yang, and Huanchong Cheng. **An application framework of digital twin and its case study**. *Journal of Ambient Intelligence and Humanized Computing*, **10**(3):1141–1153, March 2019. 1, 5

[3] Xingqin Lin, Lopamudra Kundu, Chris Dick, Emeka Obiodu, Todd Mostak, and Mike Flaxman. **6G Digital Twin Networks: From Theory to Practice**. *IEEE Communications Magazine*, **61**(11):72–78, November 2023. 1, 15

[4] Eric J Tuegel, Anthony R Ingraffea, Thomas G Eason, and S Michael Spottswood. **Reengineering aircraft structural life prediction using a digital twin**. *International Journal of Aerospace Engineering*, **2011**, 2011. 5

[5] Michael Grieves. **Digital twin: manufacturing excellence through virtual factory replication**. *White paper*, **1**(2014):1–7, 2014. 5

[6] Michael Grieves and John Vickers. **Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems**. In Franz-Josef Kahlen, Shannon Flumerfelt, and Anabela Alves, editors, *Transdisciplinary Perspectives on Complex Systems: New Findings and Approaches*, pages 85–113. Springer International Publishing, Cham, 2017. 5

[7] Yiwen Wu, Ke Zhang, and Yan Zhang. **Digital Twin Networks: A Survey**. *IEEE Internet of Things Journal*, **8**(18):13789–13804, September 2021. Conference Name: IEEE Internet of Things Journal. 5

[8] Antonino Masaracchia, Vishal Sharma, Berk Canberk, Octavia A. Dobre, and Trung Q. Duong. **Digital Twin for 6G: Taxonomy, Research**

Challenges, and the Road Ahead. *IEEE Open Journal of the Communications Society*, **3**:2137–2150, 2022. 5

[9] Yuqian Lu, Chao Liu, I Kevin, Kai Wang, Huiyue Huang, and Xun Xu. **Digital Twin-driven smart manufacturing: Connotation, reference model, applications and research issues**. *Robotics and computer-integrated manufacturing*, **61**:101837, 2020. 5

[10] Jiewu Leng, Hao Zhang, Douxi Yan, Qiang Liu, Xin Chen, and Ding Zhang. **Digital twin-driven manufacturing cyber-physical system for parallel controlling of smart workshop**. *Journal of Ambient Intelligence and Humanized Computing*, **10**(3):1155–1166, March 2019. 5

[11] Sagheer Khan, Tughrul Arslan, and Tharmalingam Ratnarajah. **Digital twin perspective of fourth industrial and healthcare revolution**. *Ieee Access*, **10**:25732–25754, 2022. 5

[12] Tianze Sun, Xiwang He, and Zhonghai Li. **Digital twin in healthcare: Recent updates and challenges**. *Digital Health*, **9**:20552076221149651, 2023. 5

[13] Syed Mobeen Hasan, Kyuhyup Lee, Daeyoon Moon, Soonwook Kwon, Song Jinwoo, and Seojoon Lee. **Augmented reality and digital twin system for interaction with construction machinery**. *Journal of Asian Architecture and Building Engineering*, **21**(2):564–574, 2022. 5

[14] Li Deren, Yu Wenbo, and Shao Zhenfeng. **Smart city based on digital twins**. *Computational Urban Science*, **1**:1–11, 2021. 5

[15] Roberto Minerva, Gyu Myoung Lee, and Noel Crespi. **Digital twin in the IoT context: A survey on technical features, scenarios, and architectural models**. *Proceedings of the IEEE*, **108**(10):1785–1824, 2020. 5

[16] Huan X. Nguyen, Ramona Trestian, Duc To, and Mallik Tatipamula. **Digital Twin for 5G and Beyond**. *IEEE Communications Magazine*, **59**(2):10–15, February 2021. Conference Name: IEEE Communications Magazine. 5, 6, 15

[17] Paul Almasan, Miquel Ferriol-Galmes, Jordi Paillisse, Jose Suarez-Varela, Diego Perino, Diego Lopez, Antonio Agustin Pastor Perales, Paul Harvey, Laurent Ciavaglia, Leon Wong, Vishnu Ram, Shihan Xiao,

Xiang Shi, Xiangle Cheng, Albert Cabellos-Aparicio, and Pere Barlet-Ros. **Network Digital Twin: Context, Enabling Technologies, and Opportunities**. *IEEE Communications Magazine*, **60**(11):22–27, November 2022. 5

[18] Aldebaro Klautau, Pedro Batista, Nuria González-Prelcic, Yuyang Wang, and Robert W Heath. **5G MIMO data for machine learning: Application to beam-selection using deep learning**. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–9. IEEE, 2018. 6

[19] Nandish P. Kuruvatti, Mohammad Asif Habibi, Sanket Partani, Bin Han, Amina Fellan, and Hans D. Schotten. **Empowering 6G Communication Systems With Digital Twin Technology: A Comprehensive Survey**. *IEEE Access*, **10**:112158–112186, 2022. 6

[20] Hnin Pann Phyu, Diala Naboulsi, and Razvan Stanica. **Machine Learning in Network Slicing—A Survey**. *IEEE Access*, **11**:39123–39153, 2023. Conference Name: IEEE Access. 6

[21] Mustufa Haider Abidi, Hisham Alkhalefah, Khaja Moiduddin, Mamoun Alazab, Muneer Khan Mohammed, Wadea Ameen, and Thippa Reddy Gadekallu. **Optimal 5G network slicing using machine learning and deep learning concepts**. *Computer Standards & Interfaces*, **76**:103518, June 2021. 6

[22] Yu Gong, Yifei Wei, Zhiyong Feng, F. Richard Yu, and Yan Zhang. **Resource Allocation for Integrated Sensing and Communication in Digital Twin Enabled Internet of Vehicles**. *IEEE Transactions on Vehicular Technology*, **72**(4):4510–4524, April 2023. Conference Name: IEEE Transactions on Vehicular Technology. 6

[23] Cheng-Xiang Wang, Xiaohu You, Xiqi Gao, Xiuming Zhu, Zixin Li, Chuan Zhang, Haiming Wang, Yongming Huang, Yunfei Chen, Harald Haas, John S. Thompson, Erik G. Larsson, Marco Di Renzo, Wen Tong, Peiying Zhu, Xuemin Shen, H. Vincent Poor, and Lajos Hanzo. **On the Road to 6G: Visions, Requirements, Key Technologies, and Testbeds**. *IEEE Communications Surveys & Tutorials*, **25**(2):905–974, 2023. 7

[24] Ahmed Slalmi, Hasna Chaibi, Abdellah Chehri, Rachid Saadane, and Gwanggil Jeon. **Toward 6G: Understanding network requirements and key performance indicators**. 7

[25] Yadong Zhang, Haibin Zhang, Yunlong Lu, Wen Sun, Lan Wei, Yan Zhang, and Bin Wang. **Adaptive Digital Twin Placement and Transfer in Wireless Computing Power Network**. *IEEE Internet of Things Journal*, **11**(6):10924–10936, March 2024. Conference Name: IEEE Internet of Things Journal. 7

[26] András Varga. **Discrete event simulation system**. In *Proc. of the European Simulation Multiconference (ESM'2001)*, **17**, 2001. 7

[27] K Agalianos, ST Ponis, E Aretoulaki, G Plakas, and O Efthymiou. **Discrete event simulation and digital twins: review and challenges for logistics**. *Procedia Manufacturing*, **51**:1636–1641, 2020. 8

[28] Daniel Schubert, Benedikt Jaeger, and Max Helm. **Network emulation using Linux network namespaces**. *Network*, **57**, 2019. 8

[29] Sohaib Manzoor, Mahak Manzoor, Hira Manzoor, Durr E Adan, and Muhammad Akbar Kayani. **Which Simulator to Choose for Next Generation Wireless Network Simulations? NS-3 or OMNeT++**. *Engineering Proceedings*, **46**(1):36, 2023. 10, 11

[30] George F Riley and Thomas R Henderson. **The ns-3 network simulator**. In *Modeling and tools for network simulation*, pages 15–34. Springer, 2010. 10

[31] Lelio Campanile, Marco Gribaudo, Mauro Iacono, Fiammetta Marulli, and Michele Mastroianni. **Computer network simulation with ns-3: A systematic literature review**. *Electronics*, **9**(2):272, 2020. 10

[32] Ravi Kishore Kodali and Bysani Kirti. **NS-3 Model of an IoT network**. In *2020 IEEE 5th International Conference on Computing Communication and Automation (ICCCA)*, pages 699–702. IEEE, 2020. 10

[33] George S Fishman. *Discrete-event simulation: modeling, programming, and analysis*, **537**. Springer, 2001. 10

[34] Arnold BD Kinabo, Joyce B Mwangama, and Albert A Lysko. **Towards wi-fi-based Time Sensitive Networking using OMNeT++/NeSTiNg simulation models**. In *2021 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–6. IEEE, 2021. 11

[35] Abdulmueen Alrashide, Mahmoud S Abdelrahman, Ibtissam Kharchouf, and Osama A Mohammed. **GNS3 communication network emulation for substation GOOSE based protection schemes**. In *2022 IEEE International Conference on Environment and Electrical Engineering and 2022 IEEE Industrial and Commercial Power Systems Europe (EEEIC/I&CPS Europe)*, pages 1–6. IEEE, 2022. 12

[36] Huadan Zheng. **A Multi-host Joint Network Simulation Method Based on GNS3**. In *Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering*, pages 1141–1146, 2020. 13

[37] Brendan Choi. **GNS3 Basics**. In *Introduction to Python Network Automation Volume I-Laying the Groundwork: The Essential Skills for Growth*, pages 639–752. Springer, 2024. 13

[38] Chong Zhou, Dong Li, Wenjie Han, and Bin Qin. **Development of a Power Supply Control System and Virtual Simulation Based on Docker**. In *2020 IEEE 1st China International Youth Conference on Electrical Engineering (CIYCEE)*, pages 1–7. IEEE, 2020. 13

[39] Latif U. Khan, Walid Saad, Dusit Niyato, Zhu Han, and Choong Seon Hong. **Digital-Twin-Enabled 6G: Vision, Architectural Trends, and Future Directions**. *IEEE Communications Magazine*, **60**(1):74–80, January 2022. 15

[40] Wen Sun, Wenqiang Ma, Yu Zhou, and Yan Zhang. **An Introduction to Digital Twin Standards**. *GetMobile: Mobile Computing and Communications*, **26**(3):16–22, October 2022. 16

[41] Gergely Pongrácz, Attila Mihály, István Gódor, Sándor Laki, Anastasios Nanos, and Chrysa Papagianni. **Towards extreme network KPIs with programmability in 6G**. In *Proceedings of the Twenty-fourth International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*, pages 340–345, 2023. 19

[42] Cyril Shih-Huan Hsu, Danny De Vleeschauwer, and Chrysa Papagianni. **SLA Decomposition for Network Slicing: A Deep Neural Network Approach**. *IEEE Networking Letters*, 2023. 20

[43] João Paulo Tavares Borges, Ailton Pinto De Oliveira, Felipe Henrique Bastos E Bastos, Daniel Takashi Né Do Nascimento Suzuki, Emerson Santos De Oliveira Junior, Lucas Matni Bezerra, Cleverson Veloso Nahum, Pedro dos Santos Batista, and Aldebaro Barreto Da Rocha Klautau Júnior. **Reinforcement learning for scheduling and MIMO beam selection using Caviar simulations**. In *2021 ITU Kaleidoscope: Connecting Physical and Virtual Worlds (ITU K)*, pages 1–7. IEEE, 2021. 21

[44] Ailton Oliveira, Felipe Bastos, Isabela Trindade, Walter Frazao, Arthur Nascimento, Diego Gomes, Francisco Muller, and Aldebaro Klautau. **Simulation of machine learning-based 6G systems in virtual worlds**. *arXiv preprint arXiv:2204.09518*, 2022. 21

[45] Ilan Correa, Ailton Oliveira, Bojian Du, Cleverson Nahum, Daisuke Kobuchi, Felipe Bastos, Hirofumi Ohzeki, João Borges, Mohit Mehta, Pedro Batista, et al. **Simultaneous beam selection and users scheduling evaluation in a virtual world with reinforcement learning**. *ITU Journal on Future and Evolving Technologies*, **3**(2):202–213, 2022. 21

[46] Sheng-Ming Tang, Cheng-Hsin Hsu, Zhigang Tian, and Xin Su. **An aerodynamic, computer vision, and network simulator for networked drone applications**. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 831–833, 2021. 21

[47] Esteban Egea-Lopez, Fernando Losilla, Juan Pascual-Garcia, and Jose Maria Molina-Garcia-Pardo. **Vehicular networks simulation with realistic physics**. *IEEE Access*, **7**:44021–44036, 2019. 21

[48] Andres Ruz-Nieto, Esteban Egea-Lopez, Jose-Maria Molina-Garcia-Pardo, and Jose Santa. **A 3D simulation framework with ray-tracing propagation for LoRaWAN communication**. *Internet of Things*, **24**:100964, 2023. 21

[49] Marco Mezzavilla, Sourjya Dutta, Menglei Zhang, Mustafa Riza Akdeniz, and Sundeep Rangan. **5G mmWave module for the ns-3 network simulator**. In *Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 283–290, 2015. 51

[50] Natale Patriciello, Sandra Lagen, Lorenza Giupponi, and Biljana Bojovic. **An Improved MAC Layer for the 5G NR ns-3 module**. In *Proceedings of the 2019 Workshop on ns-3*, pages 41–48, 2019. 51

[51] Giovanni Nardini, Giovanni Stea, Antonio Virdis, Dario Sabella, et al. **Simu5G: a system-level simulator for 5G networks**. In *Proceedings of the 10th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, pages 68–80. SciTePress, 2020. 51

Appendix