POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master Thesis

Efficient Service Provisioning in the Musical Metaverse Using a Custom Network Simulator

Supervisors
Prof. Cristina Rottondi (Politecnico di Torino)
Prof. Sebastian Troia (Politecnico di Milano)
Prof. Omran Ayoub (Universita Svizzera Italiana)

Candidate
Ali Al Housseini

Academic year 2024-2025

Abstract

Addressing the critical challenge of efficiently allocating network resources to support the demanding requirements of the Musical Metaverse, this thesis leverages the principles of Network Virtualization and Virtual Network Embedding (VNE) to propose a novel framework for optimizing the placement and routing of musical metaverse services. The unique characteristics of the Musical Metaverse, including ultra-low latency communication, stringent Quality of Service (QoS) requirements, and the need for precise synchronization in multi-user interactions, necessitate a specialized approach beyond traditional VNE solutions. To this end, the thesis introduces the Musical Metaverse Optimization (MusMOPT) model, which extends classical VNE to account for these specific demands. Furthermore, a dedicated simulation environment, SiMusMet, is developed to rigorously evaluate the proposed models and algorithms. SiMusMet provides a comprehensive platform for configuring network topologies, simulating various scenarios, and assessing performance metrics relevant to the Musical Metaverse. The thesis also details the design of a heuristic placement and routing algorithm, incorporating techniques such as community detection and probabilistic iterative refinement, to efficiently map musical metaverse service graphs onto underlying cloud network infrastructures. Through extensive simulations and analysis, this research demonstrates the effectiveness of the proposed MusMOPT model and heuristic algorithm in achieving near-optimal resource utilization and ensuring a high-quality user experience within the Musical Metaverse.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisors, Professoressa Cristina Rottondi, Professor Sebastian Troia, and Professor Omran Ayoub for their invaluable guidance and unwavering support. Some encounters leave a lasting mark — through their mentorship, I discovered my potential and was inspired to dream bigger. Their influence profoundly shaped not only this work but also my academic and professional aspirations.

A special thanks goes to my family for their constant encouragement. Their belief in me, even during the most challenging moments, has been a strong source of motivation.

Above all, my deepest appreciation goes to Rim—for her presence, patience, and love. Without her, none of this would have had the same meaning.

Contents

List of Tables

List of Figures

Nomenclature

3C	Connected Collaborative Computing
6-DOF	Six degrees of freedom
DAG	Directed Acyclic Graph
DC	Data Center
HMD	Head mounted display
LB	Load-Balancing
MILP	Mixed Integer Linear Formulating
MM	Musical Metaverse
NMP	Networked Musical Performance
PN	Physical Network
QoE	Quality Of Experience
QoS	Quality of Service
SN	Subtrate Network
SP	Service Provider
VLE	Virtual Link Embedding
VME	Virtual Machine Embedding
VNE	Virutal Network Embedding
VNO	Virtual Network Operator
VNP	Virtual Network Provider
VNR	Virtual Network Request

The Scientific Revolution has not been a revolution of ***knowledge****.*
It has been above all a revolution of ***ignorance****.*
The great discovery that launched the Scientific Revolution, was the ***discovery*** *that humans do not know,*
the ***answers*** *to their most important questions.*

[Yuval Noah Harari, Sapiens: A Brief History of Humankind]

Chapter 1

General Introduction

1.1 Summary

In an increasingly interconnected world, digital platforms are transforming how we interact, work, and entertain. This evolution is leading towards the emergence of the metaverse, a persistent virtual world that mixes physical and digital realities. Within this context, music plays a pivotal role, giving rise to the "Musical Metaverse" (MM).

The MM promises immersive, interactive experiences, from collaborative performances across continents to massive virtual concerts with thousands of participants. It envisions a future where musicians and audiences can co-create and engage in real-time, transcending geographical boundaries. However, realizing the full potential of the Musical Metaverse presents formidable challenges, particularly concerning its underlying network infrastructure. Unlike traditional online activities, musical interactions in the MM demand ultra-low latency, stringent Quality of Service (QoS) guarantees, and precise synchronization across multiple users and devices. Even a few milliseconds of delay or minor packet loss can disrupt the illusion of co-presence and compromise the musical timing, leading to a degraded user experience.

Current network architectures and conventional Virtual Network Embedding (VNE) models, designed primarily for less demanding applications, fall short in addressing these unique requirements. They often lack the necessary flow-level intelligence, struggle with dynamic, multi-modal data streams, and are not suitable for handling the complex interplay of compute and network resources across a distributed edge-cloud continuum. This highlights a critical need for a specialized approach to efficiently allocate and orchestrate network resources within the Musical Metaverse.

1.2 Contributions

This thesis addresses the above-mentioned challenges by introducing a **novel** framework and **associated tools** for *efficient service provisioning in the Musical Metaverse.* Our **primary** contributions are as follows:

- **Musical Metaverse Optimization (MusMOPT) Framework**: We propose

MusMOPT, a novel optimization framework that reconceptualizes how services are mapped onto network infrastructure. Unlike classical VNE, MusMOPT treats musical metaverse services as directed acyclic graphs (DAGs) of information processing, enabling a unified approach to service placement, routing, and resource allocation. This framework inherently supports multicast distribution, understands information flows, and integrates compute-network decisions, directly addressing the unique demands of the MM.

- **Two-Layer Abstraction Model**: We introduce a two-layer abstraction for MM service design: a graph-layer responsible for the strategic placement of functions and routing of commodities, and a data-layer that manages real-time information flow behaviors, reacting to dynamic network conditions like jitter and congestion. This decomposition allows for both long-term optimization and real-time adaptability.
- **SiMusMet Simulator**: To validate our theoretical advancements, we developed SiMusMet, a Python-based discrete-event simulator. SiMusMet provides a robust and extensible environment for simulating dynamic service embeddings, fine-grained traffic flows under realistic network impairments, and comprehensive evaluation of QoS and Quality of Experience (QoE) metrics in musical metaverse scenarios.
- **Topology-Aware Cost-Load Heuristic Algorithm**: Recognizing the computational complexity of optimal solutions for large-scale MM deployments, we designed a novel heuristic algorithm for placement and routing. This heuristic balances computational speed and efficiency with cost-consciousness and load restraint, offering near-optimal results in practical timeframes, making it suitable for real-time decision-making in dynamic MM environments.
- **Comprehensive Evaluation and Analysis**: We benchmark our proposed heuristic against two Mixed-Integer Linear Programming (MILP) formulations (Min-Cost and Load-Balancing) across various cloud network topologies and user scales. Our extensive simulations demonstrate the practical value of the heuristic, showcasing its superior runtime performance while maintaining competitive cost, load fairness, and resource utilization compared to optimal, but computationally expensive, MILP solutions.

The remainder of this thesis is structured as follows: **Chapter 2** provides an introduction to communication networks, laying the groundwork for understanding the underlying infrastructure. **Chapter 3** delves into the fundamentals of Virtual Network Embedding (VNE), outlining its problem formulation, components, and inherent complexities. **Chapter 4** introduces the concept of the Musical Metaverse, detailing its technological foundations and the specific networking infrastructure requirements. **Chapter 5** presents the Musical Metaverse Optimization (MusMOPT) model in detail, highlighting the limitations of classical VNE and explaining our novel approach. **Chapter 6** focuses on Service-Graph Foundations, describing the interaction model for pre-recorded concerts and the two-layer problem decomposition. **Chapter 7** introduces the MUSMET Simulator, detailing its design principles, core modules, and capabilities for simulating MM scenarios. **Chapter 8** discusses the Placement and Routing Policy Design, including the general MILP formulation and our proposed heuristic algorithm. **Chapter 9** presents the Simulation Results and Analysis, comparing the performance of our heuristic against MILP baselines across various experimental setups. **Chapter 10** concludes the thesis by summarizing our findings and outlining future work.

Chapter 2

Introduction to Virtualization in Telecommunication Networks

2.1 Why does this matter?

Imagine it's Friday night. You curl up on the sofa, open your favorite streaming app, and press "Play". In that instant, millions of tiny data packets sprint across the globe: leaping from the content provider's data center, zig-zagging through long-haul fibre, dodging congestion at Internet exchange points, and finally hopping across your home Wi-Fi to paint an ultra-high-definition scene on your screen. To you, it feels effortless: *click*, *watch*, *relax*. But beneath that simplicity lies a vast, hidden machinery: routers, switches, cables, and servers all working in concert, orchestrating a seamless experience that reaches your screen before you even think to question how.

> Why does this matter? Because every *"simple"* online action, whether streaming a film, scrolling social media, or backing up files to the cloud, rests on the same fundamental question: **how do we move data from A to B, on time, every time, for everybody?**

Now consider a different scenario: you're playing an online multiplayer game. Your actions (e.g. firing a virtual weapon, dodging an opponent) must reach a game server in milliseconds. That server must update the game state and relay it back to you and all other players just as quickly. Every delay, every spike in latency, becomes a lag that could mean the difference between victory and defeat. To ensure this low-latency interaction, your input is passed through load balancers, firewalls, and routing optimizers, each acting as a service embedded within the network.

In both scenarios, video streaming and online gaming, users are not simply "using the Internet." They are **requesting network services**: services that deliver data continuously with guarantees. HD video needs sustained throughput; online games need ultra-low delay. But these aren't standalone services. They are made possible by a shared, global physical network that must dynamically and intelligently allocate resources for millions of **concurrent demands**.

This underlying infrastructure, known as the substrate network, consists of physical hardware (switches, routers, fiber-optic cables, data centers, etc..). It's the real-world canvas on which virtual services are painted. Every time a user makes a service request, the substrate network is responsible for stitching together a path across this physical mesh and allocating the right amount of bandwidth, processing power, and sometimes even custom service nodes along the way.

But here's the challenge: the physical network must support not just one, but thousands - even millions - of different service requests simultaneously, each with different requirements. Some need speed, others need bandwidth, still others require specific types of processing along the path (like firewalls, caches, or deep packet inspection). And all of this has to happen **efficiently, fairly, and in real time**. So how does the network make this possible?

2.2 Network Virtualization

The answer lies in a powerful concept that has transformed the way networks are built and managed: **network virtualization**.

At its core, network virtualization is the process of abstracting the physical infrastructure of a network (routers, switches, cable and servers) into flexible, programmable components that can be recombined and reused to meet different service demands. It's a bit like urban planning: imagine a city with roads, power lines, and utility services. Traditionally, if you wanted to build a new neighborhood, you'd need to lay down new roads and wires for that specific purpose. But with virtualization, it's as if the same roads and infrastructure can be instantly reshaped to serve any neighborhood design, residential one day, commercial the next, without laying a single new brick.

In networking terms, this means that instead of dedicating physical hardware to a single service (like video streaming or online gaming), the same physical network can be **sliced** into multiple, logically separate **virtual networks**, each customized for a particular use case. These slices are isolated from one another, ensuring that heavy traffic in one (say, a video-on-demand platform) doesn't interfere with performance in another (say, real-time financial trading).

Each virtual network consists of virtual nodes (representing functions like firewalls, caches, or processing units) and virtual links (representing bandwidth, latency, or quality-of-service guarantees). These virtual components are mapped onto the real, physical infrastructure in a way that's invisible to the end user, but carefully orchestrated by the network operator or orchestration system.

This approach has several major benefits:

- **Flexibility**: New services can be deployed without having to physically reconfigure hardware.
- **Isolation**: Services can run independently, with guaranteed performance and security.
- **Efficiency**: Resources are shared dynamically, increasing overall network utilization.

- **Scalability**: Operators can support millions of users and services without building a separate network for each.

Network virtualization is made possible by two enabling technologies: **Software-Defined Networking (SDN)** and **Network Functions Virtualization (NFV)**. SDN decouples the control logic (which decides how traffic flows) from the data plane (which forwards the traffic), enabling centralized, programmable control of the network. NFV, meanwhile, replaces dedicated hardware appliances - like firewalls or load balancers - with **Virtual Network Functions (VNFs)** that run on general-purpose servers. Together, SDN and NFV make the network programmable, modular, and service-aware.

For end users, this architectural shift is invisible, but essential. When you start a video stream or connect to a game server, you're not just sending packets. You're implicitly requesting a temporary virtual network—one that meets your specific needs for speed, reliability, and responsiveness. That network is created in real time, mapped onto a set of physical routers, links, and compute nodes, and then torn down when the session ends.

But this seemingly effortless experience masks a hard technical problem. Behind every user request lies a critical challenge: **how should the network decide where and how to place each virtual service?** How can it ensure that every virtual link gets the bandwidth it needs, that virtual nodes don't overload physical servers, and that different services remain isolated while still sharing the same physical hardware?

This challenge becomes even more intricate when we realize that many services aren't just requesting bandwidth, but they're asking for *functionality*: sequences of operations that traffic must pass through to be delivered correctly and securely. Before we explore the full complexity of placing virtual networks onto physical infrastructure, let's take a closer look at a real-world example: **Service Function Chaining**.

2.3 From Service Chains to Virtual Networks

To understand how virtual networks operate in real systems, let's look at one of the most widely deployed and practical use cases: **Service Function Chaining (SFC)**.

Modern network services often require more than just moving data from point A to point B, they require traffic to pass through a specific sequence of operations, such as security checks, traffic shaping, or load distribution. These operations are performed by **network functions**, and the ordered list of them forms what is known as a **Service Function Chain**.

The Internet Engineering Task Force (IETF) defines an SFC as "an ordered set of service functions and the subsequent steering of traffic through them." In practice, each **service function** in the chain performs a distinct role. For example:

- A **firewall** may filter unwanted or malicious traffic.
- An **Intrusion Detection System (IDS)** can monitor packets for suspicious behavior.
- A **load balancer** distributes incoming requests evenly across a set of servers.

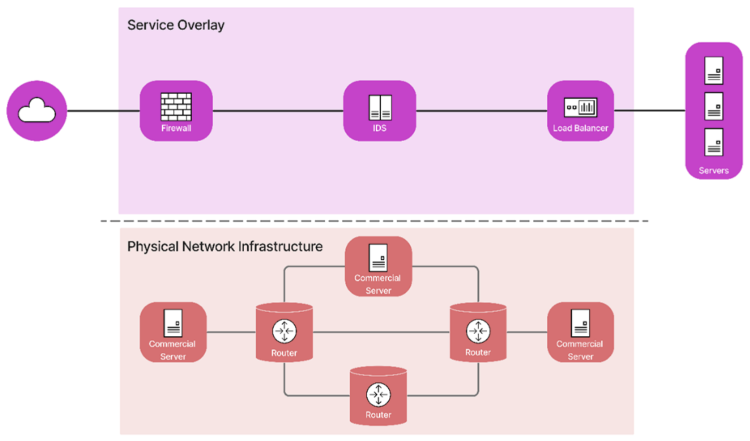


Figure 2.1. An illustration of the service function chaining example

Imagine an enterprise deploying a public-facing web application. To ensure security and performance, they might request an SFC that routes traffic first through a firewall (to block malicious requests), then through an IDS (to monitor for attacks), and finally through a load balancer (to direct traffic to one of many backend servers). Traditionally, this would require a set of physical appliances connected in a strict order: expensive, inflexible, and hard to scale.

With network virtualization, all these functions can now be implemented as software, **Virtual Network Functions (VNFs)**, running on general-purpose servers. The chain itself is enforced using software-defined forwarding rules that steer traffic through the correct VNFs, in the correct order. This allows service providers to dynamically deploy, scale, or reconfigure service chains as needed, without touching any hardware.

Crucially, this service chain represents more than a list of functions. It is a form of a **virtual network**.

Each VNF in the chain is a **virtual node**, and the traffic flow between them forms the **virtual links**. These links may carry performance constraints like bandwidth requirements to support video traffic, or latency constraints for interactive services. The entire service chain becomes a logical topology that must be embedded onto the physical infrastructure.

While SFCs often take the form of linear chains, virtual networks in general can have more complex topologies like stars, trees, meshes, or arbitrary graphs, depending on the service logic. For example:

- A content delivery platform might require a mesh of VNFs for replication and failover.
- A real-time analytics service might need a tree-shaped topology for data aggregation.

- A large-scale gaming backend might use a hub-and-spoke model to coordinate traffic among regional data centers.

Because they are software-defined, these virtual networks are highly flexible. They can be spun up on demand, scaled elastically, and removed once no longer needed. Importantly, **many virtual networks can coexist on the same physical infrastructure**. A service chain handling enterprise traffic can operate alongside a video delivery network and a multiplayer gaming backend, all isolated logically, yet sharing the same physical routers, switches, and compute nodes.

This is the essence of network virtualization: the ability to construct heterogeneous, purpose-built virtual networks over a shared, general-purpose substrate. But this powerful capability introduces a central question:

How should the network decide where and how to place all these virtual components (nodes and links) onto the physical infrastructure, while satisfying constraints and avoiding conflicts?

This is the challenge that brings us to the next chapter: the **Virtual Network Embedding Problem**.

Chapter 3

A primer on Virtual Network Embedding

Before we delve into the intricacies of the embedding problem, we first present a straightforward mathematical model that formally represents the critical components involved, namely *virtual network requests* VNRs and *substrate network* SN. Subsequently, we break down the embedding process into two sub-problems: virtual machine embedding (VME) and virtual link embedding (VLE). While various papers in the literature have provided disparate representations of VNE [1], [20], we believe that a simplified modeling approach can aid first-time readers and those unfamiliar with the domain in quickly grasping the complexities and comprehending the subsequent information presented [1].

3.1 Formulating the VNE Problem

Before formulating the VNE problem, we introduce the components involved in the embedding process. Before this, we discuss the various approaches for modeling a VNR. In this context, a directed graph representation may offer a more accurate model than the weighted undirected graph. Furthermore, the choice between directed and undirected graphs for modeling VNRs depends on the application's communication patterns and the specific goals of the embedding process. We will then offer insights on which model could be adopted based on the use case.

3.1.1 Directed vs. Undirected vs. Directed Graphs with Bidirectional Edges for VNR Modeling:

Directed graphs are optimal representations for VNRs where data flow direction is essential, as seen in content delivery networks (CDNs) and streaming services that require

[1]AI has been used as an assistant to generate the content of this chapter

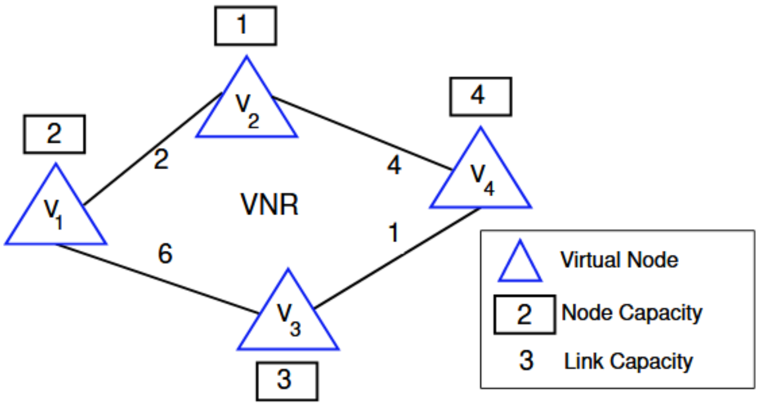


Figure 3.1. A virtual network request (VNR) with VMs and virtual links (VLs) and their respective resource demands. The value *2* of v_1 represents the resource demand of the VM, whereas the number *2* between v_1 and v_2 indicates the bandwidth demands for a VNR.

one-way data transmission from servers to users [6]. This approach captures asymmetric resource constraints, such as distinct bandwidth and latency needs, and enables prioritization for critical paths, enhancing security through directional restrictions [36]. In contrast, undirected graphs suit bidirectional communication applications, like traditional client-server models, simplifying embedding by allowing two-way traffic and improving fault tolerance through rerouting [22]. This setup also facilitates flexible resource allocation in general-use networks, like VPNs, and handles dynamic traffic changes in MEC and IoT networks without re-embedding [25]. While directed graphs with bidirectional edges offer precision for sequence sensitive tasks by highlighting dependencies among VMs, they add complexity, increasing computational demands and resource consumption, especially in more extensive networks. Managing such models requires sophisticated algorithms and may necessitate protocol updates, complicating scalability and implementation. Table 3.1 captures a comparative study of different VNE models. Note that the appropriate model selection should align with the application's communication patterns, operational dependencies, and network requirements, ensuring the most effective representation for each scenario. As most of the literature reviewed models the VNR as an undirected graph, we follow the same definition. However, the undirected graph model can be adapted to a directed graph with minimal adjustments.

Definition 1. ***(virtual network request (VNR))*** *A VNR, as illustrated in Figure 3.1, comprises multiple interacting VMs with communication dependencies captured as VLs.*

Modeling a VNR: Formally, a VNR is represented as an undirected weighted graph $\mathcal{G}^V = (\mathcal{N}^V, \mathcal{E}^V)$. The set of vertices in the graph, i.e., $\mathcal{N}^\mathcal{V} = \{v_1, v_2, ..., v_i, ...\}$ capture the set of VMs and the i^{th} VM is uniquely identified as v_i. The set of edges, i.e., $\mathcal{L}^\mathcal{V} = \{(1{,}2), (2{,}4), ..., (i, i'), ...\}$ capture the VLs interconnecting any two VMs in N^V. Note that an edge (i, i') implies that the VMs v_i and $v_{i'}$, exhibit communication dependencies. Every VM $v_i \in \mathcal{N}^{\mathcal{V}}$ requests service resources for seamless operation and this demand is captured as $d(v_i)$ and the requisite physical bandwidth resource demand of a VL (i, i') interconnecting v_i and $v_{i'}$ is captured as $d(i, i')$.

Definition 2. ***(Substrate Network (SN)))*** *A substrate network refers to the physical organization of servers, switches, and physical links providing user services.*

Modeling a SN: Formally, a SN is also modeled as an undirected weighted graph $\mathcal{G}^{\mathcal{S}} = (\mathcal{N}^{\int}, \mathcal{L}^{\mathcal{S}})$. Here, $\mathcal{N}^{\mathcal{S}} = \{s_1, s_2, s_k, ...\}$ be the set of physical servers and $\mathcal{L}^{\mathcal{S}} = \{(1{,}2), (3{,}4), ..., (l, l'), ...\}$ is the physical link set. The available resources of server $s_k \in \mathcal{N}^{\int}$ is represented as $a(s_k)$, and that of a substrate link $(l, l') \in \mathcal{L}^{\mathcal{S}}$ is captured in $a(l, l')$.

Modeling the Computational Resources: The SN and the VNR resources are often multidimensional, and different authors have adopted disparate representations. Therefore, we impose various representations adopted in the literature before delving deeper into the modeling of VNE. Ideally, the VM and server resources are multi-dimensional, comprising CPU and memory demands. Most works express the demands individually for each resource type or as resource vectors. However, such a model adds to the notational complexity. Therefore, some works in [24, 23, 26] have adopted a more simplified approach to express the multi-dimensional resources. The authors use computational resource blocks (CRBs) as initially reported in [35, 34], wherein one CRB resource corresponds to *1 CPU* core and *512 MB* of memory. A two CRB demand implies *2 CPU* cores and *1 GB* of memory. Moreover, large-scale SPs widely adopt such sizing policies as they reduce resource management complexity. We adopt this resource modeling in the report due to its obvious advantages. We would also like to mention that the above-discussed model is easily adaptable to multiple multi-dimensional resources. On the other hand, the bandwidth demand of a VL (i, i') and a physical link (l, l') is unidimensional and is captured as $d(i, i')$ and $a(l, l')$ (in Gbps).

Note that the VNE comprises two interconnected subproblems: firstly, the allocation of VMs of VNRs to substrate servers, termed as virtual machine embedding (VME), and secondly, the assignment of interconnecting VLs of VNRs to physical/substrate paths after embedding the VMs termed as virtual link embedding (VLE). Both sub-problems are challenging, computationally intractable, and are proven to be $\mathcal{NP}-Hard$ [2]. Next, we formally define each subproblem and present high-level example scenarios to assist the readers in understanding the sub-problems.

Definition 3. ***(virtual machine embedding (VME))*** *Virtual machine embedding is a function $f_{vm} : \mathcal{N}^{\mathcal{V}} \rightarrow \mathcal{N}^{\mathcal{S}}$, where in $\forall v_i \in \mathcal{N}^{\mathcal{S}}, \exists s_k \in \mathcal{N}^{\mathcal{S}}$ such that $d(v_i) \leq a(s_k)$.*

The VME sub-problem is captured in Definition 3. It essentially states that at least one host machine s_k should have enough resources (in CRBs) to accommodate the VM v_i. Ideally, a VM is assigned to only one server, but in cases of parallel workflows, a VM may be mapped onto multiple servers. However, this model is explainable for

Aspect	**Directed Graphs**	**Undirected Graphs**	**Directed Graphs with Bidirectional Edges**
Use case	CDNs and streaming services	Client–server	Sequence-sensitive operations
Data Flow	One-way data transmission	Flexible two-way traffic	Directional flow
Resource Constraints	Specific to a direction	Generic both direction	Distinct in each direction
Traffic Rerouting	Complex	Easier	Moderately complex
Management Complexity	Moderately complex	Less complex	Highly complex
Resource Consumption	Moderate	Low	Higher
Traffic Management Flexibility	Less	More	Moderate
Implementation Challenges	Moderately complex	Easier	More complex

Table 3.1. Comparative study of different virtual network representations.

applications with splittable jobs. On the other hand, a server can host multiple VMs of the same/other VNRs. However, some works in [22, 24] have disallowed such assignments to facilitate distributed services and avoid single-point failures. Additionally, for a VNR, VME is said to be complete if all its VMs (set $\mathcal{N}^{\mathcal{V}}$) obtain a successful assignment.

Definition 4. ***(virtual link embedding (VLE))*** *Virtual link embedding is a function $f_{vl}: \mathcal{L}^{\mathcal{V}} \rightarrow \mathcal{L}^{\mathcal{S}}$ between a VL and a physical/substrate link. The VL (i, i') between VMs v_i and $v_{i'}$ is assigned to a substrate paths (l, l') if $d(i, i') \leq a(l, l')$.*

In contrast to the VME, where a VM is assigned to a substrate server, VLE assigns a VL to a substrate path, constituting multiple physical links, each satisfying the VL's bandwidth requirement.

Definition 5. ***(virtual network embedding (VNE))*** *A VNR is said to be successfully embedded if all the VMs and VLs are respectively assigned substrate servers and paths satisfying their resource demands.[22, 24]*

3.1.2 Optimization Goals for VNE

VME goals may include improving resource utilization [22], consolidating virtual machines on energy-efficient servers [13], increasing the acceptance ratio [26], and maximizing service provider revenue [27]. In contrast, the objectives of VLE typically focus on meeting various quality of service measures, such as bandwidth, communication, and delay requirements [11]. Additionally, both sub-problems entail a set of constraints essential for successful VNE. While it is challenging to mathematically encompass the various objectives and constraints found in the literature, we have concentrated on modeling and selected a few of the most critical metrics.

Before formulating various objectives, we introduce two binary indicator variables, i.e., VM indicator (Equation 3.1) and VL indicator (Equation 3.2), which subsequently assist in defining the objectives and constraints of the VNE. The former is set to 1 if v_i is hosted on s_k, and the latter is set to 1 if the VL (i, i') is mapped onto the physical link (l, l').

$$x_{ik} = \begin{cases} 1 & \text{if } v_i \text{ is assigned to server } s_k, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

$$x_{l,l'}^{i,i'} = \begin{cases} 1 & \text{if VL } (i, i') \in \mathcal{L}^{\mathcal{V}} \text{ is mapped to the physical link } (l, l') \in \mathcal{L}^{\mathcal{S}} \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Objectives of VNE Schemes: We subsequently formulate some optimization goals of popular VNE schemes.

(A) *Minimize Embedding cost*: One of the key metrics that has gained significant attention in the literature on VNE is the reduction of embedding costs, which refers to the cost of SN resources utilized during embedding a VNR. The overall objective function is captured in Equation 3.3, where c_{ik} and $c_{l,l'}^{i,i'}$ respectively capture the cost of embedding v_i on s_k and (i,i') on the physical link (l,l').

$$min \sum_{s_k \in \mathcal{N}^{\mathcal{S}}} \sum_{v_i \in \mathcal{N}^{\mathcal{V}}} c_{ik} \cdot x_{ik} + \sum_{(i,i') \in \mathcal{L}^{\mathcal{V}}} \sum_{(l,l') \in \mathcal{L}^{\mathcal{S}}} c_{l,l'}^{i,i'} \cdot x_{l,l'}^{i,i'} \tag{3.3}$$

(B) *Maximize Revenue-to-cost ratio:* Another key metric that dictates the performance of VNE schemes is the revenue-to-cost ratio, which captures the ratio of the number of resources requested to the actual amount of SN resources used to embed a VNR. Equation 3.4 captures the overall objective of maximizing the ratio, which assists in compact embedding, boosting the SPs revenue.

$$max \frac{\sum_{v_i \in \mathcal{N}^{\mathcal{V}}} d(v_i) + \sum_{(i,i') \in \mathcal{L}^{\mathcal{V}}} d((i,i'))}{\sum_{s_k \in \mathcal{N}^{\mathcal{S}}} \sum_{v_i \in \mathcal{N}^{\mathcal{V}}} c_{ik} \cdot x_{ik} + \sum_{(i,i') \in \mathcal{L}^{\mathcal{V}}} \sum_{(l,l') \in \mathcal{L}^{\mathcal{S}}} c_{l,l'}^{i,i'} \cdot x_{l,l'}^{i,i'}} \tag{3.4}$$

(C) *Maximize Acceptance Ratio:* It captures the number of VNRs successfully embedded (V^{emb}) to the number of VNRs in the system (V^{sys}) as depicted in Equation 3.5. A higher acceptance implies more profit for the SPs.

$$max \frac{V^{emb}}{V^{sys}} \tag{3.5}$$

Constraints on VNE Schemes: Any VNE scheme must adhere to the following constraints.

(A) *VM Embedding Constraint:* A VM v_i can be assigned to at most one server as reflected in Equation 3.6

$$\sum_{s_k \in \mathcal{N}^{\mathcal{S}}} x_{i,k} = 1 \quad , \forall v_i \in N^V \tag{3.6}$$

(B) *Server Capacity Constraint:* A server s_k cannot host VMs beyond its capacity captured by $c(s_k)$ as shown in Equation 3.7.

$$\sum_{v_i \in \mathcal{N}^{\mathcal{V}}} d(v_i) \cdot x_{ik} \leq c(s_k), \quad \forall s_k \in \mathcal{N}^{\mathcal{S}} \tag{3.7}$$

(C) *VL Embedding Constraint:* A VL $(i,i') \in \mathcal{L}^{\mathcal{V}}$ must be mapped to at least one physical link as per equation 3.8.

$$\sum_{(l,l') \in \mathcal{L}^{\mathcal{S}}} x_{l,l'}^{i,i'} \geq 1, \quad \forall (i,i') \in \mathcal{L}^{\mathcal{V}} \tag{3.8}$$

(D) *Link Capacity Constraint:* The number of VLs hosted by a physical link is bounded by its maximum capacity $c(l,l')$ as captured in Equation 3.9

$$\sum_{(i,i')\in\mathcal{L}^{\mathcal{V}}} x_{l,l'}^{i,i'} \cdot d(i,i') \leq c(l,l'), \quad (l,l') \in \mathcal{L}^{\mathcal{S}} \tag{3.9}$$

(E) *Flow Constraint:* Given the mapping of v_i and v_i' mapped onto servers s_k and s_k' the flow conservation constraint for the VL (i,i') is expressed in Equation 3.10. The first case captures the flow constraint at intermediate points, whereas the others capture the start and end points dictated by the hosting servers.

$$\begin{cases} \sum_{(l,l')\in\mathcal{L}^S} x_{l,l'}^{i,i'} \cdot d(i,i') = \sum_{(l,l'')\in\mathcal{L}^S} x_{l,l''}^{i,i'} \cdot d(i,i') \\ \sum_{(k,l)\in\mathcal{L}^S} x_{k,l}^{i,i'} \cdot d(i,i') = d(i,i') \\ \sum_{(l'',k')\in\mathcal{L}^S} x_{l'',k'}^{i,i'} \cdot d(i,i') = d(i,i') \end{cases} \tag{3.10}$$

3.1.3 NP-Hardness

VNE is well-established to be $\mathcal{NP}-Hard$ and shares similarities with the multi-way separator problem [2]. Even after the VMs are embedded, allocating VLs to substrate paths reduces the unsplittable flow problem, which is also proven to be $\mathcal{NP}-Hard$. Thus, achieving an optimal solution for VNE essentially involves tackling two computationally complex problems: VME (Definition 3) and VLE (Definition 4). Additionally, due to the intractable nature of these problems, obtaining an optimal solution for larger problem instances poses significant challenges. A problem instance of VNE is subsequently discussed.

3.2 A High-Level Illustration of a VNE Problem Instance

To illustrate the problem of VNE more effectively, we provide an example scenario depicted in Figures 3.2 and 3.3. Referring to Figure 3.2, we have a SN consisting of servers S_1, S_2, S_3, S_4. A VNR with two VMs v_1, v_2 needs to be embedded into the SN. The resource demands of the VMs and the servers' capacities are expressed in CRBs, which are placed adjacent to the respective node in a rectangular box. The capacities of the hosts, expressed in CRBs, are denoted as numerals in rectangular boxes adjacent to the servers in Figure 3.2. For example, the capacity of server S_1 and VM v_2 is 2 and 1 CRB, respectively. Alternatively, the link demands are assumed to be gigabits per second *(Gbps)*. For instance, the VL interconnecting v_1 and v_2 has a bandwidth demand of 2 *Gbps*. Similarly, the physical link between servers S_1 and S_2 has a capacity of 3 *Gbps*. In the provided setup, an embedding solution is depicted in Figure 3.3. The VMs v_1 and v_2 are mapped onto servers S_1 and S_4, respectively. The VL interconnecting the VMs is assigned onto the substrate path $S_1 - S_2 - S_4$. The updated capacities of the substrate resources after embedding are also indicated in Figure 3.3.

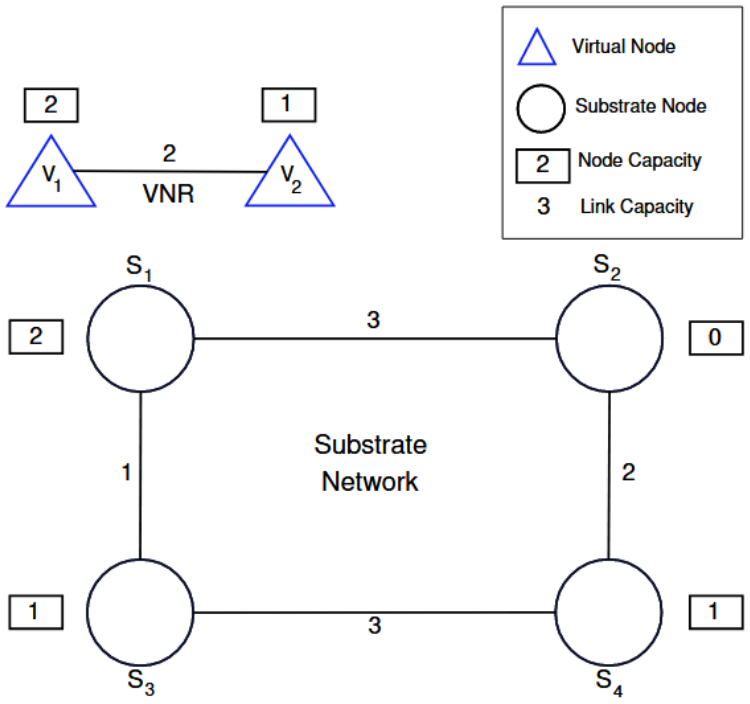


Figure 3.2. An embedding instance. It consists of (i.) a VN with two VMs and a VL with their resource demands and (ii.) a substrate network with four servers and four physical links and their available capacities.

3.3 Components and Modules

In any VNE system, as shown in Figure 4.1, there are generally two main components: (1) the VNR and (2) the SN. The control modules are crucial for any embedding solution and are broadly categorized into (1) the Resource Manager and (2) the Embedding & Monitoring module. The Resource Manager primarily logs the status of resources, tracking both used and unused resources within the network. Meanwhile, the Embedding & Monitoring module is responsible for identifying the optimal embedding on the SN and continuously monitoring the VNRs and SN resources. This module also collaborates with the Resource Manager to re-embed or reallocate resources in response to dynamic network changes, such as fluctuating VM or VL demands of a VNR or in cases of hardware or software failures in the SN. Furthermore, these two modules interact regularly to execute essential management tasks, such as resource consolidation during offpeak times, disaster recovery, and failure management.

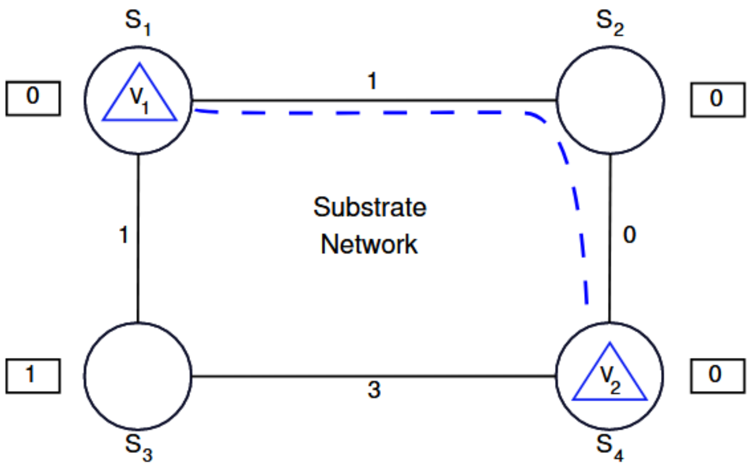


Figure 3.3. The state of the substrate network after VM and VL embedding of the VNR in Figure 3.2

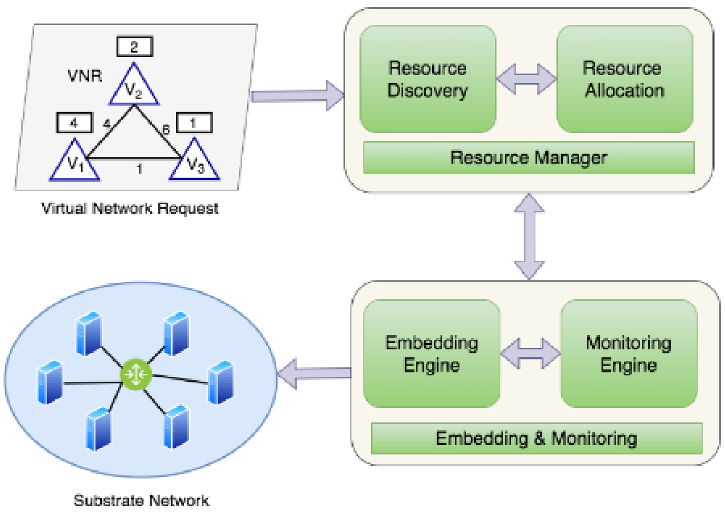


Figure 3.4. Signal Flow Diagram

3.4 Taxonomy for VNE

Depending upon the nature of deployment, the VNE strategies are either (1) *static* or (2) *dynamic*. The former infrastructure remains consistent, whereas the latter encompasses changing VNRs and SN. Moreover, the VNRs may be spread over multiple SNs owned by different SPs. In such cases, the VNE may be performed using resources of different SNs in a distributed manner instead of a centralized setup with a single SP. Considering the number of resources consumed to provision service requirements, the VNE can be classified as *precise*, where the VNRs are provisioned with minimum possible substrate resources, or *redundant*, where multiple substrate resources are provisioned together to realize a virtual component. On the other hand, depending on the stakeholders to which the embedding caters, the strategies can be broadly classified into *SP-centric* or *user-centric*. Based on the above-discussed classification, the overall literature on VNE can be categorized as *static* or *dynamic*, *centralized* or *distributed*, *precise* or *redundant*, *SP-centric* or *user-centric*.

Static or Dynamic (S/D) In a static setup, the embedding algorithm is exposed to a fixed set of VNRs executing over a SN for a pre-determined time with no change in the demanded resources. On the other hand, modern applications are dynamic because they arrive and leave the system dynamically. Additionally, during their stay in the system, the resource requirements of the components, i.e., VMs and VLs, fluctuate dynamically [37]. In theory, the static embedding policies can operate in a dynamic setup but are not designed to handle the relocations of VNRs in their entirety or parts. Relocation or rearrangement of the VNRs' can also be performed to handle the fragmentation of resources caused by the dynamic arrival and departure of VNRs. Over a while, the SN may be upgraded to deal with scalability, calling for the re-organization of VNRs.

Centralized or Distributed (C/D) A single central entity generates the embedding for centralized setups. Such an approach has a significant advantage over non-centralized approaches due to the availability of global knowledge about the state of the SN and other VNRs, significantly assisting the embedding protocol in making optimal decisions. However, like any other centralized setup, these approaches suffer from single-point failures and congestion owing to overwhelming requests during peak load. On the other hand, in a distributed setup, multiple stakeholders generate the embedding, which improves the scalability compared to centralized approaches but requires proper synchronization to achieve optimal results [3]. Moreover, the lack of global information in decision making mandates synchronization between decision-makers, resulting in an apparent trade-off between performance and communication overhead.

Precise or Redundant (P/R) Ideally, the VNR components, i.e., a VM and a VLs, are assigned to a substrate server and path having requisite resources. Such embedding policies are precise as they map the VNRs with the minimum possible substrate resources without foreseeing the possibility of failure. However, reserving redundant resources for failure-sensitive applications is critical to recovering from unforeseen failures [17]. An instance of such mapping is reserving multiple paths for a VL and splitting the requisite bandwidth over the selected paths. Unlike single-path mapping, the multipath reservation policy keeps the communication channel intact when faced with failures in any of the paths. An obvious consequence of reserving additional resources is the embedding cost, and the users are often required to strike a trade-off between cost and reliability.

SP-Centric or User-centric (S/U) Depending on the objective of the embedding strategies, the VNE schemes can be either SP-centric or user-centric. Note that if the VNE approaches focus on boosting the SP-specific metrics such as the embedding cost, revenue, revenue-to-cost ratio, and acceptance ratio, they are categorized as SP-centric. However, if the goal is to optimize parameters such as delay, throughput, and path length directly impacting the user service, it is referred to as user-centric.

3.5 Embedding Metrics

To assess the performance of different embedding strategies, we consider multiple embedding metrics that can broadly be classified as *quality-of-service (QoS) based*, *cost-based*, *resilience-based*, and *other metrics* [11]. A visual representation/classification of different embedding metrics in the literature is provided in Table 3.2. A brief insight into each category and type is subsequently presented.

Table 3.2. The relevant metrics used in the literature to evaluate different VNE schemes.

Embedding Metric	**Category**	**Description**
Path Length	QoS	Captures the number of substrate links reserved to provision a VL.
Stress Levels	QoS	Indicates the number of virtual entities assigned to a substrate entity.
Utilization	QoS	Ratio between aggregating the resources provisioned/spent due to embedding to the total capacity of a substrate entity (e.g., server or link).
Delay	QoS	Maximum tolerable delay for communication between any two communicating VMs mapped onto substrate servers.
Embedding Cost	Resource Spending	Total substrate resources expended in provisioning VNRs, including both VM provisioning (server resources) and VL provisioning (bandwidth).
Embedding Revenue	Resource Spending	Total amount of resources a VNR requests, considering both the VMs and VLs.
Cost/Revenue	Resource Spending	Ratio of resources provisioned to the demanded resource for a VNR.
Acceptance Ratio	Resource Spending	Ratio between the number of successfully embedded VNRs and the total number received.
Backups	Resilience and Robustness	Number of backups reserved for a virtual component (VMs or VLs).
Path Redundancy	Resilience and Robustness	Ratio between the number of backup and direct paths.
Node Redundancy	Resilience and Robustness	Number of running and backup nodes.
Network Criticality	Resilience and Robustness	Measures robustness against environmental changes, including traffic request variation, topology, and transmitter/receiver states.
Algorithm Runtime	Others	Local execution overhead from generating the embedding.
Communication Overheads	Others	Overhead from control message exchange between DCs or SPs, possibly leading to congestion and delayed processing.
Migration Overheads	Others	Overhead of relocating VMs and/or VLs, including consumed/released resources, disruption time, and relocation time.
Security Features	Others	Features added to improve embedding reliability.
Energy Concerns	Others	Energy consumed by nodes and/or links provisioned by the VNRs.

3.5.1 QoS-based metrics

These metrics measure service quality post-embedding of VNRs.

(1) *Path Length:* It reflects the number of substrate links reserved to provision a VL. A longer path length indicates a higher reservation of resources and escalated packet forwarding delays, thereby impacting the latency requirements of real-world applications such as video conferencing.

(2) *Stress Levels:* It indicates the number of virtual entities assigned to a substrate entity. The more entities mapped, the higher the stress levels. A higher stress level can negatively impact applications QoS. For instance, a substrate server loaded with multiple VMs keeps the operating system busy, resulting in degraded responses. On the other hand, a highly stressed physical link can result in congestion and packet transmission delay due to resource sharing.

(3) *Utilization:* It is the ratio between aggregating the resources provisioned/spent due to embedding to the total capacity of a substrate entity, i.e., a substrate server or a substrate link.

(4) *Delay:* Many real-world applications such as online gaming, video conferencing, and distributed computing often impose stringent restrictions on the maximum tolerable delay for communication between any two communicating VMs mapped onto the substrate servers. The VNE strategies must ensure that the delay requirements of the users are met.

3.5.2 Resource spending-based

The metrics discuss the number of resources expended in VNRs provisioning.

(1) *Embedding Cost:* Embedding cost refers to the aggregate amount of substrate resources expended in provisioning the VNRs, which includes the server resources for provisioning the VMs and the bandwidth resources for provisioning the VLs. The dominant factor is the VL embedding cost, as they are mapped onto substrate paths comprising multiple physical links. Other resources, such as storage, can also be considered while computing the embedding cost.

(2) *Embedding Revenue:* It is the total amount of resources a VNR requests considering both the VMs and VLs constituting it.

(3) *Cost/Revenue:* It is the ratio of resources provisioned to the demanded resource for a VNR. This metric helps compare different embedding protocols often tested over different topologies that make the embedding cost or revenue-based comparisons unfair.

(4) *Acceptance Ratio:* It captures the ratio between the number of VNRs successfully embedded and the number of VNR requests received.

3.5.3 Resilience and Robustness-based

These metrics focus on computing the resilience and reliability of an embedding.

(1) *Backups:* It captures the number of backups reserved for a virtual component, i.e., VMs and VLs. These backups can be used when the primary entity fails.

(2) *Path Redundancy:* It is the ratio between the number of backup and direct paths. These redundant paths can be used as backups in case of total/partial failures of the primary path.

(3) *Node redundancy:* It captures the number of running and backup nodes. In contrast to path redundancy, this metric captures the demanded resources only and not the connectivity.

(4) *Network criticality:* It is a measure of robustness that dictates the ability of a network to be stable irrespective of environmental changes such as (i.) changes in traffic request, (ii.) changes in topology and connections, and (iii.) changes in the state of transmitters and receivers. The detailed computation of network criticality can be found in [28].

3.5.4 Other metrics

These focus on different aspects of embedding, such as security, energy, migration, and communication overheads during embedding/re-embedding.

(1) *Algorithm runtime:* It refers to the total execution overhead generating the embedding. It is often a crucial metric for comparative studies of different embedding, especially for real-time applications.

(2) *Communication overheads:* For distributed setups, synchronization is essential and involves the exchange of messages between DCs or, in some cases, SPs. A higher degree of exchange can result in network congestion and delayed processing.

(3) *Migration Overheads:* For dynamic setup, there are scenarios where the VNRs have to be reorganized, implying the relocation of VMs and/or VLs. Such operations often incur overheads in terms of relocation, including additional resources consumed/released, service disruption time, and time expended in generating the relocation. Altogether, we categorize such costs as migration overheads.

(4) *Security Features:* To avoid malicious attacks on VMs and mapped VL paths, security features are often added to improve embedding reliability.

(5) *Energy Concerns:* As the VNR uses substrate resources such as CPU, memory, and bandwidth, energy is expended to use such resources. Therefore, this parameter encapsulates the energy consumed by the nodes and/or links provisioned by the VNRs.

3.6 Coordination between VME and VLE

VNE consists of two interdependent sub-problems: VME and VLE. Solving VME independently, without considering VLE, often results in suboptimal solutions due to a limited search space. Such methods, known as uncoordinated approaches, solve VME first, using its output as input for VLE. Recent uncoordinated solutions, as explored in [22], aimed to improve the revenue-to-cost ratio and resource utilization in substrate networks. Although promising, these approaches face several challenges: (1) restricted VLE search space leading to suboptimal resource utilization, (2) higher embedding costs due to inefficient VME, increasing bandwidth usage, (3) increased likelihood of embedding failures when VLE lacks sufficient bandwidth resources after VME, (4) poor network performance with higher latency and inefficient paths, (5) difficulty in meeting diverse QoS requirements without joint optimization, and (6) added computational overhead when reprocessing is required due to embedding failures.

In contrast, VME and VLE can be coordinated using one of two approaches. The first is a single-stage approach, where VME and VLE are solved simultaneously, mapping the corresponding VLs immediately after the VMs are mapped. As discussed in [10], this approach jointly optimizes both VME and VLE, avoiding the inefficiencies of uncoordinated methods where sub-optimal VM placement can limit the VLE search space. Considering VMs and VLs holistically reduces latency, improves path efficiency, and minimizes the risk of embedding failures, especially in applications requiring strict performance guarantees, like low latency or high bandwidth. The second method is a two-stage approach, as illustrated in [18], where VME is first performed with future VLE requirements in mind. In the second stage, VLE is solved based on VM placement, ensuring optimized link embedding. Although the one-stage approach offers tighter resource optimization, it can be computationally expensive. In contrast, the two-stage method provides better scalability and flexibility, making it more suitable for more extensive networks or scenarios where computational efficiency is a priority without sacrificing significant performance.

3.7 Exact and Approximate VNE strategies

The VNE problem has been demonstrated to be $\mathcal{NP} - Hard$ [2]. Similarly, the issues of VME and VLE are also computationally intractable. Consequently, as the size of the problem increases, the complexity of finding an optimal solution within an acceptable time frame rises exponentially. Given these challenges, researchers have mainly adopted two different solution approaches, namely (1) *exact*, and (2) *approximate*. This section aims to introduce the most important strategies used to solve the VNE problem, It also serves as an introduction that facilitates understanding how these formulations are later extended for the CNG problem.

Exact methods produce optimal solutions for small-scale problem instances. However, they prove to be *compute-intensive*, *time-consuming*, and *non-scalable* when confronted with more prominent test cases. Despite these drawbacks, exact solutions serve as benchmarks for evaluating the optimality bounds of approximate approaches. To address the limitations inherent in exact methods, some researchers have adopted *approximation* techniques, aiming to generate satisfactory solutions within reasonable time constraints.

3.7.1 Exact Solutions

Optimal VNE solutions can be achieved by means of Linear Programming (LP). More exactly, Integer Linear Programming (ILP) can be used to optimally formulate the VNE, including the virtual node and link mapping subproblems in the same formulation. there are exact algorithms (e.g., branch and bound, branch and cut, and branch and price) that solve small instances of the problem in a reasonable time.

3.7.2 Heuristic Solutions

Execution time is crucial in VNE. Network virtualization deals with dynamic online environments where VNRs arrival time is not known in advance. Therefore, to avoid delay in the embedding of a new VNR, the execution time of VNE algorithms should be minimized. Accordingly, heuristic-based VNE solutions are proposed. They attempt to find an acceptable solution, compromising optimality for short execution time. One good example of heuristic-based VNE approaches is the following: It proposes two main phases:

- (VNom) to map virtual nodes to substrate nodes based on CPU availability and connectivity.
- (VLim) Map virtual links onto the substrate paths using shortest paths that have sufficient bandwidth.

As an example, an algorithm begins by ranking substrate nodes based on their available CPU and node degree. Similarly, virtual nodes from the incoming VNR are ranked by CPU and degree as well. For each virtual node v , we select a substrate node s such that it satisfies the capacity constraint, and that s is not assigned to another v from the same request. After this stage, for each link $(u, v) \in \mathcal{L}^{\mathcal{V}}$, Dijkstra algorithm is used to find the shortest path going from $s_u = map(u)$ to $s_v = map(v)$ such that each link in that path satisfies the bandwidth constraint. If a valid path is found, we reserve bandwidth on each substrate link, otherwise, we backtrack previous node mappings and try alternative node mappings.

3.7.3 Metaheuristic and Evolutionary Solutions

While heuristics significantly reduce execution time, they may still become trapped in local optima when tackling large-scale or highly-constrained instances. *Metaheuristics* extend the search beyond local neighborhoods by applying stochastic operators inspired by natural or physical processes. Widely–used metaheuristic frameworks for VNE include:

- **Genetic Algorithms (GA)** [14]: chromosomes encode either a complete node–link mapping or separate mappings; crossover and mutation explore new embeddings while fitness combines resource utilisation and acceptance ratio.
- **Particle Swarm Optimization (PSO)** [15]: particles represent candidate embeddings and cooperate through velocity updates to converge toward high-quality regions of the search space.
- **Ant Colony Optimization (ACO)** [9]: artificial ants incrementally construct embeddings by probabilistically selecting substrate resources, with pheromone trails reinforcing successful mappings.
- **Simulated Annealing (SA)** [16]: a single solution is gradually improved by random moves that are accepted according to a temperature-controlled probability, enabling occasional uphill steps to escape local minima.
- **Tabu Search (TS)** [12]: short-term memory forbids recently visited embeddings, promoting exploration of unvisited areas.

3.7.4 Concluding Remarks

The Virtual Network Embedding problem encapsulates the trade-off between optimality and scalability. Exact formulations deliver gold-standard solutions for small instances, while heuristic, metaheuristic, and hybrid techniques unlock practicality for real-time, large-scale deployments. The methodological insights gathered here will underpin our discussion of resource orchestration in more imaginative digital realms. In the next chapter we move beyond classical telecommunication contexts and begin our exploration of the *musical metaverse*, where network virtualization meets immersive, interactive soundscapes.

Chapter 4

The Musical Metaverse

Throughout history, the way music is composed, performed, learned, and experienced has continually evolved alongside technological advancements. Today, the emergence of the **metaverse**, a persistent virtual world parallel to the physical world, offers a new frontier for musical activities. In general terms, the metaverse can be defined as a blend of the physical and digital worlds, enabled by the convergence of Internet of Things (IoT) and Extended Reality (XR) technologies. Users participate in the metaverse via avatars in immersive 3D environments, overcoming physical distance to socialize, work, play, and create together [1].

Within this broad vision, **music** is poised to play a prominent role: the **Musical Metaverse** refers to the sector of the metaverse dedicated to musical experiences and interactions. The MM is currently in its infancy, but it is rapidly evolving as researchers and practitioners explore how musicians and audiences can meet, perform, and co-create in shared virtual spaces. Recent work has begun to articulate a vision for the Musical Metaverse and identify the opportunities it opens for musical stakeholders, as well as the challenges that must be addressed to realize this vision [30].

4.1 From IoMusT and IoS to the Musical Metaverse

One way to understand the Musical Metaverse is as an extension of existing trends in networked music technology, in particular, the **Internet of Musical Things (IoMusT)** and the broader **Internet of Sounds (IoS)**. The IoMusT is an emerging field at the intersection of IoT and music technology that focuses on embedding computing and connectivity into musical objects and instruments. From a computer science perspective, IoMusT refers to networks of computing devices embedded in physical objects (called

[1]AI has been used as an assistant to generate the content of this chapter

Musical Things) that are dedicated to the production and/or reception of musical content. Examples of Musical Things include *smart musical instruments* (instruments augmented with sensors, processors, and wireless communication) and wearable devices for performers or listeners. These devices form local or remote networks enabling novel interactive music scenarios. Crucially, the IoMusT infrastructure supports **multidirectional communication** among Musical Things and human users, both co-located and geographically distributed. This allows performers, audiences, and other stakeholders to be connected in an ecosystem that supports musician-musician, audience-musician, and audience-audience interactions in real time [31]. In effect, IoMusT provides a technological backbone for connecting the physical and digital domains of music, opening the door to new musical applications and services.

The **Internet of Sounds (IoS)** generalizes this concept to encompass *all* networked devices dealing with sound and music. It can be seen as the union of two paradigms: the IoMusT (musical domain) and the Internet of *Audio* Things (non-musical sound domain) [32]. In other words, the IoS covers networks of devices capable of sensing, processing, and exchanging sound-related data, from musical instruments and smart speakers to acoustic sensor networks. Both IoMusT and IoS reflect a convergence of Sound and Music Computing with IoT, laying groundwork for immersive, interconnected sonic experiences. The **Musical Metaverse** builds on these foundations by situating IoMusT/IoS devices and data streams within *immersive virtual environments*. In the MM, Musical Things (smart instruments, wearables, etc.) become interfaces between real musicians and their virtual avatars or venues. XR technologies (VR/AR) provide the interactive, 3D context in which musical collaboration and audience participation can occur, while IoMusT provides the connected instruments and sensors that feed real-world musical actions into the virtual world [30]. In fact, IoMusT is expected to play a vital role in the network infrastructure of the Musical Metaverse, effectively bridging the real and virtual musical worlds. Through this bridge, data from performers actions and the physical environment can be captured and transmitted into the metaverse, and feedback can be sent back out to influence real-world devices (for example, actuators or haptic wearables).

Figure 4.1 represents a conceptual framework of the Musical Metaverse, illustrating the **physical layer** (real-world musical stakeholders and Musical Things), the **link layer** (networking and integration infrastructure), and the **virtual layer** (shared digital music environments and content). The physical layer includes real musicians (performers, students, audiences) equipped with IoMusT devices (e.g. smart instruments, wearables, XR headsets) that capture musical actions and environmental data. The link layer acts as a bidirectional bridge: it transmits sensor data from the physical layer into the virtual layer and returns feedback (e.g. audio streams, control signals) from the virtual layer back to physical devices. This layer comprises the communication network and services (such as data processing and storage) needed to synchronize and integrate the real-time musical interactions. Finally, the virtual layer hosts the metaverse's digital musical world - 3D venues, virtual instruments and objects, avatars, and interactive musical content - where users (as avatars) engage in musical activities within an immersive, interactive, and social experience.

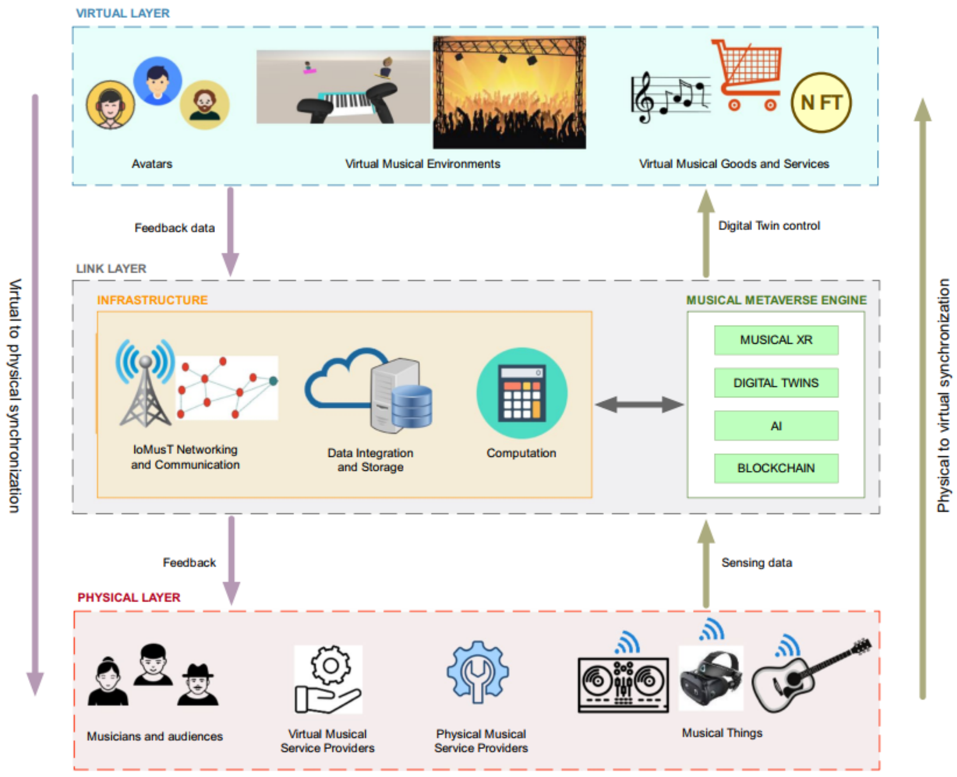

VIRTUAL LAYER
Avatars
Virtual Musical Environments
NFT
Virtual Musical Goods and Services
Feedback data
Digital Twin control
LINK LAYER
INFRASTRUCTURE
IoMusT Networking and Communication
Data Integration and Storage
Computation
MUSICAL METAVERSE ENGINE
MUSICAL XR
DIGITAL TWINS
AI
BLOCKCHAIN
Feedback
Sensing data
PHYSICAL LAYER
Musicians and audiences
Virtual Musical Service Providers
Physical Musical Service Providers
Musical Things
Virtual to physical synchronization
Physical to virtual synchronization

Figure 4.1. Framework of the Musical Metaverse, which consists of a physical layer, a link layer, and a virtual layer [30]

Equipped with IoMusT devices and connected through the link layer, users can conduct a variety of musical activities in the virtual realm of the MM. Envisioned scenarios range from intimate co-located performances enhanced by augmented reality, to **massive online music events** gathering thousands of participants in a shared virtual concert space. For example, researchers have proposed use cases including *augmented live concerts* (overlaying live performances with interactive AR/VR content), *audience participation systems* (where audience members inputs influence the music in real time), *remote rehearsals and jamming* across long distances, *networked music education* (virtual music lessons and practice in XR), and even *cloud-backed studio production environments* where distributed performers and producers collaborate in virtual studios. These scenarios illustrate how the Musical Metaverse could transform musical experiences, making them more accessible, inclusive, and enriched by digital content. A music student in Milan could play a virtual duet with a teacher in New York in a simulated concert hall, or thousands of fans could attend a virtual music festival from their homes, each represented by avatars and sharing an interactive audio-visual experience. The MM vision also encompasses new economic and creative opportunities: musicians might perform as virtual avatars or create interactive music NFTs, and content creators could build virtual instruments or stages to be traded and used in metaverse platforms.

While high-profile **virtual concerts** and music events (e.g. in gaming platforms or VR applications) have demonstrated the audience appetite for such experiences, the current metaverse technology stack is **"not ready for musicians"** to fully embrace interactive music-making in shared virtual worlds [5]. In other words, one-off events with a single performer streaming to a virtual audience (as seen in Fortnite or other platforms) have been feasible, but *synchronous co-creation* where multiple musicians actually **play together in real time** within a virtual environment remains extremely challenging with today's technology. Key limiting factors include technical constraints in networking and audio processing (especially latency), as well as the lack of standards and interoperability for music-related data and audio formats across different platforms. Researchers have only begun to investigate real-time networked musical interactions in XR settings, and early studies are few in number [30]. This gap between the vision and the current state-of-the-art highlights the need for focused development. To truly enable a Musical Metaverse that is *interactive, immersive, and widely accessible*, significant advancements are required in the underlying technologies. The following sections will delve into these foundations: first examining the core technologies that make a Musical Metaverse possible from immersive audio to smart instruments and XR interfaces (as this part is out of scope of this research we will provide only a brief overview, readers to get to know more, are invited to check references cited in this chapter). And then addressing the **networking infrastructure** and **design** required to support seamless, low-latency musical collaboration in the metaverse.

4.2 Technological Foundations of the Musical Metaverse

The Musical Metaverse (MM) rests on four tightly-coupled technology pillars: **immersive audio**, **musical extended-reality (XR)**, **smart musical things**, and a **cloud/edge backbone**. Together they allow physically distributed users to feel, hear, and act as if they were co-present in the same musical space.

4.2.1 Immersive Audio

Spatial-audio techniques: binaural rendering, ambisonics, and multichannel loudspeaker layouts recreate a believable acoustic scene so that every virtual instrument or audience reaction has a clear direction and distance. Rendering must remain **real-time and head-tracked**, yet keep end-to-end delay under ≈ 30 ms to preserve musical timing [21]. Remaining challenges are:

- **Standardisation**: proprietary engines and HRTF [21] libraries prevent content portability across XR and metaverse platforms.
- **Latency vs. fidelity**: uncompressed multichannel audio and complex room modelling increase bandwidth and computation. Lightweight codecs and GPU/ASIC-accelerated spatializers are active research areas.

4.2.2 Musical Extended Reality (XR)

VR, AR and MR supply the visual and **gestural layer** of the MM. Head-mounted displays place performers and listeners inside volumetric concert halls or studios; AR headsets can overlay scores or virtual band-mates onto a real room. Recent prototypes cover composition workspaces, educational tools, live VR concerts and virtual-studio mixing.

The main limitation is **networked multi-user scale**: only a handful of studies demonstrate geographically-separated musicians rehearsing in XR, because audio/graphics synchronization must share the same ultra-low-latency budget. Consequently, XR headsets and controllers are treated as Musical Things that stream sensor data alongside audio and therefore inherit the MM's strict timing constraints.

4.2.3 Smart Musical Instruments & Wearables

Smart instruments integrate sensors, embedded DSP and wireless links; wearables add motion capture or haptic feedback. Together they:

- convert **analog gestures into digital streams** (position, pressure, bow speed, etc.)
- transmit **high-quality audio** or control data
- receive commands, firmware or effect parameters in return.

4.2.4 Cloud- and Edge-based Music Services

Because performers, audiences and devices are geographically dispersed, MM workloads are off-loaded to a **cloud/edge continuum**, as described Table 4.1.

A representative architecture places **local 5G base-stations plus edge servers** in each city where musicians reside; those edge nodes exchange only the mixed or down-sampled audio, shrinking backbone traffic and equalising latency paths.

4.2.5 Interdependence

These four layers are **inter-locking**. For example, a guitarist's smart instrument captures gesture and audio, an edge mixer merges it with remote performers, the cloud's spatialiser renders a 6-DoF soundfield, and a VR headset displays synchronised visuals, all within the same 30 ms deadline. Any bottleneck (Wi-Fi congestion, HRTF mismatch, delayed cloud uplink) immediately breaks the illusion of co-presence. Understanding these dependencies is essential before tackling the next chapter's networking design, where virtual network embedding and QoS guarantees turn conceptual building blocks into a reliable, real-time musical infrastructure.

Service	Role in MM	Latency considerations
Real-time mixing & spatialisation	Combines N audio streams into personalised binaural output	Deployed on Multi-access Edge Computing (MEC) nodes close to users (around 10 ms round-trip)
Global clock / state	Distributes metronome and synchronises avatars & effects	Requires sub-millisecond clock alignment across nodes
Content delivery (3-D models, samples)	Fetches virtual instruments & stage assets on demand	Less time-critical but bandwidth-hungry
AI/ML services	Transcription, tutoring feedback, avatar animation	Off-loaded to GPU clusters; non-blocking if pipelined
Elastic scaling	Spins up extra mixers or physics engines for large events	Must not degrade active performers' latency

Table 4.1. Selection of services and their roles in MM scenarios

4.3 Networking Infrastructure for the Musical Metaverse

The most critical enabler for the Musical Metaverse is the **networking infrastructure** that connects all participants, devices, and services with the required performance. Musical interactions are highly sensitive to delay and disruptions, much more so than typical internet applications. This chapter focuses on the networking aspects that must be engineered to support the MM: **ultra-low latency communication, Quality of Service (QoS) and Quality of Experience (QoE) guarantees, network topology and resource design, synchronization mechanisms**, and support for **multi-user real-time interaction**. We discuss current approaches and challenges, linking them to the virtual network embedding problem of how to allocate and orchestrate network resources for these demanding scenarios.

4.3.1 Ultra-Low Latency Communication

Latency is the time it takes for data (such as an audio packet or a sensor message) to travel through the network, is arguably the single most important network metric for the Musical Metaverse. In a collaborative music performance, latency directly translates to **delay between a musician's action and another musician's perception of it**. If this delay is too large, it becomes impossible to stay in sync rhythmically. Decades of research in **Networked Music Performance (NMP)** have shown that performers

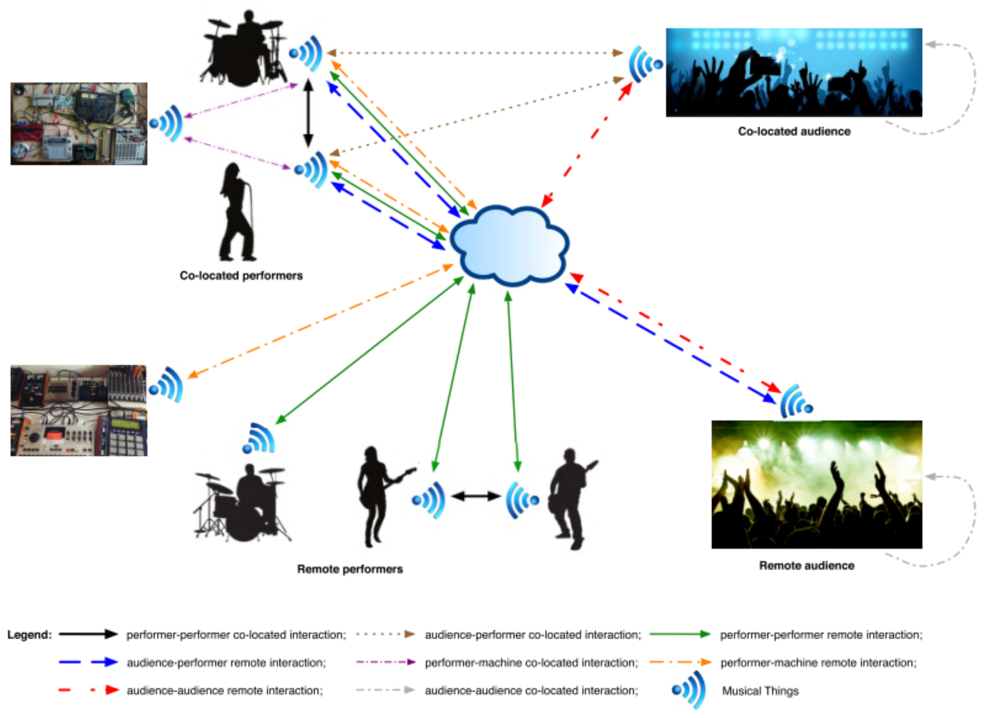

Co-located audience
Co-located performers
Remote performers
Remote audience
Legend:
performer-performer co-located interaction;
audience-performer co-located interaction;
performer-performer remote interaction;
audience-performer remote interaction;
performer-machine co-located interaction;
performer-machine remote interaction;
audience-audience remote interaction;
audience-audience co-located interaction;
Musical Things

Figure 4.2. Interaction possibilities between musicians, audience members, and machines. [31]

start to feel the effect of delay on musical timing at very low thresholds: generally, one-way latencies above roughly 25–30 milliseconds make it difficult to play tightly together, especially for fast tempo music. This corresponds to about a 50–60 ms round-trip delay (musician A's sound to B and back to A), beyond which the musical interaction degrades, notes might come in late, rhythmical grooves fall apart, or the performers might unconsciously slow down to compensate. Hence, **ultra-low latency (ULL)** networking (targeting tens of milliseconds or less) is a foundational requirement for the MM. For reference, typical latencies on today's Internet (hundreds of milliseconds) are far too high, and even a video call's latency (around 100 ms) would be inadequate for real-time jamming. The MM requires networking performance closer to that of local-area networks or specialized audio links.

Another technique to reduce *perceived latency* is to handle as much processing in parallel as possible. For instance, audio capture, encoding, network transmission, decoding, and playback should be pipelined efficiently. Specialized protocols can be used instead of generic ones, many NMP systems avoid the overhead of TCP and instead use UDP with custom loss recovery, to cut down latency. Some even send uncompressed audio to skip encode/decode delay (at the expense of bandwidth). Every millisecond counts, so engineers consider everything from kernel-level audio drivers to avoiding congested routers on the path. The **metaverse infrastructure may employ dedicated network routes or slices** for music traffic to bypass Internet congestion and hops, ensuring a faster, more predictable path (this is related to virtual network embedding, where a virtual dedicated circuit is mapped onto physical links with reserved capacity).

4.3.2 QoS, Reliability, and QoE Considerations

Beyond just latency, the **Quality of Service (QoS)** provided by the network encompasses bandwidth (throughput), reliability (packet loss/error rates), and jitter (latency variation). Musical applications can be quite demanding in these aspects as well. High-fidelity audio streams (especially multi-channel or many streams at once) require significant bandwidth; likewise, video or XR data streams in a musical performance (for visuals of avatars or gesture data) add to the load. The network must support these data rates without introducing congestion that leads to packet loss. Even a small packet loss can mean an audible glitch or drop-out in an audio stream, which is very disruptive during a performance. Thus, reliability mechanisms or over-provisioning are needed to ensure near-zero loss. Some systems use forward error correction (FEC) or redundant audio packet streams to mask losses, but these add overhead; ideally, the network should be reliable enough not to drop packets in the first place.

Quality of Experience (QoE) is the user-centric counterpart to technical QoS metrics. It measures the actual satisfaction or perceived quality by the musicians and listeners. In a musical context, QoE involves subjective elements like *does the performance feel "in sync" and together?, is the audio clear and natural?, is there any distracting artifact?*, it is possible to meet QoS targets on paper but still have poor QoE if, for instance, the latency, while low, is just high enough to create subtle timing issues that bother trained musicians, or if jitter causes slight rhythm fluctuations. Thus, system designers often aim for stricter QoS than the bare minimum to provide a comfortable margin for QoE. Studies have indicated that musical experts might detect and be annoyed by even smaller latencies under certain conditions (for example, in highly rhythmic tasks or with certain instruments), meaning the network should ideally keep delays as low as possible, not merely below a threshold.

The **link layer** infrastructure of the MM is envisioned with these stringent requirements in mind. It must support synchronous musical interactions with strict latency and reliability, while carrying potentially large volumes of data (audio, video, haptic feedback, etc.). Potential solutions recommended by recent research include the use of **5G and beyond-5G networks**, and deployment of ultra-dense networks (to shorten wireless distances and increase capacity). These approaches all aim to support QoS by reducing physical distance, increasing bandwidth, and localizing traffic. Additionally, the network should be adaptable: if a sudden surge in traffic happens (say a new musician joins with a high-bandwidth instrument feed), or if a route degrades, the infrastructure should quickly adjust (reroute, allocate more resources, etc.) to maintain the quality.

4.3.3 Synchronization and Multi-User Interaction

Achieving tight **synchronization** in a Musical Metaverse session is a multifaceted challenge. It's not just the network transport latency that matters, but also aligning the musical timing among all participants and media. There are a few synchronization aspects to consider:

- **Audio-Visual Sync**: Within each user's experience, the audio should sync with visual cues. If a drummer's avatar is seen hitting a drum, the sound should coincide with that action. This implies that the pipeline for audio and the pipeline for graphics (and possibly haptic feedback) must be coordinated. Any differences in their latency need compensation, for instance, if audio is faster than video, one **might delay audio slightly to match**. Metaverse platforms often have internal clocks to tag media timestamps, so they render frames and play audio in a synchronized fashion for each user.

- **Inter-user Musical Sync**: This is the core of playing music together. Even if the network has low latency, there will always be some delay between one player and another. To maintain a stable tempo, groups often implicitly follow a single timing leader or a metronome. In a networked setting, a common technique is to designate a master clock (could be one participant or a server) and have everyone synchronize to that. Protocols like RTP with timestamping, or specialized sync protocols (similar to NTP but for media) can align clocks with sub-millisecond precision over networks.

- **Event Synchronization**: In virtual worlds, beyond continuous media, there might be musical *events* (like starting a song, triggering a visual effect on a downbeat) that need to happen simultaneously for everyone. This requires distributing a trigger signal that all clients execute at the same agreed time. Techniques from online gaming (lockstep protocols, or using the network delay to schedule a future simultaneous moment) are applicable. For example, "we will all start playing at time T according to the shared clock", each client waits until their clock hits T, thus compensating for any difference in when they got the message.

For **multi-user interaction**, scaling to larger numbers of active participants introduces further complexity. With two players, the main concern is their mutual latency. With, say, five players, latency differences can form a complex matrix. you might have to ensure that the slowest link doesn't ruin it for everyone.

One exciting possibility in the MM is having *audience participation* where many audience members become quasi-performers (for instance, everyone can trigger a sound or sing along). This starts to look like a massively multi-user scenario. Handling that might require clever architectural solutions like local grouping (e.g., if 1000 audience members sing, perhaps their inputs are aggregated into a "crowd audio" mix per region and then those region mixes are combined, to avoid sending 1000 individual streams). These kinds of hierarchical mixing and network aggregation strategies will be important as we push into larger metaverse events.

In conclusion, synchronization and multi-user interaction in the Musical Metaverse demand meticulous coordination. It's an area where human factors (perception of timing) meet network engineering. By using global time references, careful buffering, and adaptive strategies, and by confining all users to a well-managed virtual network slice, the MM systems strive to give the illusion that everyone is playing *in the same room at the same time*. Achieving this illusion is precisely the challenge that motivates advanced networking solutions: including virtual network embedding, edge computing, and custom protocols as discussed throughout this chapter.

Chapter 5

Musical Metaverse Optimization (MusMOPT)

5.1 Limitations of classical VNE for Musical Metaverse requirements

The **Musical Metaverse**, places extreme demands on networking and computing that expose fundamental limitations of the classical Virtual Network Embedding (VNE) model. In practice, ensuring end-to-end latencies on the order of only a few milliseconds with near-zero jitter and packet loss is necessary to preserve the illusion of co-presence and maintain musical timing. Moreover, musical metaverse scenarios involve **multi-modal, interactive media flows** beyond just audio. Musicians and audiences exchange not only high-fidelity audio streams but also real-time control signals, sensor data (e.g. gestures or motion capture), and even shared virtual environment data. Delivering and synchronizing these **multimodal streams** (audio, video, haptic feedback, etc.) introduces additional challenges in maintaining quality of service (QoS) and quality of experience (QoE).

Classical VNE, however, was not designed with these demands in mind. In the traditional VNE model, a service is **abstracted** as a *virtual network*: a set of virtual nodes (virtual machines or functions) connected by virtual links with fixed capacities or bandwidth demands. The goal is to map (embed) this virtual network onto a physical *substrate network* (infrastructure of phyiscal nodes and links) while respecting resource capacities. **VNE assumes a one-to-one mapping:** each virtual node is placed on a single substrate node, and each virtual link is assigned a specific path through substrate links. Crucially, *classical VNE treats each virtual link as an independent point-to-point traffic demand, without awareness of the content or relationships between flows.*

This assumption becomes a critical **drawback** in the musical metaverse. For example, if one musician's audio stream needs to be delivered to multiple other participants,

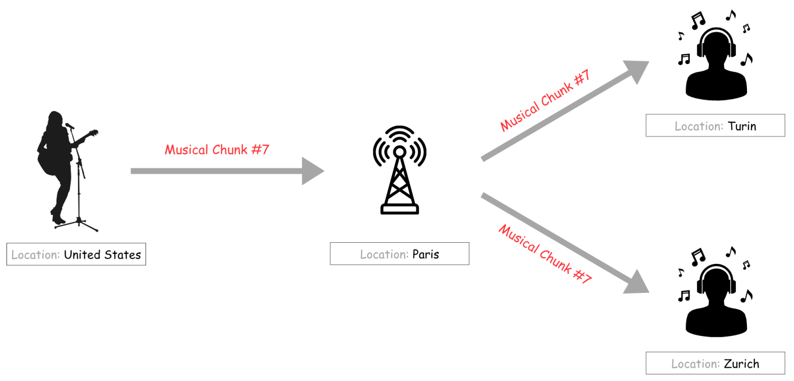


Figure 5.1. Illustrative example of a musical chunk data stream sent by a musician to two different consumers (listeners)

the VNE model would represent this as multiple separate virtual links carrying identical audio data. The VNE embedding algorithms would attempt to route each link separately, potentially duplicating data over the network and consuming bandwidth on each path redundantly. Figure 5.1 shows a simple example. Imagine the data being sent (here denoted as "Musical Chunk #7") must be delivered to two different destinations. Here, following the definition of classical VNE, we need to create two different replicas to represent a data stream originating in the United States and going to Paris. However, this creates an additional cost because we are simply replicating this "link" for each consumer.

Definition 6. *There is* ***no mechanism in basic VNE*** *to realize that these flows carry the same information and could be served by a single multicast distribution.*

Similarly, if a virtual audio processing function (e.g. a mixer or effect) needs to be applied for multiple users, VNE would typically allocate one instance of that function at a single location, forcing all relevant flows through that point. It cannot natively model **replicating** that function across edge servers to process streams closer to each user.

In summary, the classical VNE model fails to meet the musical metaverse needs in several aspects:

- **Latency-aware placement:** VNE embeddings often optimize resource usage or cost, but without built-in guarantees of strict latency bounds. Ensuring that, say, an audio path stays below 10 ms end-to-end would require manually adding latency constraints, which makes an already NP-hard problem even more complex. There is no straightforward way in classical VNE to integrate distributed edge computing decisions purely to minimize delay e.g. placing audio processing on a nearby edge server to reduce round-trip time, unless explicitly formulated.

- **Lack of flow-level intelligence:** VNE sees traffic demands as abstract quantities between virtual nodes. It has **no awareness of the** ***actual information flows*** (e.g. which audio stream is being sent) and thus cannot exploit commonalities between flows.

- **Static and one-to-one mappings:** Musical experiences are dynamic and often many-to-many (many performers to many listeners). Classical VNE's one-to-one mapping and point-to-point links do not naturally support *multicast* or *anycast* delivery, nor the concept of **multiple simultaneous placements** of a function. If a virtual service requires duplicating a stream to multiple endpoints or instantiating an audio effect in multiple locations, a VNE-based approach has to approximate this by embedding multiple separate virtual nodes/links, losing efficiency and increasing management complexity.

- **Limited resource heterogeneity:** NE typically considers two resource types – substrate node CPU (or processing capacity) and substrate link bandwidth. Musical metaverse applications require a broader view of resources: not only compute and network, but potentially storage (for caching audio assets or recorded media), and they operate across a **device-edge-cloud continuum**. These additional resource dimensions and the need for **modalities-aware routing** (choosing paths appropriate for audio vs. less time-sensitive data) are beyond the scope of traditional VNE formulations.

All these limitations indicate that *classical VNE*, on its own, is insufficient to orchestrate real-time, immersive musical services. The musical metaverse calls for a paradigm that natively understands **information flows, multicast distribution, and integrated compute-network decisions** under strict QoS constraints. This motivates a transition to a new model: **MusMOPT** which we discuss next.

5.2 MusMOPT Model

To address the shortcomings of VNE, we introduce **Musical Metaverse Optimization** as a graph-based optimization framework that reconceptualizes how services are mapped onto infrastructure. In essence, MusMOPT treats the joint placement of functions, routing of data, and allocation of resources as **a single unified information flow problem** on an augmented network graph. Rather than embedding a "virtual network" of nodes and links, MusMOPT considers the *service* as a *directed graph* of information processing (often modeled as a Directed Acyclic Graph, DAG). And the physical infrastructure is a rich **cloud–network graph**.

This cloud-network substrate graph includes not only physical links and nodes but also represents available **compute and storage resources** at those nodes. Each substrate node (e.g. user device, edge server, cloud data center) has capacities for hosting service functions (processing) and possibly storing content, and each link has communication capacity and propagation delay.

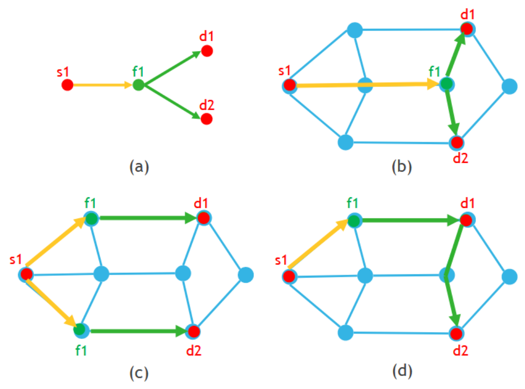


Figure 5.2. Illustration of the lack of isomorphism between an information-aware service graph and its instantiation into the physical infrastructure: (a) depicts an information-aware service graph with colors indicating the information carried by each data stream; (b), (c), and (d) illustrate three different possible information-aware embedding into an 8-node network.

On the service side, MusMOPT's virtual service model is an **information-aware service graph** that captures the structure of the application's data flows and processing requirements. For a musical application, this could be a DAG where vertices represent processing tasks or *functions* (e.g. audio analysis, mixing, spatial rendering, sensor processing) and *edges* represent **data streams** (audio signals, control messages, etc.) flowing between those functions or to end-users. Importantly, this model carries information attributes for each stream: for example, an audio stream might be one information object with a certain bitrate and latency requirement, a video or sensor stream another object, etc. MusMOPT introduces the notion of **commodities** to represent these distinct information flows (See Figure 5.2). Each *information commodity* (say, a particular musician's audio stream) can have multiple destinations (all other performers and audience members who should receive that audio). The MusMOPT formulation then allows these commodity flows to be *routed, replicated, and processed* through the substrate in an optimal way.

Key characteristics of the MusMOPT model include:

- **Service DAG with splitting and merging:** The virtual service is a general DAG, not restricted to a chain or single tree. This allows a function to have multiple outputs (branching) and multiple inputs (merging), capturing complex interactions (e.g. an audio mixing node takes inputs from several instruments). MusMOPT enforces *service chaining constraints* so that each flow goes through the required sequence of functions in the DAG (respecting input-output dependencies).

- **Flow replication (multicast support):** MusMOPT inherently handles multicast distribution. If a data stream (commodity) is needed in multiple places, the model can create *replication flows* that duplicate the packets toward different destinations. Multiple receivers can thus share parts of the path. The formulation allows overlapping flows: if two virtual flows carry the same information object, they can traverse the same substrate link without consuming separate capacity for each copy. In other words, the network only carries one copy of a stream up until the point it needs to split toward different receivers (akin to a multicast tree). This is a fundamental departure from classical VNE, which lacked awareness of shared information and could not overlap demands in this way.

- **Flexible function placement and replication:** Rather than fixing one location per virtual function, MusMOPT can deploy **multiple instances of a function** across the network. For example, if a spatial audio rendering function is needed for audiences on different continents, MusMOPT might spawn that function at two edge servers (one per region) and send each audience member's stream to the nearest server, reducing latency. The model decides *the number of replicas and the location of each service function* as part of the optimization.

- **Flow splitting and multi-path routing:** MusMOPT supports both *unsplittable* flows (a commodity's data must follow one route) and *splittable* flows (traffic can be split across multiple paths). In the majority of musical scenarios, splitting an audio stream over parallel links might not be meaningful (due to needing in-order audio), so flows are be treated as **unsplittable**.

- **Flow scaling (volume changes):** As data passes through processing functions, its rate can change. For instance, raw audio might be compressed (shrinking data rate) or a sensor stream might trigger generation of a larger media stream. MusMOPT explicitly accounts for this by linking the rates of output flows to input flows via scaling factors

Definition 7. *By casting the problem in this flow-based way, MusMOPT essentially generalizes VNE. In fact, if we restrict MusMOPT to a scenario with **unsplittable unicast flows (no multicast)** and disallow function replication, it reduces to the classical VNE problem.*

5.3 Cloud Network Graph

The MusMOPT discussed earlier was originally introduced in [7] as CNFlow. However, the framework was designed for 3C networks (Connected Collaborative Computing) and lacks a clear and generalized graph formulation suitable for metaverse scenarios. In this section, we are going to explain our first contribution, represented by the **Cloud Network Graph (CNG)**. Unlike VNE, where the substrate network is modeled as an undirected graph that consists of a set of nodes and links. CNG is an directed augmented cloud network graph that represents the network infrastructure as a set of commodities (information flows), allowing for a seamless design for MM scenarios, and enabling studying complex connections between components for data-intensive applications.

The cloud network graph (CNG) is a directed *augmented* graph denoted as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Where $\mathcal{V}$ represent cloud-network nodes (e.g. core cloud nodes, edge cloud nodes, compute-enabled base stations, or end devices with embedded computing resources), and edges, $e \in \mathcal{E}$, represent network links between computing locations. Each node $u \in \mathcal{V}$, is further augmented as illustrated in Figure 5.3 where:

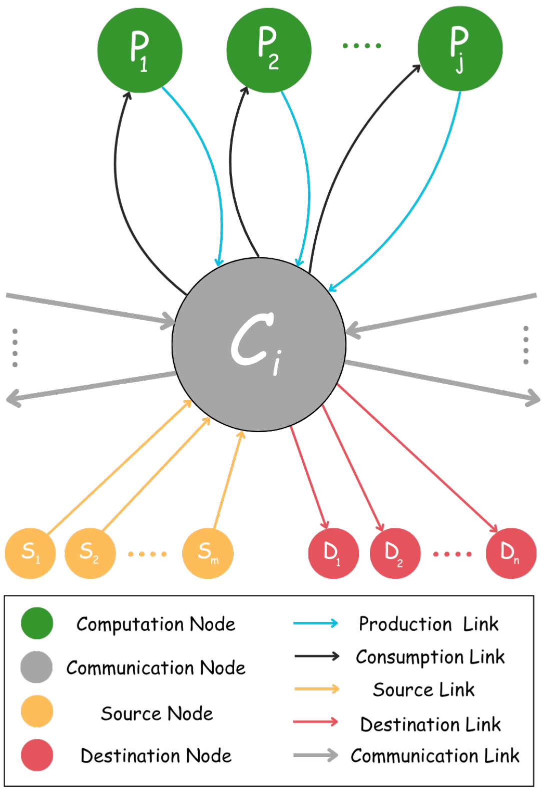


Figure 5.3. Augmented node G_{C_i}. Grey edges are inter-site communication links. Coloured edges: source, production, consumption, destination.

- $s \in \{S_1, S_2, ..., S_m\}$, $d \in \{D_1, D_2, ..., D_n\}$, and $p \in \{P_1, P_2, ..., P_j\}$, and associated links are used to model *production*, *consumption*, and *processing* of data streams respectively.
- The resulting network graph is denoted by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{V}^c \cup \mathcal{V}^p \cup \mathcal{V}^s \cup \mathcal{V}^d$, and $\mathcal{E} = \mathcal{E}^c \cup \mathcal{E}^p \cup \mathcal{E}^s \cup \mathcal{E}^d$ denoting the set of *communication*, *computation*, *source*, and *destination* nodes, and links respectively.
- In $\mathcal{G}$, each link $(i, j) \in \mathcal{E}$ is characterized by its capacity c_{ij} and cost w_{ij} parameters. In particular:

  - Each communication link $(i, j) \in \mathcal{E}^c$, c_{ij} and w_{ij} denote the capacity in communication flow units (e.g., bits per second or bps) and the cost per unit flow at link (i, j), respectively. Additionally, we denote by t_{ij} the propagation delay (in ms) for a communication link.
  - Analogously, for each computation link $(i, j) \in \mathcal{E}^p$, c_{ij} and w_{ij} denote the capacity in computation flow units (e.g., FLOPS) and the cost per unit flow at link $(i, j) \in \mathcal{E}^p$, respectively. The propagation delay between a communication node and its computation nodes is assumed to be negligible.

- We further expand the set $\mathcal{E}^p$. We use the set of *computation out* links $\mathcal{E}^{p_{out}} \subset \mathcal{E}^p$ with original at a computation node $p \in \mathcal{V}^p$ and target at a communication node $u \in \mathcal{V}^c$, to represent the processing resources (e.g., CPU) available at that computation node/cluster.

- Similarly, we define the set of *computation in* links, $\mathcal{E}^{p_{in}} \subset \mathcal{E}^p$ with original at a communication node $u \in \mathcal{V}^c$ and target at a computation node $p \in \mathcal{V}^p$ to represent the memory resources (e.g., RAM) available at that computation node/cluster.

- Source and destination links $\mathcal{E}^s, \mathcal{E}^d$ are assumed to have zero cost and high enough capacity, acting as network ingress and egress points, respectively.

- Finally, we denote by $\mathcal{N}^-(u)$ and $\mathcal{N}^+(u)$ the set of incoming and outgoing neighbors of node $u \in \mathcal{V}^c$, respectively.

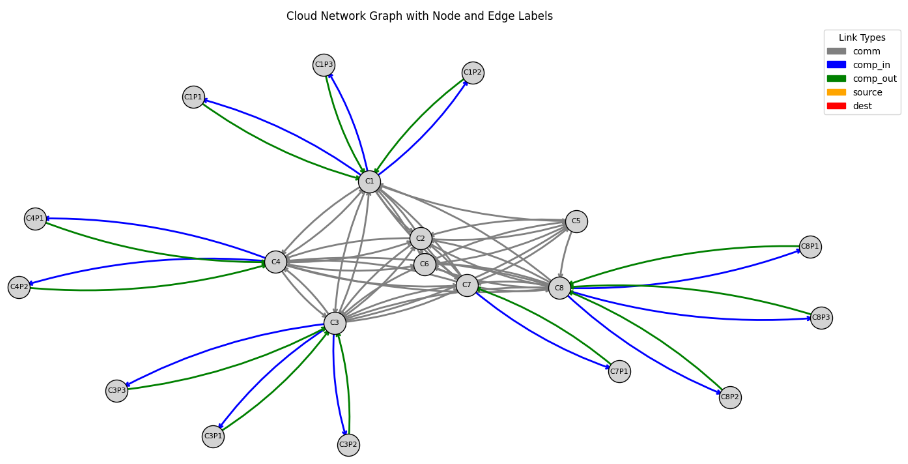


Figure 5.4. A cloud-augmented network graph of 8 communication nodes

In Figure 5.4 we present a sample example of a CNG, as shown in the illustration, it is not necessary for a communication node to have computation nodes (acts just as forwarding stubs). On the other hand, the number of computation nodes is independent for each communication node.

As a conclusion, in this chapter, we motivated and formalized a shift from classical **virtual network embedding** to the **cloud network flow** paradigm in order to satisfy the extreme real-time requirements of Musical Metaverse applications. We introduced a novel, concrete directed, augmented graph tailored to metaverse scenarios. A physical node is expanded into communication, processing-out, processing-in, source, and destination interfaces, allowing unified treatment of bandwidth, CPU, and memory. Link capacities and costs are assigned per modality, enabling latency-aware, modality-aware routing across the device–edge–cloud continuum.

In the next chapter, we will discuss our **second** contribution: A service graph for one of the musical metaverse scenarios, and discuss both its *graph layer* and *data layer*.

Chapter 6

Service-Graph Foundations

In chapter 4, we discussed the application space and highlighted the network stressors introduced by immersive musical scenarios, while in Chapter 5, we introduced the cloud network graph (CNG) as a principal graph infrastructure to orchestrate scenarios at scale. We now pivot from the general framework to a concrete use case: **audience interaction in pre-recorded virtual concerts**. Unlike real-time networked performances, where the performers themselves generate live media, pre-recorded concerts start from static, studio-quality content. The novelty lies in enabling large, geographically-dispersed audiences to interact with that content, and with one another, as though attending a traditional live concert. The result is an experience that merges the production values of studio recordings with the social energy of a live venue.

This chapter has two goals:

- Introduce, and motivate the design, a **novel** service graph for the above-mentioned scenarios
- Formulate the "MM problem" as a two-layer problem: Data layer, responsible for handling real-time traffic data, and a graph layer, that focuses on the embedding of services over the network infrastructure.

6.1 Interaction Model for Pre-Recorded Concerts

A pre-recorded concert in the Musical Metaverse typically begins with multitrack studio stems (audio), synchronised high-resolution video, lighting cues, and stage metadata. By themselves, these assets constitute a passive immersive scene. Interactivity is unlocked by overlaying three tightly-coupled feedback channels:

- **Downstream high-bit-rate media delivery**: a multicast distribution of audio–visual flows to every participant, rendered locally into spatial audio and immersive video.
- **Upstream micro-interaction events**: low-latency, low-bit-rate signals such as avatar gestures, emoji reactions, haptic triggers, "clap" packets, or crowd noise captured by user microphones.

- **Global state updates**: computed analytics (e.g., cumulative applause intensity, trending emotions, dynamic spot-lighting of fan avatars) that are re-inserted into the media pipeline to shape the concert atmosphere in real time.

6.2 Graph-layer and Data layer

Designing services for the Musical Metaverse (MM) is inherently more complex than a **single-layer** embedding problem. In classical VNE, one typically treats an incoming *virtual-network request* (VNR) as a static graph whose nodes and links must be mapped onto a substrate to optimize some objective (cost, revenue, utilization, etc.). By contrast, MM workloads demand a two-layer perspective:

- **Graph-layer**: At this level the *concert* (or more generally, the MM application) is modelled as a virtual service graph. The canonical question is: ”How should we embed this graph onto the infrastructure to meet a given objective?” still applies, **but** with a crucial twist: the topology is dynamic. Audience members can join or leave at any moment, meaning the embedding must adapt continuously without violating stringent latency or QoS/QoE constraints.

- **Data Layer**: Above the graph mapping lies a **data-centric** layer that captures the real-time behaviour of flows after they are placed. Classical VNE often idealises the substrate, assuming the network can absorb *jitter, packet loss*, or *congestion* within a single link-capacity constraint. In MM scenarios, we cannot make that simplification. End-to-end musical interaction is hypersensitive to **latency variation**, **packet loss**, and even **congestion** spill-over from neighbouring concerts or experiences. The data layer must therefore monitor and react to these runtime conditions e.g., by rerouting specific commodities, invoking redundant edge renderer, or dynamically adjusting bit-rates to maintain the illusion of co-presence.

6.2.1 Modeling a general service graph

In this part, we will discuss how to model the service graph for MM scenarios. Later, we will use this generalized framework to design a customized service graph for the pre-recorded interaction scenarios. A generic service can be described by a *directed acyclic graph (DAG)* $\mathcal{R} = (\mathcal{I}, \mathcal{K})$ where vertices represent the service functions (e.g., stream processing operators), and edges correspond to data streams (or commodities). Figure 6.1 shows an example of a general service graph where source commodities pass through a set of processing functions (Tracking, Synthesis, and Rendering) to finally multicast to their corresponding destinations.

The vertices with no incoming edges of the service graph represent source functions that **produce** *source data streams* (e.g., video capture), and the vertices with no outgoing edges represent destination functions that **consume** *processed data streams* (e.g., video display). Source/destination functions may also represent purely ingress/egress points injecting/ejecting data in/out of the network, and are always associated with a

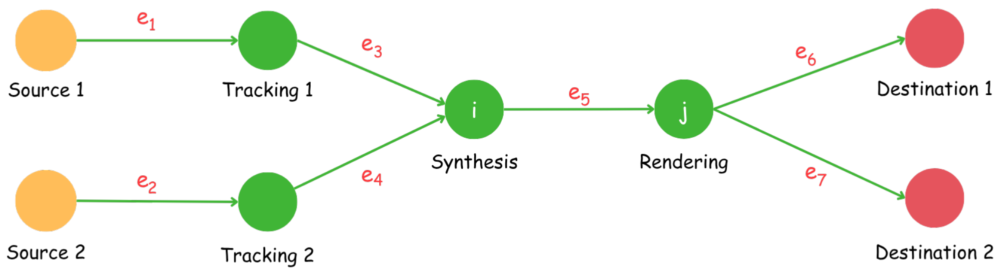


Figure 6.1. Example of a service graph, where edges represent data streams (commodities) and vertices service functions. The example represents a NextG media application, in which streams from two sources go through tracking, synthesis, and rendering functions before being delivered to corresponding destinations.

fixed and unique location in the cloud-network. While the remaining functions are subject to **placement optimization** as shown in Figure 6.1.

An edge $k \equiv (i, j) \in \mathcal{K}$ represents a commodity or data stream produced by function $i \in \mathcal{I}$ and consumed by function $j \in \mathcal{I}$. We use $\mathcal{X}(k)$ to denote the set of incoming edges of node $i \in \mathcal{I}$, i.e. the set of inputs commodities required to produce commodity $k \in \mathcal{K}$ via function $i \in \mathcal{I}$.

To clarify these concepts, we use as an example Figure 6.1. $\mathcal{X}(e_5) = \{e_3; e_4\}$, i.e. data traversing from *synthesis* to *rendering* cannot exist without having data produced by *traking 1* and *tracking 2* ready at node i.

We denote by $\mathcal{K}^s \in \mathcal{K}$, the set of source commodities, i.e. the commodities produced by a source function (in the figure $\{e_1, e_2\}$), and by $\mathcal{K}^d \in \mathcal{K}$ the set destination commodities, i.e., the commodities consumed by a destination function (in the figure $\{e_6, e_7\}$). Finally, Table 6.1 shortlists the type of each commodity in Figure 6.1.

Commodity Set	**Commodities**
$\mathcal{K}^s$	e_1, e_2
$\mathcal{K}^p$	e_3, e_4, e_5
$\mathcal{K}^d$	e_6, e_7

Table 6.1. Type of commodities: $\mathcal{K}^s$ source commodities; $\mathcal{K}^p$ processing commodities; $\mathcal{K}^d$ destination commodities

Analogously, we define $\mathcal{I}^s$, $\mathcal{I}^d$, and $\mathcal{I}^p$ as the set of source, destination, and computation (processing) functions, respectively. In $\mathcal{R}$, each commodity is characterized by its multidimensional rate requirement R^k_{ij}, which denotes the average rate of commodity $k \in \mathcal{K}$ when it goes over link $(i, j) \in \mathcal{E}$. Hence, the rate of a given commodity k will depend on the **type of link** (resource) (i, j) it goes through. That is, commodity k will impose a certain communication rate (e.g., in bps) when it goes over a communication link $(i, j) \in \mathcal{E}^c$, a certain processing rate (e.g., in FLOPS) when it goes over a "computation out" link $(i, j) \in \mathcal{E}^{p_{out}}$, and certain memory rate (e.g. in bits) when it goes over a ”computation in” link $(i, j) \in \mathcal{E}^{p_{in}}$. Note also that communication, processing, and memory rates will be different for different commodities along the service graph.

Finally, one of the most important aspects of the information-aware service DAG model, which allows efficiently leveraging the multicast nature of real-time data streams and their possible replication over the network, is the ability to characterize the actual information or content carried by each commodity. As such, we differentiate between the set of commodities $\mathcal{K}$ and the set of information objects $\mathcal{O}$, and use the surjective *information mapping function* $g : \mathcal{K} \to \mathcal{O}$ to indicate the information object $o \in \mathcal{O}$ associated with each commodity $k \in \mathcal{K}$.

Before concluding this part, we summarize all model notation already mentioned in Table 6.2.

Notation	**Description**
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Cloud network graph, and its associated $\mathcal{V}$ and links $\mathcal{E}$
$\mathcal{V}^p; \mathcal{V}^s; \mathcal{V}^d$	Computation nodes; source nodes; destination nodes
$\mathcal{E}^c; \mathcal{E}^p; \mathcal{E}^s; \mathcal{E}^d$	Communication links; computation links; source links; destination links
$\mathcal{E}^{p_{out}}; \mathcal{E}^{p_{in}}$	Computation in links (storage resources); Computation out links (processing resources)
$\mathcal{N}^+(u); \mathcal{N}^-(u)$	Incoming and outgoing neighbors of node $u \in \mathcal{V}^c$
$c_{ij}; w_{ij}$	Capacity and cost of link (i, j)
$\mathcal{R} = (\mathcal{I}, \mathcal{K})$	Service graph DAG
$\mathcal{I}^s; \mathcal{I}^d; \mathcal{I}^p$	Source functions; Destination functions and processing functions
$\mathcal{X}(k)$	Set of input commodities required to produce commodity k
$\mathcal{K}^s; \mathcal{K}^d; \mathcal{K}^p$	Source commodities; Destination commodities and processing commodities
$\mathcal{Y}^{src}(k); \mathcal{Y}^{dst}(k)$	Source node hosting the function producing commodity $k \in \mathcal{K}^s$; Destination node hosting the function consuming commodity $k \in \mathcal{K}^d$.
$\mathcal{Z}^{src}; \mathcal{Z}^{dst}$	Node hosting source function $i \in \mathcal{I}^s$;Node hosting destination function $i \in \mathcal{I}^d$
$\mathcal{V}^{p,\mathcal{I}}(i) \equiv \mathcal{V}^{p,\mathcal{K}}(k)$	Computation nodes that can host function i and hence produce commodity k
R^k_{ij}	Rate of commodity $k \in \mathcal{K}$ when it goes over link (i, j)
$\mathcal{O}; g : \mathcal{K} \to \mathcal{O}$	Set of information objects; Information mapping function

Table 6.2. Main System Model Notations

6.2.2 Modeling pre-recorded scenario's service graph

In the previous section, we defined the general graph-based formulation of any service graph. In this section, our focus is on building the service graph specific to "audience interaction in pre-recorded virtual concert".

The modeling is based on the following components:

- **Producers** ($\mathcal{I}^s$): The entities generating the continous flow of data
- **Processors** ($\mathcal{I}^p$): The entities responsible for processing the data generated by the producers, possibly receiving information also from databases
- **Consumers** ($\mathcal{I}^d$): The entities receiving the information computed by the processors.

The service graph $\mathcal{R} = (\mathcal{I}, \mathcal{K})$ is the graphical topology of the concert. On the other hand, each user $u \in \mathcal{U}$ will be represented in the topology by its *producing* and *consuming* components. Figure 6.2 illustrates the service graph, assuming that there are two users at the concert.

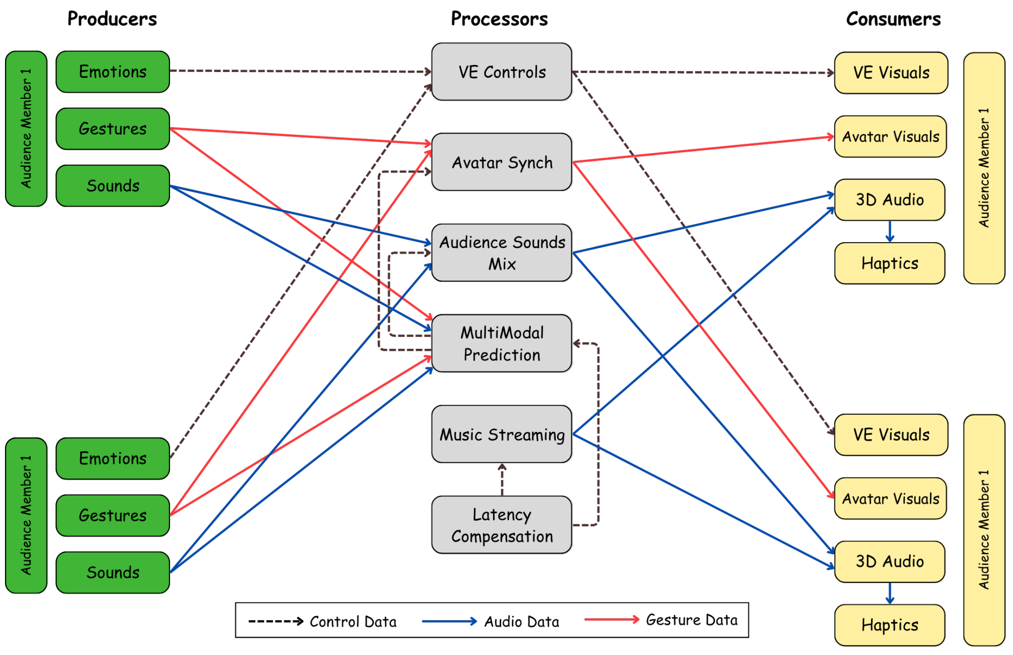


Figure 6.2. Service graph for the "pre-recorded" scenario

The producers are as follows:

- **Emotions:** retrieves in real-time from biometric signals (EEG, heart rate) the emotional state of the audience member (as a class of 4 basic emotions as reported in [33]), and transmits it to the network every 5s using UDP.

- **Gestures:** Tracks the movements of the audience member (typically the position and rotation of head and hands), and transmits them to the network every 5 ms using UDP.

- **Sounds:** Tracks, via the microphone embedded in the HMD (head-mounted display), the voice and contextual sounds generated by the audience member (e.g., while cheering, clapping, singing), and transmits them to the network, as a mono signal, using RTP over UDP with packets encapsulated in IPv6 packets (considering 480 samples per packet, at a sampling rate of 48 KHz and a resolution of 16 bit).

The **processors** are as follows:

- **VE Controls:** takes as input the emotional state of each audience member and produces as output (e.g., via majority voting schemes) a set of control messages for changing the parameters of the VE).

- **Avatars Synch:** takes as input the gestures from each audience member and uses them to control the embodiment of the corresponding avatar, synchronized across all instances of the multi-player VR application. The synchronized gestures are sent to the network every 5ms using UDP, this processor also receives as input the data from the Multimodal Prediction module.

- **Audience Sounds Mix:** takes as input the audio stream of each audience member and creates a mix of the sounds of all avatars, far from the avatar of the receiving audience member, while delivering to him/her the independent sound streams of the closest avatars (up to 10, if their distance from the user is compatible with perceptual constraints). Therefore, the outgoing communication rate can be computed as 11 mono audio streams: (10 users + mix) * 0.816 $Mbps$ = 8.976 $Mbps$. This processor also receives as input the data from the Multimodal Prediction module.

- **Multimodal Prediction:** takes as input the data related to gestures, sounds and delay compensation, and produces predictions for gestures and sounds (via ML methods based on previous data) in order to cope with the excessive latencies that would prevent audio-visual synchronization.

- **Music Streaming:** Takes in input a database of compensation delays and uses them to stream music content to the connected audience members each with a different delay (in this way all geographically displaced audience members receive the music at the same time, regardless of the latency introduced by the network). Music is streamed (as stereo signals, via TCP/IP) with a varying level of audio quality depending on the service plan *(basic, medium,premium).*

- **Latency Compensation:** A network controller that updates a database of network-related information that is used to compute compensation delays for the Music Streaming and Multimodal Prediction modules.

The consumers are as follows:

- **VE Visuals:** runs the multi-player VR application which *updates* the parameters of the VE based on the control messages received from the VE Controls.
- **Avatars Visuals:** runs the multi-player VR application which *updates* the movements of the avatars of the connected users based on the control messages received from the Avatars Synch.
- **3D Audio:** receives as input the position of the 10 closest avatars surrounding the avatar of the audience member and delivers to him/her their sounds spatialized according to 3D audio methods, along with the mix of the sounds of the farther avatars.
- **Haptics:** provides a tactile representation of the music and other sounds, based on the audio signal produced by the 3D Audio device.

6.3 Conclusion

This chapter describes two connected objectives:

- Concrete Service-Graph Instantiation: Using the general CNFlow template introduced in Chapter 5, we derived a fully-annotated DAG for the *audience-interaction in pre-recorded concerts* use-case.
- Two-Layer Problem Decomposition: We argued that the orchestration task cannot be solved by a single, static embedding:
  - The **graph layer** decides where to place each function and how to route/replicate every commodity. This layer operates on a seconds-to-minutes horizon and must remain feasible under churn (users joining or leaving) and replica scale-out/scale-in events.
  - The **data layer** closes the loop at sub-second granularity, reacting to jitter, burst-loss, and cross-concert congestion by micro-rerouting flows. These layers are *coupled*, changes at the data layer may invalidate latency guarantees assumed by the graph layer, while large shifts in audience distribution may trigger a new global embedding.

To satisfy all these requirements, we need a framework where we can place all these variables in a single place. For this reason, in the next chapter, we will introduce a complete Python-based **simulation framework** for musical metaverse scenarios.

Chapter 7

MUSMET Simulator

In this chapter, we present **SiMusMet**, the first Python-based discrete-event simulator that brings the concepts of graph layer and data layer into executable artefacts. SiMusMet allows researchers to *(i)* embed dynamic service graphs on arbitrary cloud-network topologies, *(ii)* stream fine-grained traffic commodities through those embeddings under realistic link-level impairments, and *(iii)* measure end-to-end musical-quality, quality of experience, and quality of services for users.

The simulator follows five practical design principles.

- **Modularity**: Keeps each part: Network model, service-graph engine, traffic generator, and log collector, its module does changes in one place, do not break the rest.

- **Extensibility**: New algorithm and processing blocks can be added as plug-ins rather than rewriting core code. Through a central YAML/CLI interface, every key knob (e.g. capacity, rate) remains **configurable**, making large experiments straightforward.

- **Observability**: The system provides comprehensive metrics and visualization (built-in dashboards and detailed event logs).

- **Simplicity**: Only the essential features are implemented, ensuring the code is easy to read now while leaving clear hooks for future expansion.

In this section, we present the general architecture of our simulator. The design is illustrated in Figure 7.1. Our simulator consists of two modules: *Core Module* and *Scenarios Module*, the former is the principal part of the simulator responsible for the creation, management, and interactions between components. It groups all functionality that is common to every experiment. *Scenarios Module*, a directory module where users (developers) can create their customized scenarios and use cases. Each scenario in the *scenarios* module is treated independently.

7.1 Core Module

The Core Module is the part of the framework that remains intact between experiments (except for setting general configurations), all scenario-specific code lives outside it. It bundles four subsystems that correspond, one-for-one, to the formal elements introduced in Chapter 6.

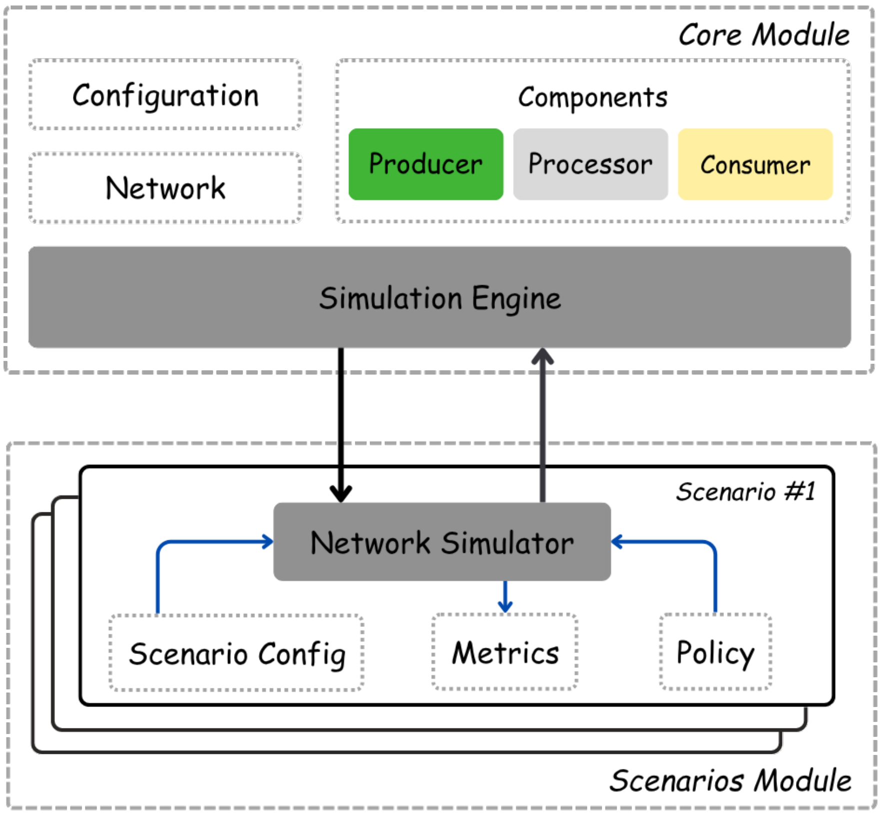


Figure 7.1. SiMusMet Architectural Design

7.1.1 Simulator Configuration

This subsystem serves as the single point of truth for every run-time parameter that the simulator might need. At start-up the engine instantiates a ‘BaseConfig‘ object that:

- **Loads** a default parameter set from a version-controlled YAML/JSON file. These parameters are for example: *simulation time*, *time step*, *random Seeds*, *logging levels*, and *logging paths*.
- **Validates** each entry against a schema and raises human-readable errors.

- **Persists** the effective configuration alongside simulation outputs to guarantee reproducibility.

Scenario authors extend this behavior by sub classing 'BaseConfig' inside their scenario directory. An inherited class can override or add fields, while still benefiting from the same validation and persistence logic. The simulation engine receives a reference to the (immutable) configuration object during initialisation; every other core subsystem (network model, metrics collector, policy plug-ins) reads its parameters through that shared reference. This pattern keeps parameter handling centralised and type-safe, yet lets each scenario tailor the knobs it cares about without touching core code.

7.1.2 Network Subsystem

The Network Subsystem is the bridge between the abstract service-graph logic discussed in Chapter 6 and the event traffic that flows through the simulator at run time. very experiment, whether it stresses only the graph layer (placement / embedding), only the data layer (flow-level dynamics), or both, must run on a coherent *cloud-network graph (CNG)*. This subsystem owns that graph from end to end.

The Network Subsystem therefore performs three high-level roles:

- **CNG Construction**: Assembling a topology whose nodes and links carry detailed resource attributes (e.g., packet loss, jitter, propagation delay). Moreover, a user can easily add some other dimensions.
- **Semantic Validation**: Ensuring that the constructed topology obeys domain-specific constraints so that no unrealistic artefacts bias the results. As an example, imagine two nodes, one is a 'core' and the second is an 'edge' node. As we already know that computing capabilities for edge nodes are limited with respect to core nodes. Such a violation of realistic conditions will be automatically raised by the Network Unit.

At the start of every run the subsystem ingests either (i) a ready-made template derived from real deployments or (ii) a synthetic topology specification generated for specific studies. By capturing heterogeneous resource types in one place, the simulator can represent composite constraints such as *compute-heavy* but *bandwidth-light* links or *memory-rich edge nodes*, features that are typical of modern 5G&Beyond, and thereby saving researchers from drawing conclusions on physically impossible networks.

7.1.3 Components

A service graph inside the simulator is realised as an **ordered set of components**, the vertices of the directed-acyclic graph introduced in Chapter 6. Each component encapsulates the internal logic of a single service-graph node and is instantiated as one of three canonical roles:

- Producer: Generate source commodities (e.g. "Gestures", "Sounds")

- Processor: Transforms or fuses incoming streams (e.g. "Audience Sounds Mix")
- Consumer: Delivers the final output to end-devices (e.g. "3D Audio")

Figure 6.2 shows how these roles combine to form the pre-recorded-concert template.

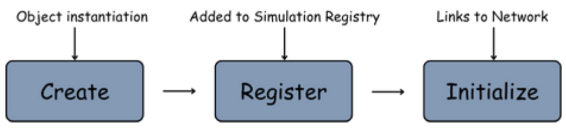


Figure 7.2. Component Lifecycle

The simulator ships with a catalogue of well-documented, ready-to-use components that cover the reference scenario. Researchers may nevertheless need custom-logic, perhaps a novel ML-based predictor. This is achieved by subclassing the appropriate abstract base class (Producer, Processor, Consumer), or, if a fundamentally new role is required, by extending the generic Component superclass. The inheritance hierarchy enforces consistent method signatures and guarantees that custom code remains compatible with the rest of the framework.
Figure 7.2 summarises the three mandatory stages for every component:

- Creation: the developer instantiates the object and invokes its *create()* method. During this phase the simulator validates the provided metadata (I/O rates, state size, latency budget) against domain constraints.
- Registration: Once validated, the component is added to the global registry, making it discoverable by the placement and routing routines.
- Initialization & Self-test: after all required components are registered, the simulator connects them via provisional data-flows and exchanges lightweight "dummy" messages to verify type compatibility, buffer sizes, and back-pressure signals. Any mismatch triggers a descriptive error before the actual experiment begins.

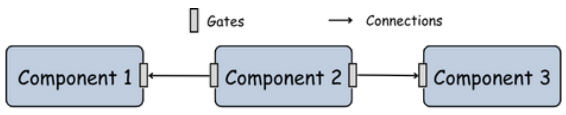


Figure 7.3. Connecting components approach

Components are joined by explicit Data Flows (Figure 7.3), each of which binds an output gate of the upstream node to an input gate of the downstream node. The connection routine rejects disjoint or cyclic graphs, ensuring that the final topology satisfies the DAG requirement and that every commodity has a well-defined path from its producer to at least one consumer. Successful initialization promotes the provisional flows to active links, completing the service-graph assembly.

This disciplined life-cycle, enforced by the Components Subsystem, guarantees that every service graph loaded into the simulator is syntactically correct, semantically coherent, and immediately ready for placement on the cloud-network graph.

7.1.4 Simulation Orchestrator

The **Orchestrator** is the control plane of the entire simulator, the entity that knows every component, sees every link, and advances simulated time. Its design is deliberately centered on a discrete-event scheduling paradigm, because audience-interaction workloads exhibit highly uneven traffic bursts and tight latency budgets that are best captured by event-driven, rather than fixed-timestep, execution. What follows is a comprehensive description of how the Orchestrator is constructed, why each design choice was made, and how it interacts with the other subsystems already introduced.

At a high level the Orchestrator fulfils four responsibilities:

- Global time monitoring: It moves the simulation clock to the timestamp of the next event in the queue. By default, the simulator works without a regular "tick", however, the simulator also allows fixed time step jumps.
- Event Dispatcher: It invokes the callback attached to each event, thereby triggering computation, communication, or rendering inside the appropriate component. The event also accepts a set of optional arguments required.
- State book-keeper: It updates link capacities, queue lengths, and component states after every callback, guaranteeing that subsequent events observe a consistent world view.
- Variability engine: It injects stochastic effects (propagation-delay distributions, packet-loss bursts, device jitter) at the precise moment they materialise, this information are initially provided by the user in the configuration files.

Every occurrence inside the simulator, sending a packet, completing a CPU task, finishing an audio render, is encoded as an **Event object** with three immutable fields: unique Id, timestamp (at which the event must start executing), and callback (the function to execute).

The Orchestrator holds a single **binary min-heap** keyed on timestamp. Inserting, peeking, and popping events all run in $\mathcal{O}(logN)$, where N is the queue length. Thanks to the full control of the orchestrator, it also manages cancelling events in case of loss of information (e.g., data packet corrupted during transmission).

Lifecycle of a packet

- **Production**: The "Sounds" producer crafts a UDP packet and hands it to the Orchestrator via *'scheduleEvent(currenttimestamp,, data packet)'*.

- **Propagation**: The orchestrator receives this request, and schedules an event (creates an event and inserts it inside the queue), this event will have a *timestamp = currenttimestamp + propagationDelay*, and the packet, may also encounter some changes in case of corruption, loss or other possible external effects. Finally, a callback, which is the function that will be executed when the event time comes (event to be processed). This function in this case will be *retrieve_data()* that will be executed by each destination endpoint.

- **Reception** After the event has been processed, the consumer component stores the packet in a rendering buffer and schedules an event (this time, within its internal buffer) for collecting (ingesting) data.

- **Rendering** The consumer calls a specific function to render this data, and accordingly, applies some changes to the current state. The network simulator here has a full view of the state of all consumers and thus, collects or updates QoE/QoS metrics.

Propagation delays, retransmission timers, and even user-behaviour spikes are sampled at the *moment* an event is scheduled, not when it fires. This separation prevents downstream callbacks from changing the past, once an event is in the heap, its timestamp is immutable. All pseudo-random draws are sourced from a single, seeded generator stored inside the configuration module and accessible by the orchestrator.

The Orchestrator is more than a scheduler; it is the simulator's *core* system. By representing every network, computation, and rendering action as a time-stamped event, it harmonises the graph-layer placement logic with the data-layer traffic dynamics, enforces reproducibility, and provides the hooks required for rigorous performance measurement. Its discrete-event architecture is therefore indispensable to studying the tight latency loops and bursty workloads characteristic of audience interaction in the Musical Metaverse.

Finally, we provide in Figure 7.4 a more in-depth view of the core engine where each module/subsystem is shown along with its parts.

7.2 Scenarios Module

This workspace allows users to create their customized scenarios. Figure 7.1 shows the general architectural design of a scenario inside the workspace, however this design may change depending on the user's goals. As mentioned in chapter 6, the need for a specific simulator for the musical metaverse is the complexity of the multi-layer problem. Our simulator allows to create experiments on a specifc layer (data layer or graph layer), or both.

Figure 7.5 shows a visual example of the hierarchical problem. First, focusing on the **data layer** (separately), the problem becomes only studying and experimenting with data-driven communication between components. In other words, the goal in this case is just to understand the behavior of such a customized inter-service-graph communication under some variability assumptions (e.g. burstiness, jitter and loss ...). Our simulator

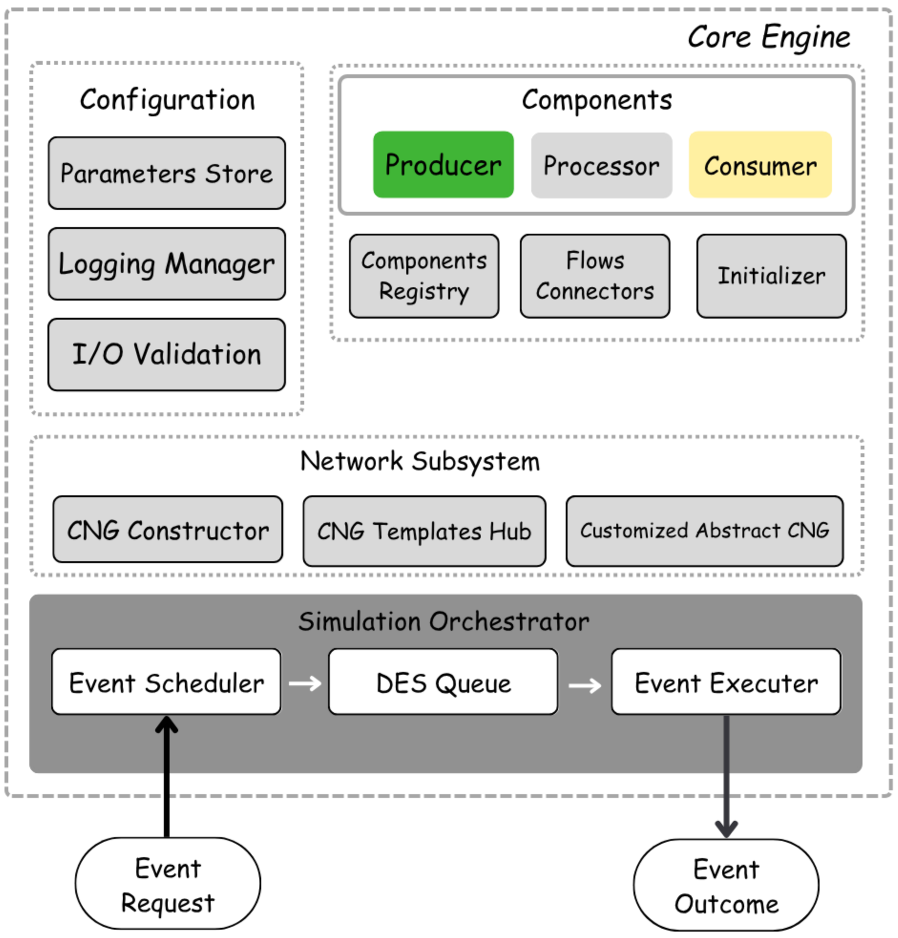


Figure 7.4. In-depth view of the ”core” engine

can offer this easily by setting the environment to data-layer, thus parameters and settings related to the CNG won't be considered. Another view of the problem is more abstract (simpler), as a graph-layer problem (reduced back to simple VNE - check chapter 4). In this case, we assume a negligible effect of variable arguments, and the focus of the scenario will be only devoted to the design of a policy to satisfy an objective related to the embedding problem.
What makes our simulator powerful is the combination of two layers, as a single problem, we are not just interested into achieving some embedding goals, but also considering instateneous changes and fluctuations like realistic networks. As an example of this, we refer to Figure 7.5. The top layer denoted as *Service Graph 1*, has been allocated over a set of nodes on top of the cloud network graph (mappings are shown with dotted green lines), small blue circles represent real-time data transmission between components. Similarly, another service graph (*Service Graph 2*), is allocated such that the red link is now shared between two commodities from the service graphs, which results

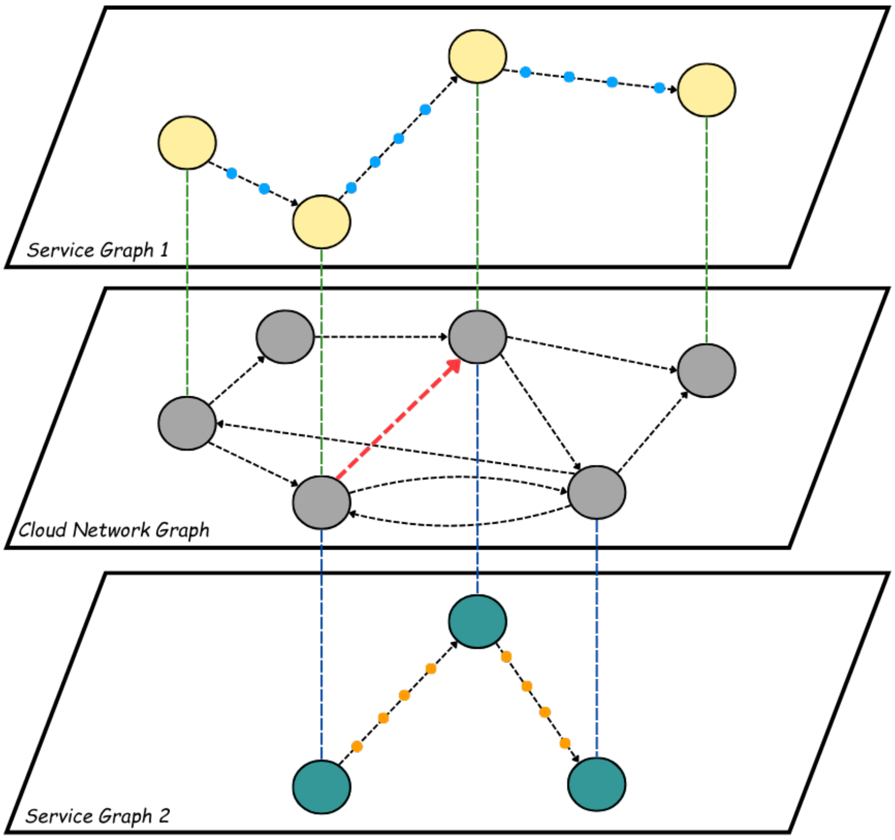


Figure 7.5. An illustration for the two-layers problem

in a more congested link, and therefore, increases variables like jitter and packet loss (more disruptive). Our simulator can easily adapt to these sub-scenarios by varying the probabilities of these actions depending on the load factors, which imitate realistic use cases.

7.2.1 Scenario Configuration Module

Can extend the set of initial configuration parameters defined in the "core" module. The reason behind this design is to ensure separability; only the necessary parameters are called, otherwise they remain set to default. In addition to this, we store the necessary scenario configuration in this module. Following the documentation, we share with the user the set of possible parameters to consider for the pre-defined scenarios; however, if the user wants to create a completely customized scenario out of the provided template, this may require additional steps as values has to be defined (and validated) first in the core part.

7.2.2 Policy Module

This module contains a set of abstract classes and methods to be extended for easy design of a customized policy. The policy's objective is to perform a placement & routing of the incoming service graphs following static and dynamic approaches. The researchers are now able to test their musical metaverse algorithms (ILP, heuristics, or ML-based models) easily on realistic MM scenarios thanks to the realistic cases designed by the simulator. The creation of the policy comes with detailed documentation and a set of examples to make the process of creation easier for users and encourage future contributions. This module is also flexible, i.e, you can just focus on either placement or routing (the other case will be handled by built-in functions like multi-dimensional resource-constrained shortest paths). In the next chapter, we will show how we used this module to build three different customized policies (two MILPs and one heuristic) for the placement and routing of processors, assuming fixed places of users (this assumption is taken to focus on a specific aspect of the problem, otherwise the problem will become similar to VNE).

7.2.3 Metrics Module

The metrics module is responsible for collecting real-time metrics; it has full access to all component states and, therefore, can monitor the changes of these statuses and update the list of metrics. This list depends on the scenario; however, a user can easily extend the base classes provided in this module to create any customized metric along with the interval of measuring or the set of components to use for collecting results.

7.2.4 Network Simulation Module

The main part of any created scenario is the component responsible for bridging the communication with the core layer, and it is the component that receives callable from the simulation engine (orchestrator) to execute over the network, and in turn, send information requests and updates to the orchestrator (for scheduling events). This subsystem is the place where all connections inside the simulator meet. From the CNG to the SG, and where metrics are collected before being sent to the "Metrics" module. It represents the core functionality of the whole experimental scenario.
This has a strict and clear prototype to follow. Initially, we have a base abstract class (this documentation provides easy-to-follow instructions for seamless extensibility) that requires the following attributes:

- A configuration file (scenario config): extended by the "core" configuration module and based on the "scenario configuration" module.
- A Network Template: configured and ready to use Cloud Network Graph topology.
- A policy (if the data-layer only experiment can be discarded).
- A copy of the registered service graphs templates.

Having these attributes is enough to start the simulation. These attributes (blocks) can be easily called inside the network simulator module as they have already been verified by their base classes. At the end, the user can easily run the experiment and start monitoring logs during the simulation, and collect results after the simulation time is reached.

7.3 Conclusion & Summary

This chapter introduced **SiMusMet**, a novel Python-based discrete-event simulator designed to bridge the theoretical concepts of graph layer and data layer into practical, executable artifacts within the context of the Musical Metaverse. SiMusMet empowers researchers to effectively embed dynamic service graphs onto diverse cloud-network topologies, simulate fine-grained traffic under realistic network impairments, and accurately measure end-to-end musical quality, quality of experience (QoE), and quality of service (QoS) for users.

The simulator's architecture consists of two primary modules: the Core Layer (Figure 7.4) hosts all run-time services that remain constant across experiments. And, the Scenarios Workspace empowers researchers to prototype, share, and reproduce experiments without touching the Core. Through this separation of concerns, SiMusMet supports three research modes: pure data-layer tests, pure graph-layer embeddings, and fully coupled studies where placement decisions and packet-level dynamics feed back into each other in real time (Figure 7.5). By combining a rigorously validated network model with an event-accurate execution kernel and an extensible scenario interface, the simulator offers a reproducible, researcher-friendly platform for exploring latency-critical, interactive audio-visual workloads at scale for musical metaverse scenarios.

The next chapter leverages this foundation to develop and evaluate three concrete placement–routing policies, two MILPs, and a **novel** topology-and-compute aware heuristic algorithm.

Chapter 8

Placement and Routing Policy Design

With **SiMusMet** in place, we can now shift from *how* to simulate to *what* to optimize. In this chapter, we introduce a new MILP formulation for the placement and routing of processors, given the locations of users. This template is the same as the template shown in Figure 6.2. Given a service graph for the pre-recorded scenario explained in chapter 6, our goal is to examine the placement and routing of these components while satisfying some constraints and achieving an optimization goal.

8.1 Problem Formulation

Figure 8.1 provides an illustrative example of a simple use case, given a set of users $\mathcal{U} = \{u_1, ..., u_n\}$ already connected to the network (their components are connected to a specific set of communication nodes), the problem we want to solve is where to place the processors of the service graph so that we min/max a specific objective value. As an example, we see in Figure 8.1 that a user (denoted as u_1) is connected to the communication node C_2, while u_2, is connected to communication node C_4.

In this example, imagine we have a single processor to place over the network, which receives input from users producers, and outputs to users consumers. Suppose also that this processor's requirements over computational commodities do not violate any of the shown computational nodes in the network; therefore, we end up having 4 different computational nodes as feasible candidates for the placement of the processor. After selecting the appropriate node, we can move to the second phase (routing) and validate feasibility and efficiency. The selection of the candidate node (for placement) does not necessarily lead to the best possible results, depending on the objective goal, and the parameters over commodities (e.g. delay, bandwidth), we may need to use longer paths.

In this cloud network graph, the number of processing nodes is limited, and the selection of candidates must be based on the satisfaction of requirements (e.g. sufficient amount of capacities over commodities, or delay constraints are not violated), while achieving the best objective value. Therefore, the problem can be written as a Mixed-Ineger Linear Programming Formulation as shown in the next section.

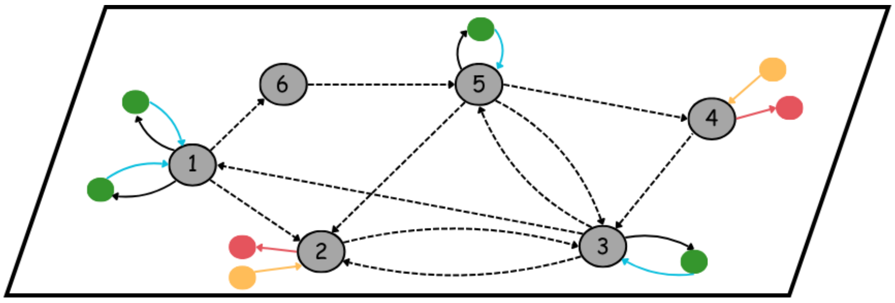


Figure 8.1. An example of a cloud network graph with 2 users already connected

8.2 General MILP formulation

First, we invite the reader to check Table 6.2 for the set of main notations. In addition to these notations, we define the following variables:

1. *Virtual Commodity Flows* $\{f_{ij}^k\}$: A dimensional binary vector indicating whether commodity $k \in \mathcal{K}$ goes (i.e., is transmitted, processed, or stored) over link $(i,j) \in \mathcal{E}$.

2. *Actual Information Flows* $\{\mu_{ij}^o\}$ *and* $\{\mu_{ij}\}$: Real variables indicating the **amount** of information flow associated with object $o \in \mathcal{O}$ and the **total** information flow, respectively, going over link $(i,j) \in \mathcal{E}$.

The resulting mixed integer linear program (MILP) is described as follows:

$$\max / \min \quad \text{Objective} \tag{8.1}$$

$$\text{s.t.} \quad \sum_{j \in N^-(i)} f_{ji}^k = \sum_{j \in N^+(i)} f_{ij}^k \qquad \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{8.2}$$

$$f_{ij}^k = \begin{cases} f_{ji}^l & \forall k \in \mathcal{K}^p \\ 0 & \text{otherwise} \end{cases} \qquad \forall l \in \mathcal{X}(k), i \in \mathcal{V}^p, j \in \mathcal{N}^+(i) \tag{8.3}$$

$$f_{ij}^k = \begin{cases} 1 & \forall k \in \mathcal{K}^s \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in \mathcal{Y}^{src}(k), j \in \mathcal{N}^+(i) \tag{8.4}$$

$$f_{ij}^k = \begin{cases} 1 & \forall k \in \mathcal{K}^d \\ 0 & \text{otherwise} \end{cases} \qquad \forall j \in \mathcal{Y}^{dst}(k), i \in \mathcal{N}^-(j) \tag{8.5}$$

$$f_{ij}^k R_{ij}^k \leq \mu_{ij}^o \qquad \forall (i,j) \in \mathcal{E}, k \in \mathcal{K}, o = g(k) \tag{8.6}$$

$$\sum_{o \in \mathcal{O}} \mu_{ij}^o \leq \mu_{ij} \leq c_{ij} \qquad \forall (i,j) \in \mathcal{E} \tag{8.7}$$

$$l^k = \sum_{(i,j) \in \mathcal{E}} l_{ij}^k f_{ij}^k \qquad \forall k \in \mathcal{K} \tag{8.8}$$

$$l_T^k = l^k \qquad \forall k \in \mathcal{K}^s \tag{8.9}$$

$$l_T^k \geq l^k + l_T^l \qquad \forall k \in \mathcal{K}^p \cap \mathcal{K}^d, l \in \mathcal{X}(k) \tag{8.10}$$

$$l_T^k \leq L^k \qquad \forall k \in \mathcal{K}^d \tag{8.11}$$

$$f_{ij}^k \in \{0,1\}, \mu_{ij}^o \in \mathbb{R}^+, \mu_{ij} \in \mathbb{R}^+, l^k \in \mathbb{R}^+, l_T^k \in \mathbb{R}^+ \qquad \forall (i,j) \in \mathcal{E}, k \in \mathcal{K}, o \in \mathcal{O} \tag{8.12}$$

In the provided MILP, the objective is not yet given. The reason is to focus first on the set of decision variables and constraints that are fixed for any objective design. Later, we will show how, by selecting an objective, we may need to add some additional constraints. For now, we consider an unknown objective function.

Equation 8.2 states generalized (communication, computation, storage) flow conservation constraints, requiring the total incoming flow to a given communication node $i \in \mathcal{V}$ for a given commodity $k \in \mathcal{K}$, to be equal to the total outgoing flow from node i for commodity k. I.e., *no data is lost or created.* This assumption holds when we solve this for a specific instant of the time horizon (δt).
Equation 8.3 states flow chaining constraints, which impose that to generate commodity $k \in \mathcal{K}$ at the output of computation node $i \in \mathcal{V}^p$, all input commodities $l \in \mathcal{X}(k)$ must be present at the input of node i. As an example, suppose that k is a video enhancement output, (the processing function enhances the quality of the video), thus, l is needed as input (decoded video) to produce k.

Equations 8.4 and 8.5, are source and destination constraints that initialize the ingress/egress of the source/destination commodities at their corresponding source/destination nodes. So, each commodity $k \in \mathcal{K}^s$ is **generated at a unique node** $i = \mathcal{Y}^{src}(k)$. That node must push the flow $f_{ij}^k = 1$ to its outgoing neighbors. This is how the MusMOPT formulation injects data into the network. If the flow isn't initialized, no downstream processing or delivery can happen.
One of the most important elements of this MusMOPT formulation is the connection between virtual commodity flows and actual information flows. Recall that a unique aspect of NextG services (musical metaverse services) that cannot be captured via VNE models is the sharing of data streams by multiple processing and/or destination functions. Such multicast nature of NextG media streams means that different virtual commodity flows carrying the same information must be able to overlap when going through the same link $(i, j) \in \mathcal{E}$. This is assured by Equation 8.6, where we first multiply the commodity flow variables by their corresponding rate requirement and then allow the overlap of the resulting sized commodity flows that are associated with the same information object. The total information flow at a given link $(i, j) \in \mathcal{E}$ is then computed by summing over all information flows, which is naturally constrained to be no larger than the total capacity of link (i, j), as stated in Equation 8.7.
The end-to-end service latency constraints are governed by equations Equations 8.8 8.9 8.10 8.11. Equation 8.8 computes the local latency of commodity k, l^k, (i.e., the time taken to produce, deliver, and consume a unit of commodity k) as the sum, over the links carrying commodity k, of the latency to transmit or process a unit of commodity k over the given link, denoted by l_{ij}^k. Equations 8.9 8.10 compute the cumulative latency of commodity k, l_T^k , which represents the service latency that has been accumulated until the consumption of commodity k. Equation 8.9 first sets the cumulative latency to be equal to the local latency for all source commodities. Equation 8.10 then computes the cumulative latency for all remaining commodities recursively by setting the cumulative latency of commodity k to be larger than or equal to the local latency of commodity k plus the cumulative latency of input commodity l, for all input commodities in $\mathcal{X}(k)$. Lastly, Equation 8.11 imposes the cumulative latency at each destination commodity to be no greater than the maximum allowed service latency L^k.
Finally, Equation 8.12 imposes the binary nature of commodity flow variables and the real positive nature of information flow and latency variables.

8.3 Objective-Specific MILP Instantiations

The generic formulation in previous section, captures all hard constraints that every musical-metaverse deployment must observe: flow conservation, information-aware multicast, multiresource link capacities, and end-to-end latency bounds. What drives the optimizer toward one placement rather than another is the objective function sitting atop those constraints (Equation 8.1). In real networks, operators rarely optimise a single metric; they trade off monetary cost against resilience, jitter tolerance, or fairness. To expose that design space we propose the template of two complementary MILPs:

1. **Minimum-Cost MILP (Min-Cost):** This model seeks the cheapest feasible

embedding by minimizing the sum of ”costs” over utilized link resources.

2. **Load-Balancing MILP (LB)**: Interactive music streams are hypersensitive to jitter bursts that arise when a few network hot spots saturate. Here the goal is to **spread load evenly** so that the *most* congested resource is as lightly used as possible.

The min-cost objective is shown in Equation 8.13. The goal is to minimize the total cloud-network resource cost, where recall that edges in $\mathcal{E}$ can represent communication, computation, or storage resources. As a showcase example, we define two processors p_1 and p_2, and the goal is to place these processors over the CNG shown in Figure 8.1. Since the computation node placed in node C_5 can accomodate both p_1 and p_2, and since this placement will lead to the lowest possible cost, the solution will be to route data from p_1 and p_2 to/from u_1 and u_1 over the links $C_5 \rightarrow C_4$, and $C_5 \rightarrow C_2$. However, such routing makes both links more congested.

$$\sum_{(i,j)\in\mathcal{E}} \mu_{ij} w_{ij} \tag{8.13}$$

On the other hand, making the network balanced in terms of load means that we have to minimize the maximum utilization over links (i.e. maximum loaded link), a link is loaded when its capacity is near saturation. This can be modeled as described in Equation 8.14.

$$\min(\max_{(i,j)\in\mathcal{E}} u_{ij}) \quad u_{ij} = \frac{\mu_{ij}}{c_{ij}} \tag{8.14}$$

where u_{ij} is the utilization (or load) of links (i, j). The nested min-max breaks linearity, so we linearize it back using the standard epigraph trick. To do that, we use a single auxiliary variable $\mathbf{Z}$, resulting in an additional constraint. The final form becomes:

$$\begin{aligned}
\text{minimize} \quad & \mathbf{Z} && && (8.15)\\
\text{s.t.} \quad & \mu_{ij} \leq \mathbf{Z} \cdot c_{ij} && \forall (i,j) \in \mathcal{E} && (8.16)\\
& \mathbf{Z} \in \mathbb{R}^{+} && && (8.17)\\
& \text{Equations 8.2 - 8.12} && && (8.18)
\end{aligned}$$

The exact MILP models derived above yield optimal embeddings, but their solver runtime grows super-linearly with both user count and topology size. For concert-scale scenarios (hundreds of avatars, tens of edge sites) even a warm-started Gurobi run can exceed practical planning windows. To reconcile **solution quality** and **computational tractability**, we next introduce a lightweight heuristic that blends greedy placement with load-aware path selection. The algorithm completes in sub-second time on our largest testbed, yet, as the results in the following chapter show, it achieves near-optimal results.

8.4 Heuristic Placement & Routing Algorithm

The two MILP formulations just presented give us the performance ”end-points” of the orchestration space. *Min-Cost* delivers the cheapest feasible embedding, whereas *LB* yields the flattest utilization profile, but neither scales to the decision latencies required by a live, audience-driven concert. Imagine that at every δt (a tiny instant of the network/simulation state) you need to solve the MILP that takes a longer time than expected, leading to a degraded performance. In this section, we introduce a **Topology-Aware Cost-Load** heuristic algorithm designed to balance the trade-off between:

- Computational Speed and Efficiency
- Cost consciousness
- Load Restraint
- Topology awareness
- Fair distribution of utilized resources

8.4.1 High-level Algorithmic Flow

Our heuristic consists of the following five phases:

1. Community Detection (cluster the CNG)
2. Processors Demand-aware prioritization
3. Scoring of “grouped” processors over a community-level
4. Initial Placement & Routing
5. Iterative exploration based on multi-variable probabilistic attenuation

Figure 8.2 is a flow chart for the sequence of steps done by the algorithm. First, the algorithm takes the CNG and SG as input, applying a clustering method to the former and a ranking method to the latter. Then, we compute a score for each processor across all communities. Finally, we start with an initial placement and routing step, and iterate for a fixed number of iterations T_{MAX} to get the best possible placement and routing solution. The algorithm returns as output a mapping with keys that are the set of commodities of the SG, and the values correspond to the path in the CNG.

In the upcoming sections, we will describe each phase independently and explain what each phase shares with the successive one.

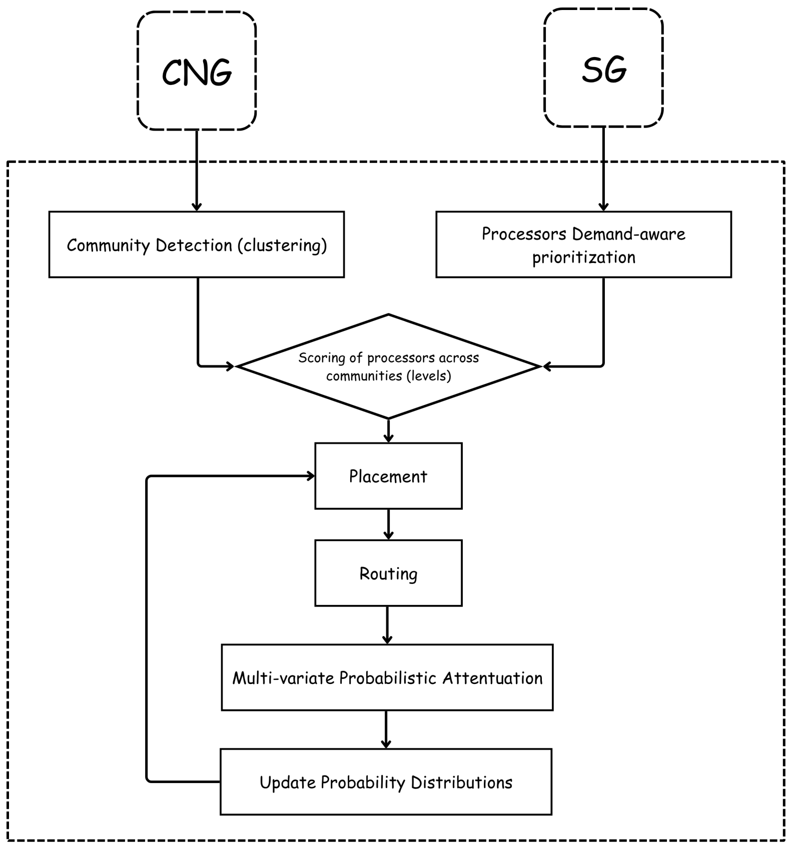

CNG
SG
Community Detection (clustering)
Processors Demand-aware prioritization
Scoring of processors across communities (levels)
Placement
Routing
Multi-variate Probabilistic Attentuation
Update Probability Distributions

Figure 8.2. High-level flowchart for the heuristic algorithm

8.4.2 Community Detection

Real-world graphs (e.g. communication networks) are not random; their nodes (and links) form naturally into groups or communities. Communities are sets of dense internal connections, while this density reflects the definition of connection between components. Detecting these communities has a lot of benefits; these benefits motivate the selection choice of this approach:

- Simplify the problem space: By dividing the search space into groups, the search becomes less expensive.
- Preserve structural/topological properties: The grouping is done based on a computation of similarity between components; therefore, elements within the same cluster tend to share similar features or structures.
- Enhance embedding or partitioning heuristics

Definition 8. *Community detection is the task of partitioning a graph into clusters (communities) where: Nodes within the same community are* ***densely*** *connected. Nodes from different communities are* ***sparsely*** *connected, and the concept of* ***density*** *is relative based on the definition of the scoring function (e.g., number of hops, weight, distance, costs ...).*

A widely adopted way to evaluate how well a partition captures that internal-dense / external-sparse pattern is **modularity**, introduced in [19]. For a given graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $m = |\mathcal{E}|$ and a partition $C = \{C_1, C_2, ..., C_k\}$ of the vertices, modularity Q is defined as :

$$Q = \frac{1}{m} \sum_{i,j} (A_{ij} - \frac{k_i k_j}{2m}) \delta(\sigma_i, \sigma_j) \tag{8.19}$$

where:

- A_{ij} counts **observed** links inside the community.
- $\frac{k_i k_j}{2m}$ is the **expected** number of such links in a null model that preserves the degree sequence but randomizes endpoints.
- The sum therefore measures "observed minus expected" internal connectivity, normalized to lie in $[-1,1]$
- High positive values (typically 0.3-0.7 for realistic graphs) imply many more internal edges than randomness predicts; values near zero indicate no community structure.

Because the space of all partitions grows super-exponentially with $|\mathcal{V}|$, finding the global maximum of Q is $\mathcal{NP}$-Hard. Practical algorithms, therefore, rely on heuristics that **approximate** the maximum while scaling to graphs with millions of nodes. The most popular is the **Louvain method** [4]:

- **Local moving**: Start with each vertex in its own community. Iteratively move a vertex to the neighbouring community that yields the greatest positive ΔQ until no local move can improve Q.
- **Aggregation**: Collapse each community into a "supernode", recompute edge weights (sum of original weights), and repeat step 1 on the condensed graph.
- **Hierarchy**: Iterate until ΔQ becomes negligible; the algorithm outputs a dendrogram of communities across scales (level).

Louvain runs in roughly $\mathcal{O}(|\mathcal{E}|)$ time per pass, making it suitable for the cloud network graphs (a few hundred to a few thousand nodes) used in our simulator. Louvain offers the best trade-off between speed and Q-quality for our purposes compared to other algorithms like Leiden [29] (solves disjoint graphs in Louvain) and CNM [8] (greedy agglomeration).

Figure 8.3 provides an example of how Louvain works: First, as mentioned before, nodes are sorted into communities based on how the modularity of the graph changes when a node moves communities. Then the graph is reinterpreted so that communities are seen as individual nodes.
The Louvain method begins by considering each node v in a graph to be its own community. For each node v, we consider how moving v from its current community C into a neighboring community C' will affect the modularity of the graph partition. This happens in the for-loop. We select the community C' with the greatest change in modularity, and if the change is positive, we move v into C'; otherwise, we leave it where it is. This continues until the modularity stops improving.

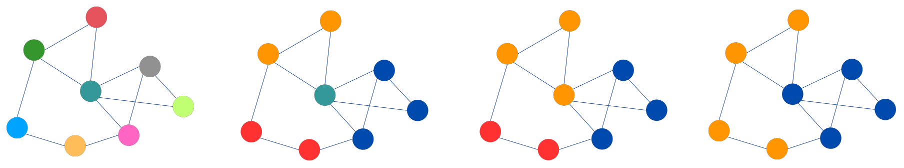


Figure 8.3. Louvain algorithm grouping stage

Now, to compute the clusters (communities), we need to have A_{ij}, the matrix that represents the weights of any pair of nodes, at each level. We need to compute an **informative and efficient** weighting for each link that can be carefully representative of all the information related to this link. For this reason, we propose our novel weighting function. Given a link (i, j).

$$w_{ij} = f \cdot (\alpha \frac{w_{ij}}{c_{ij}} + \beta \max(\Lambda, \nabla) + \gamma\zeta) \tag{8.20}$$

where:

- $\alpha = 0.4; \beta = 0.3; \gamma = 0.3$
- w_{ij} & c_{ij} are the cost and capacity of the link (i, j) respectively
- We denote by $N((i, j))$ the set of two endpoints forming the link (i, j), since our method is applied over the communication subgraph of the CNG, each endpoint may have from zero to a finite number of computational nodes. We denote by $\lambda_{in} = \sum_{(i,j)\in\mathcal{E}^{\rangle\setminus}} c_{i,j}$ the sum of all RAM resources over the network. Finally, $\Lambda = \frac{\sum_{p\in N(i,j)}}{\lambda_{in}}$ is the ratio between the cost of RAM resources related to computation nodes that are connected to communication nodes forming this link, and the overall RAM resources of the network.
- ∇ has a similar definition, it is the ratio between the sum of all CPU resources of the computation nodes that are connected to communication nodes forming this link, over all CPU resources of the network.
- ζ is the betweenness centrality score of the link, and is calculated by counting how many shortest paths between all possible pairs of nodes in a network pass through a specific edge.

By this, we defined all parameters of our equation (except f), and we can see that the formula takes into consideration several aspects of the link (topology, cost, ability to host, closeness to computational node). However, as our scenario is related to the placement of processors given fixed places of users. Our formula contains f factor that accordingly changes based on the existence of a connection with a user on one of the endpoints of the link. As shown in Figure 8.4, blue edges represent a communication node with no user connection, while a read node represents a communication node with a user connected to that node. In the case of 2 users, we show the three possible outcomes. In the first, $f = f_0 \cdot 1$ because of a lack of users, in the second, the factor is multiplied by 1.5, so it becomes higher, and similarly for the third case. This means that the weight of the edge increases if one of the endpoints hosts a user, which reflects informative clustering depending on the information provided.

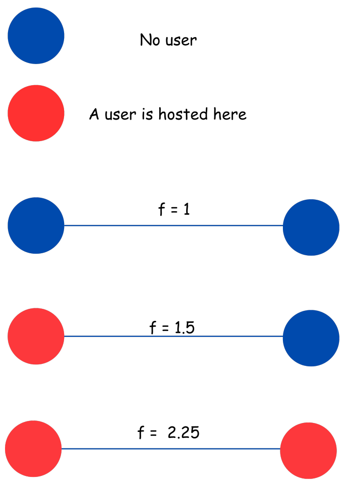


Figure 8.4. Example how f evolves depending on number of users

At the end of this phase we run our modified Louvain algorithm on the sub-graph of communication nodes. Because Louvain is iterative, we record the partition produced after every pass; the successive partitions form a dendrogram, shown in Figure 8.5. The leaves correspond to single-node communities, while higher levels reveal progressively coarser clusters obtained by merging the denser groups identified below.

8.4.3 Service Graph Ranking

This phase is also known as "demand-aware processors prioritization" which consists of ranking our 6 processors (as mentioned above, our focus is on the scenario of pre-recorded concerts). We rank our processors according to the demand for their input and output commodities.

$$d_{p \in \mathcal{V}^p} = \sum_{k \in \mathcal{Y}^{src}(k), k=(i,j)} R_{ij}^k \tag{8.21}$$

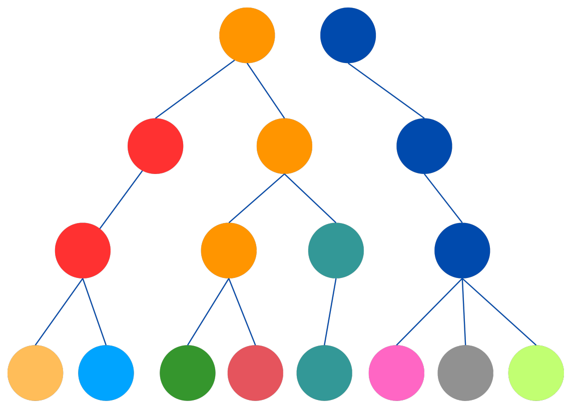
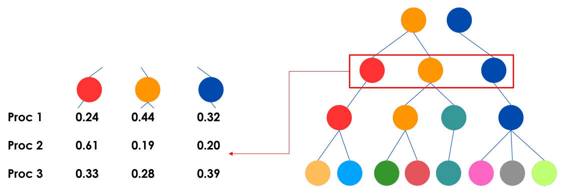


Figure 8.5. Tree-based view of the Louvain Method

This means that the processors having more demand (sum of multi-dimensional requirements) as input and output will get a higher rank because by handling heavy-demand processors first, we reduce the complexity of the search.

8.4.4 Community Scoring and Candidate Selection

Having ranked processors and partitioned the cloud graph into a multilevel community tree, the heuristic now needs a principled way to **decide which set of communities (level) should host the processors** in the queue. This is done with a single scalar metric denoted as κ (community score). The process is described in details in Algorithm 1, where the goal is to find a level where each processor is *"feasible"* for at least one community. The reason behind this is to reduce the search space while carefully selecting a smaller search space that lead to better results.



	●	●	●
Proc 1	0.24	0.44	0.32
Proc 2	0.61	0.19	0.20
Proc 3	0.33	0.28	0.39

Figure 8.6. An illustrative example of W^{inter}

Finally, Figure 8.6 shows an example of W^{inter} considering 3 different processors. After selecting a feasible set of communities in a specific level for the processors (here, it turns out that all communities are feasible to the processors), we normalize the scores. Thus, we obtain that the sum of scores of a processor over the *"feasible"* communities is equal to 1.

Algorithm 1 Processors score computation over communities

Input: List of Processors $\mathcal{V}^p$, List of levels L, each level is a list of $|CM_L|$ communities
Output: A matrix W^{inter} where rows correspond to processors and columns correspond to communities ($\kappa_{CM_i, v \in \mathcal{V}^p}$)

```
begin
    for $l$ in $L$: do
        for $p$ in $\mathcal{V}^p$: do
            for $CM$ in $\{CM_1, CM_2, ..., CM_L\}$: do
                /* Extract all computation nodes in the community (i.e. if a
                   communication node has some computation nodes, add them to the list) */
                CompNodeList = ExtractCompNodes($C_i \in CM_i \forall i \in |CM_L|$)
                if CompNodeList is empty then
                    /* This community doesn't host any computational node; Unfeasible */
                    Skip Community
                end
                /* Filter the list of computational nodes to keep only nodes having enough
                   capacity (CPU & RAM) to host the processor */
                filteredCommunityNodes = FilterCompNodes($p$, CompNodeList)
                if filteredCommunityNodes is empty then
                    /* The community cannot host this processor */
                    Skip the community
                end
                /* Create a list of tuples where each corresponds to a feasible
                   computation node */
                $M_1 = [\ (\frac{d_{in}}{\lambda_{in}}, \frac{d_{out}}{\lambda_{out}})$ for each CompNode in filteredCommunityNodes]
                /* for each element in the list, extract the maximum value out of the
                   tuple */
                $M_2 = [\ \max(a, b)$ for $(a, b)$ in $M1\ ]$
                /* Compute $CompScore$ value as the minimum out of the elements in the
                   obtained list; Idx corresponds to the index of the computational node
                   having the best score */
                $CompScore$, Idx = $\min(M_2)$
                /* Extract costs of the computation node having $CompScore$ */
                $w_{in} = \text{argmin}_{Idx}(w_{ij})_{in}$ $w_{out} = \text{argmin}_{Idx}(w_{ij})_{out}$
                /* Compute $\kappa_{p,CM_i}$ the score of $p$ on being hosted in $CM_i$ (higher is
                   better) */
                $\kappa = \frac{1}{CompScore \times (w_{in} + w_{out})}$
                /* Update rows and columns of $W^{inter}$ */
            end
            /* Check whether we found at least a "feasible" community in this level for $p$
               */
            if No feasible communities were found for p then
                SkipToNextLevel()
            end
        end
        /* Check whether all processors have at least a feasible community in that level
           $L$ */
        if At least a feasible community then
            /* We have found a level where each processor is feasible to at least one
               community */
            break
        end
    end
    return $W^{inter}$
end
```

8.4.5 Initial Placement & Routing

We now enter **Phase 4**: initial placement and routing. Up to this point the heuristic has produced:

- A hierarchical clustering of the cloud network graph from Phase 1
- A processor ranking from Phase 2
- An **inter-community score mapping** W^{inter} whose entry $\kappa_{p,CM}$ indicates how attractive community CM is for processor p

What W^{inter} lacks is any hint about which individual computation node inside **a chosen community** should host the processor. Therefore, this phase proceeds in the classic two-step fashion familiar from VNE work: node placement first, link routing second, but introduces an additional data structure for the intra-community choice.

W^{inter} is a dictionary whose keys are processors $p \in \mathcal{V}^p$. Each value is a **rectangular** table: the *rows* correspond to feasible communities for p and the *columns* to the computation nodes contained in that community (column count varies with community size).
We initialize every row with a **uniform probability** distribution. For example, if processor p can be hosted in community CM_2 and CM_2 contains four computation nodes, the corresponding row starts as [0.25,0.25,0.25,0.25].
These probabilities will be updated iteratively in Phase 5, the uniform start merely states that before any cost-load evidence is examined, all nodes in a feasible community are equally likely hosts.
Figure 8.7 extends the high-level schematic of Figure 8.2, detailing where W^{inter} and W^{intra} are produced and how they drive the subsequent routing step.

The process of initial placement works as follows: We iterate first over ranked list of processors (higher score, higher importance), for each processor we select the cluster with the highest probability $P_{p,CM} = \max(W^{inter}(p))$, then we select a computation node with the highest probability $P_{p,CM,n} = \max(W^{intra}(p, CM))$ (initially probabilities are uniform so selection is random). We perform the placement of each processor following this approach. Then, we route commodities (from sources - users - to processors, and from processors to destinations). This step involves searching for the shortest path out of a set of feasible paths that maximizes a scoring function, as described in Algorithm 2. Finally, this placement (routing mappings) is returned along with the score. Please note that during the routing stage, we may encounter that there exists no feasible path from a source to a destination, which means that the placement is unfeasible and thus, we repeat the placement. Since any route has either a processor as source, or a processor as destination, or both. We set the probability of all processors (forming this link) to zero for this specific placement.

8.4.6 Probabilistic Iterative Refinement

In the final phase of our heuristic, we perform a short, guided search that nudges the solution toward the cost-load spot without incurring the MILPs heavy runtimes. The algorithm maintains two probability tables:

Algorithm 2 Initial Placement & Routing

```
Input: List of Ranked Processors $\mathcal{V}^p$, $W^{inter}$, $W^{intra}$
Output: Mapping $M$ of each service graph commodity over a path in the cloud network graph
begin
    // Stage 1: Placement
    for $p$ in RankedProcessors do
        /* Select a community based on probability distribution $W^{inter}(p)$ */
        CM = SelectCommunity(p, $W^{inter}$)
        /* Select a computation node based on probability distribution in $W^{intra}(p)$ */
        CompNode = SelectCompNode($p$, $W^{intra}$, $CM$)
        while CompNode is NOT feasible do
            /* Capacity constraints are violated, set the probability of this node to
               zero, re-normalize probabilities, and select another compNode */
            $W^{intra}(p, CM, compNode) = 0$
            Normalize($W^{\text{intra}}(p, CM)$)
            CompNode = SelectCompNode($p$, $W^{intra}$, $CM$)
        end
    end
    // Stage 2: Routing
    for $k \in \mathcal{K}$ do
        /* for each commodity in the service graph, we extract source and destination
           endpoints */
        src , dest = ExtractEndPoints($k$)
        /* Generate 3 shortest paths from src to dest based on the cost $w_{(src,dest)}$ */
        paths = GetSP(src,dest, weighting="cost")
        /* Now we have 3 possible paths, we try to embed the commodity on one of them, if
           at least one is feasible we continue, otherwise we return a failure signal */
        for path $\in$ paths do
            isEmbedded = Embed($k$, path) if isEmbedded then
                /* Found a feasible routing; save it and move to the next commodity */
                Routes[$k$] = path continue
            end
        end
        /* We check whether a path is found for $k$ */
        if Routes[k] is None then
            /* No feasible path found; failed routing; we stop the routing loop and
               select a new placement; the function will take care of setting the
               probability of the selected computation nodes to zero */
            ReinitializePlacement(src, dest)
        end
    end
    /* We iterated over all commodities and routed them, we return the mapping and the
       score */
    $M$ = Routes
    /* Scoring of this P&R is equal to the a constant times the standard variation of
       loads (over edges) times the maximum loaded link ratio */
    score = $\frac{1}{2} \times \sigma(L_i) \times L_{max}$
    return $M$, score
end
```

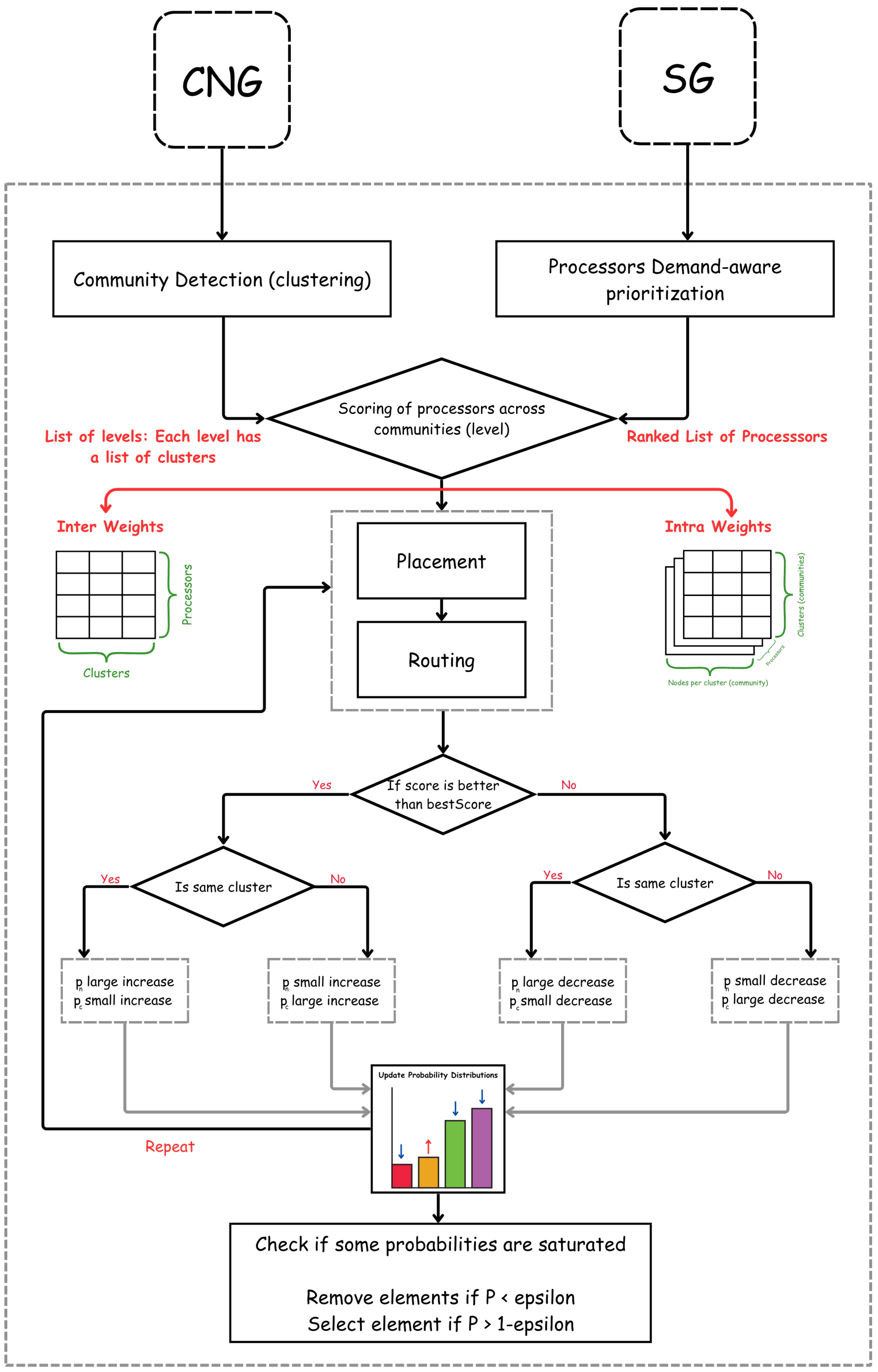

CNG
SG
Community Detection (clustering)
Processors Demand-aware prioritization
Scoring of processors across communities (level)
List of levels: Each level has a list of clusters
Ranked List of Processsors
Inter Weights
Processors
Clusters
Intra Weights
Clusters (communities)
Processors
Nodes per cluster (community)
Placement
Routing
If score is better than bestScore
Yes
No
Is same cluster
Yes
No
p_n large increase
p_c small increase
p_n small increase
p_c large increase
Is same cluster
Yes
No
p_n large decrease
p_c small decrease
p_n small decrease
p_c large decrease
Update Probability Distributions
Repeat
Check if some probabilities are saturated
Remove elements if P < epsilon
Select element if P > 1-epsilon

Figure 8.7. A detailed flowchart of the P&R Heuristic algorithm

- W^{inter} likelihood of moving a processor p to another community.
- W^{intra} likelihood of choosing a *different* compute node inside that community.

Both tables are **updated** multiplicatively, probabilities associated with improvements are scaled up by a factor $\beta > 1$, while those linked to degradations are scaled down by $\gamma < 1$. After each update, the affected row is re-normalized so that it remains a valid distribution. The full approach is explained in Figure 8.7. The process works as follows: We have the mapping obtained by the initial P&R from phase 4, based on this, we select randomly a processor to move it, the new position of this node will be based on the probability distributions in W^{inter} and W^{intra}. This means that the higher the probability, the more likely to select the same node; thus, these probabilities are adjusted based on the performance of the new placement. If the placement improves the solution (reduces the score), the probabilities are increased (both for the cluster and the compute node), and decreased otherwise.

This process is repeated for a specific number of iterations T_{MAX}, while at the end of each iteration, we check whether some probabilities are close to 1 (within some tolerance ϵ) so that we no longer explore other placements for that specific processor. The detailed pseudocode of this phase is provided in Algorithm 3.

8.5 Chapter Summary

This chapter moved the thesis from simulator mechanics to concrete orchestration policy design. First, we cast the processor-placement problem as a generic MILP that faithfully preserves the multicast, multi-resource and end-to-end-latency constraints of Musical-Metaverse workloads. We then specialised that template into two objective-driven models:

- Min-Cost MILP: an economic baseline that yields the cheapest feasible embedding
- Load-Balancing MILP: a congestion-aware variant that flattens worst-link utilization via an epigraph linearization.

Although optimal, both models scale poorly when the audience size and network reach grow. To bridge this gap, we introduced a heuristic topology-cost-load aware algorithm whose five phases are: Community detection, demand-aware ranking, community scoring, initial placement and routing, and probabilistic iterative refinement. It provides sub-second decision while keeping cost, and load, clost to the MILP frontiers (more details in the next chapter).

The next chapter leverages **SiMusMet** to benchmark all three approaches on realistic cloud topologies and audience traces, quantifying the trade-offs in monetary cost, latency, computational overhead, and thereby validating the practical value of the heuristic against its optimal counterparts.

Algorithm 3 Iterative Refinement

```
Input: Initial mapping $M$, initial score $score_0$ $W^{inter}$, $W^{intra}$
Output: Mapping $M$ of each service graph commodity over a path in the cloud network graph
begin
    $bestScore = score_0$ $bestMapping = M$
    for $iteration$ in $T_{MAX}$ do
        /* Select a random processor */
        $p$ = SelectProcessor() /* Select a community */
        $CM$ = SelectCommunity($p$) /* Select a compNode */
        $compNode$ = SelectCompNode($p$, $CM$)
        /* Perform new placement and routing - like in Algorithm 2 */
        $M_{new}, score_{new}$ = PlaceAndRoute($p$, $compNode$, $bestMapping$)
        /* We evaluate the result obtained from the new embedding attempt */
        if $score_{new} < bestScore$ then
            // *IMPROVEMENT*
            /* We check whether the new computation node belongs to the same cluster or
               not */
            if Same community then
                /* small increase in cluster probability, large increase in node
                   probability */
                $P_{p,CM,n}$ += DeductFromOthersAndAdd($p$,$CM$,0.15, $W^{intra}$)
                $P_{p,CM}$ += DeductFromOthersAndAdd($p$,$CM$,0.05, $W^{inter}$)
            end
            else
                /* Different communities; Large increase in cluster prob, small increase
                   in comp node prob */
                $P_{p,CM,n}$ += DeductFromOthersAndAdd($p$,$CM$,0.05, $W^{intra}$)
                $P_{p,CM}$ += DeductFromOthersAndAdd($p$,$CM$,0.15, $W^{inter}$)
            end
            /* Set current mapping as best */
            $bestMapping = M_{new}$; $bestScore = score_{new}$
        end
        else
            // *DEGRADATION*
            /* We check whether the new computation node belongs to the same cluster or
               not */
            if Same community then
                /* small decrease in cluster probability, large decrease in node
                   probability */
                $P_{p,CM,n}$ -= AddFromOthersAndDeduct($p$,$CM$,0.15, $W^{intra}$)
                $P_{p,CM}$ -= AddFromOthersAndDeduct($p$,$CM$,0.05, $W^{inter}$)
            end
            else
                /* Different communities; Large decrease in cluster prob, small decrease
                   in comp node prob */
                $P_{p,CM,n}$ -= AddFromOthersAndDeduct($p$,$CM$,0.05, $W^{intra}$)
                $P_{p,CM}$ -= AddFromOthersAndDeduct($p$,$CM$,0.15, $W^{inter}$)
            end
        end
        /* Normalize probabilities */
        Normalize($W^{inter}$, $p$, $CM$)
        Normalize($W^{intra}$, $p$, $CM$)
        /* End of current iteration; check if some probabilities in $W^{intra}$ or $W^{inter}$
           are close to 1 within a tolerance $\epsilon = 0.001$ so that this selection becomes
           deterministic: The corresponding processor is not explored anymore */
        CheckProbabilities($W^{intra}$ ,$W^{inter}$)
    end
    return $bestMapping$, $bestScore$
end
```

Chapter 9

Simulation Results and Analysis

This chapter puts the theoretical work of Chapters 6-8 to the test. Using **SiMusMet** we run a broad campaign of simulations that compares the two optimal MILP formulations: **Min-Cost** and **Load-Balancing (LB)** against the proposed Topology-Aware Cost–Load heuristic. Our objectives are the following:

- **Quantify performance** in terms of monetary cost, worst-link utilization, and Quality-of-Experience (QoE) for musical streams.
- **Measure computational effort** (solver runtime, memory footprint) across small, medium, and large CNGs, thereby exposing the scalability limits of exact optimization.
- **Identify trade-offs** between cost efficiency and load fairness, and assess how closely the heuristic approaches the MILP frontiers while meeting real-time decision budgets.

9.1 Experimental Setup

9.1.1 Datasets: Cloud network graphs and service graphs

The evaluation uses **seven** synthetic cloud-network graphs (CNGs) and **nineteen** service graphs (SGs).Table 9.1 summarises the CNG parameters. Each graph is an Erdős–Rényi topology $G(n, \rho)$ with edge-existence probability $\rho = 0.7$.
We generate one instance for every node count $n \in \{4,6,8,10,12,14,16\}$, so the dataset spans small to moderately large footprints. Link capacities and costs are drawn from a uniform distribution $\mathcal{U}(a, b)$, whose bounds are reported in the table.
The *computation-node percentage* is the per-communication-node probability of hosting at least one computation node; this value is set to 0.7 to reflect heterogeneous edge–core deployments.

Similarly, Table 9.2 reports the parameters used to generate the twenty service graphs. Each SG follows the template of Figure 6.2; the only variable is the audience size.

Parameter	**Range of values**
Communication Links capacity	$\mathcal{U}(1000,3000)$
Communication Links cost	$\mathcal{U}(10,30)$
Computation IN capacity (RAM)	$\mathcal{U}(1000,3000)$
Computation IN cost	$\mathcal{U}(10,25)$
Computation OUT capacity (CPU)	$\mathcal{U}(1000,2000)$
Computation OUT cost	$\mathcal{U}(10,25)$
Computation Percentage	0.7
Number pf computation nodes	$\mathcal{U}(1,3)$
Density	0.7

Table 9.1. Configuration Setup for Cloud Network Topologies

Whereas the figure depicts the case ($|U| = 2$), we instantiate the template for every $u \in \{2,3,...,20\}$ thus covering small to densely populated concerts.[1]

Edge type	**Pattern (src → dst)**	$\boldsymbol{R}_{\text{prod}}$	$\boldsymbol{R}_{\text{comm}}$	$\boldsymbol{R}_{\text{cons}}$
Source				
	_ → VEControls	$\mathcal{U}(3,10)$	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$
	_ → AvatarSynch	$\mathcal{U}(50,70)$	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$
	_ → AudienceMix	$\mathcal{U}(80,120)$	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$
	$\text{MMPred}_{\text{gest}}$ → AvatarSynch	$\mathcal{U}(50,70)$	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$
	$\text{MMPred}_{\text{sound}}$ → AudienceMix	$\mathcal{U}(80,120)$	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$
Destination				
	VEControls → _	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$	$\mathcal{U}(3,10)$
	AvatarSynch → _	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$	$\mathcal{U}(50,70)$
	AudienceMix → _	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$	$\mathcal{U}(120,200)$
	Streaming → _	$\mathcal{U}(10,50)$	$\mathcal{U}(10,50)$	$\mathcal{U}(120,200)$
proc-proc				
	MMPred → AvatarSynch	$\mathcal{U}(50,70)$	$\mathcal{U}(30,70)$	$\mathcal{U}(50,70)$
	MMPred → AudienceMix	$\mathcal{U}(80,120)$	$\mathcal{U}(30,40)$	$\mathcal{U}(80,120)$
	LatComp → Streaming	$\mathcal{U}(3,10)$	$\mathcal{U}(3,10)$	$\mathcal{U}(3,10)$
	LatComp → MMPred	$\mathcal{U}(3,10)$	$\mathcal{U}(3,10)$	$\mathcal{U}(3,10)$

Table 9.2. Range of production, communication and consumption rates assigned to each synthetic service-graph edge.

[1]Flows having same information ID, carry the same amount of data and therefore, have the same amount of requirements

9.1.2 Hardware and Solver Configuration

All simulations were executed on the same workstation to keep runtime measurements comparable. The key specifications are:

- **CPU**: Intel Core i7-13620H (13th Gen, 10 cores, 2.4 GHz base).
- **Memory**: 16 GB DDR4 @3200 MHz.
- **Operating system**: Windows 11 Home 24H2.
- **Python environment**: Python 3.11.8, NetworkX 3.4.2, NumPy 2.2.6
- **MILP solver**: Gurobi 12.0.2, with academic license.
- Random seed fixed to 42 for all operations (python RNG, Numpy, GurobiPy)

9.1.3 Metrics

In this section, we will introduce the set of metrics that will be presented in the upcoming experiments.

Cost

The cost (or monetary cost) is the sum of cost per edge times the amount of resources used as described in Equation 8.13.

Load

More precisely, maximum load is the amount of maximally loaded edges over the network. The load of an edge is computed as $L_{ij} = \frac{u_{ij}}{c_{ij}}$.

Load Degree (Standard Deviation of Load)

This metric measures the variation in load across all edges in the network. The standard deviation gives insight into how **unevenly** the traffic or resource demand is distributed.

- A high standard deviation indicates that some edges are heavily loaded while others are lightly used a sign of imbalance.
- A low standard deviation means that the load is evenly spread across the network, which is typically desirable in distributed systems.

Fairness (Jain's Fairness Index of Load)

Jain's Index is used to quantify how fairly the load is distributed among the edges. It is defined as:

$$JI = \frac{(\sum_{i=1}^{n} L_i)^2}{n \cdot \sum_{i=1}^{n} L_i^2} \tag{9.1}$$

where L_i is the load on edge i, and n is the number of edges.

- A value close to 1 indicates perfect fairness (equal load across edges).
- A value close to 0 means that the load is highly skewed toward a few edges.

This metric **complements** the standard deviation by offering a normalized view of fairness.

Resource Utilization

This metric reflects the percentage of total network resources actually used during execution. It can be calculated separately for different resource types (e.g., CPU, bandwidth, memory), and is generally defined as:

$$ResUtil = \frac{UsedResources}{TotalAvailableResources} \tag{9.2}$$

CPU Time

CPU Time measures the computational effort or processing time required to complete the embedding or allocation process. This is an indicator of algorithmic efficiency and scalability. A lower CPU time is desirable for real-time or large-scale systems, as it enables faster decision-making and deployment.

9.2 Results and Discussion

This section presents and analyzes the results obtained from the experimental testbed conducted on seven different cloud network graphs (CNGs). In the first part, we examine the results for each CNG individually[2]. In the second part, we analyze the overall outcomes to draw general conclusions regarding the performance and effectiveness of our proposed heuristic.

The results shown in the figures for each experiment represent the average outcomes obtained by repeating the experiment 100 times with randomly varying configurations (including user locations, resources, and requirements).

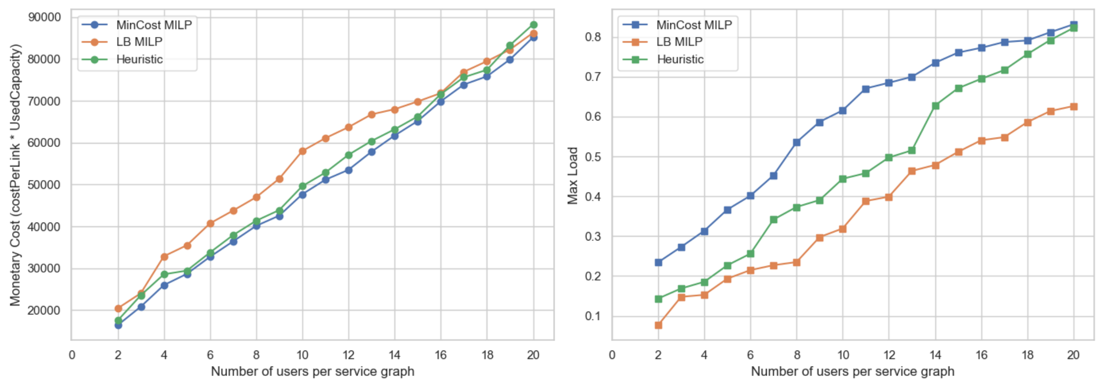


Figure 9.1. Cost-load trade-off for the 4-node Cloud Network Graph (CNG). **Left:** total monetary cost incurred by each placement strategy as the number of users per service graph increases. **Right:** corresponding maximum link-utilisation (*Max Load*).

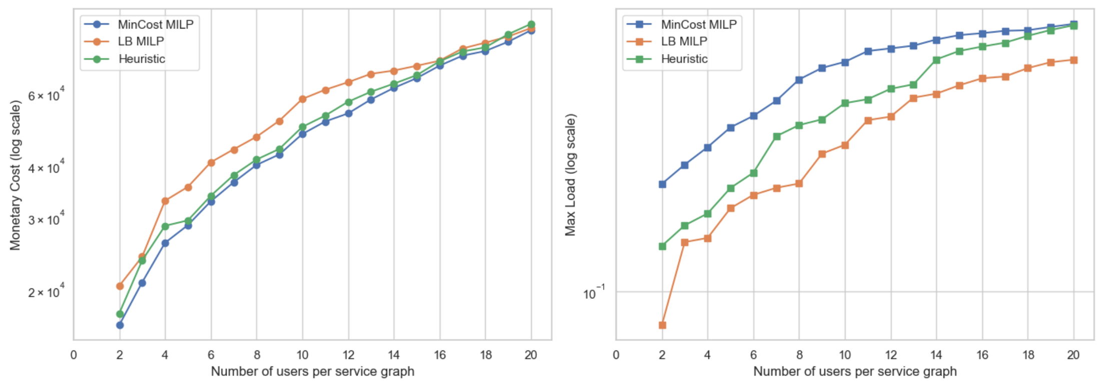


Figure 9.2. Cost-load **log scale** trade-off for the 4-node Cloud Network Graph (CNG)

9.2.1 Experiment 1: Performance Evaluation on CNG with 4 Nodes

Figure 9.1 summarizes the behavior of the three placement strategies *MinCost MILP*, *LB MILP*, and the proposed *Heuristic*, as the number of users per service graph grows from 2 to 20.

The left-hand panel reports the total monetary cost, while the right-hand panel shows the maximum link-utilization (*Max Load*) observed in the network.

[2]Results for 4 experiments out of 7 will be discussed in detail. For the remaining experiments, results and figures are reported in the Annex

MinCost MILP is, as expected, the least-cost solution across the entire demand range. Its advantage is most pronounced for small user populations (15k monetary units at 2 users versus vs 20k for LB MILP), but the gap narrows as the workload intensifies (above 16 users). The LB MILP, which is optimized for load balancing rather than cost, is consistently the most expensive, roughly 10–15 % higher than MinCost for most loads. The Heuristic closely tracks MinCost at low and medium loads and remains within 5 to 8 % of MinCost even at the highest demand levels, indicating that it preserves most of the cost efficiency of the exact optimization.

A complementary picture emerges for Max Load. The LB MILP keeps the peak link utilization well below 0.6 (maximum utilized link has around 40% free space) over the full demand range, demonstrating its effectiveness in spreading traffic. Min Cost MILP, by contrast, drives one or more links to saturation: utilization rises steeply beyond ten users and approaches 0.83 at 20 users. The Heuristic again occupies the middle ground. It begins close to LB at light load, but from 14 users onward, its utilization jumps to around 0.75-0.80, substantially lower than MinCost, yet higher than the LB target.

#	**MinCost**	**LB**	**Heuristic**	**MinCost/Hr**	**LB/Hr**
2	**0.042**	0.125	0.098	-	x1.28
3	**0.060**	0.083	0.128	-	x0.65
4	**0.125**	0.274	0.177	-	x1.55
5	**0.092**	0.094	0.139	-	-
6	**0.119**	0.171	0.168	-	x1.02
7	0.146	0.280	**0.119**	x1.22	x2.35
8	0.183	0.330	**0.161**	x1.14	x2.05
9	0.206	0.236	**0.159**	x1.30	x1.49
10	0.295	0.413	**0.214**	x1.38	x1.93
11	0.312	0.373	**0.276**	x1.13	x1.35
12	0.333	0.463	**0.255**	x1.31	x1.82
13	0.488	0.498	**0.348**	x1.40	x1.43
14	0.470	0.631	**0.321**	x1.46	x1.97
15	0.611	0.570	**0.207**	x2.95	x2.75
16	0.476	0.775	**0.314**	x1.52	x2.47
17	0.624	0.817	**0.328**	x1.90	x2.49
18	0.752	1.006	**0.335**	x2.25	x3.01
19	0.877	1.015	**0.649**	x1.35	x1.56
20	0.954	1.226	**0.671**	x1.42	x1.83
Average	0.377	0.494	**0.267**	x1.42	x1.79

Table 9.3. CPU runtime comparison for the 4-node CNG. (in ms)

The execution times reported in Table 9.3 highlight how the three placement methods scale on the 4-node CNG.

For the smallest workloads (service graphs with **2 − 6 users**) the **MinCost MILP** is the quickest, requiring roughly **40 − 70** % of the **heuristic's time** (see MinCost/Hr < 1). As the user population grows, however, its branch-and-bound tree explodes: beginning with the 7-user instance, its relative cost crosses the unity line, and by **15 users** it is almost three times slower than the heuristic. Beyond that point, the gap never closes again.

The proposed heuristic therefore becomes the fastest option from 7 users onward, while still remaining competitive in the small-case regime (no instance shows more than a 2.4× slowdown w.r.t. MinCost). Averaged over all 19 instances it delivers a 29 % speed-up over MinCost and a 46 % speed-up over the load-balancing MILP (LB).

The LB MILP is consistently the most time-consuming approach. Even for the lightest load it is three times slower than MinCost, and at moderate-to-high loads it trails the heuristic by factors between 1.3 and 3.0. This confirms that the richer constraint set required for explicit load balancing incurs a non-negligible computational penalty.

In summary, the heuristic offers the best runtime once the traffic reaches realistic levels, while MinCost remains a viable choice only for minimal demand scenarios where its optimality can be enjoyed at negligible extra cost.

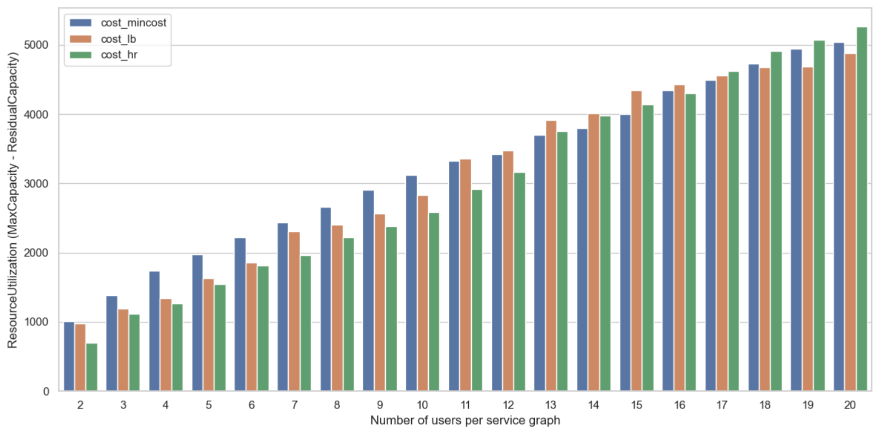


Figure 9.3. Resource Utilization comparison for the 4-node CNG

The bar graph in Figure 9.3 shows the aggregate utilization of resources defined as the total capacity activated across all links with respect to increasing user populations in the 4-node CNG. Two main patterns emerge:

- Light to Medium loads (2 − 10 users): The MinCost-MILP consumes the most capacity, between 10 % and 25 % more than LB-MILP, because its objective spreads service components over a smaller set of resources. The Heuristic, records the smallest footprint (around 15 % below both MILPs).
- Heavy load (≥ 14 users): From 14 to 20 users the three curves climb almost linearly, yet their ordering fluctuates: LB-MILP is the most resource-intensive in 6 of the

final 10 instances. LB remains in the middle, about 5%–10% below MinCost, than becomes the most-efficient at the final experiments (18,19 and 20). The Heuristic retains the lowest or second-lowest footprint in every case except the final three, and is on average 7%–12% less expensive than LB-MILP over the range.

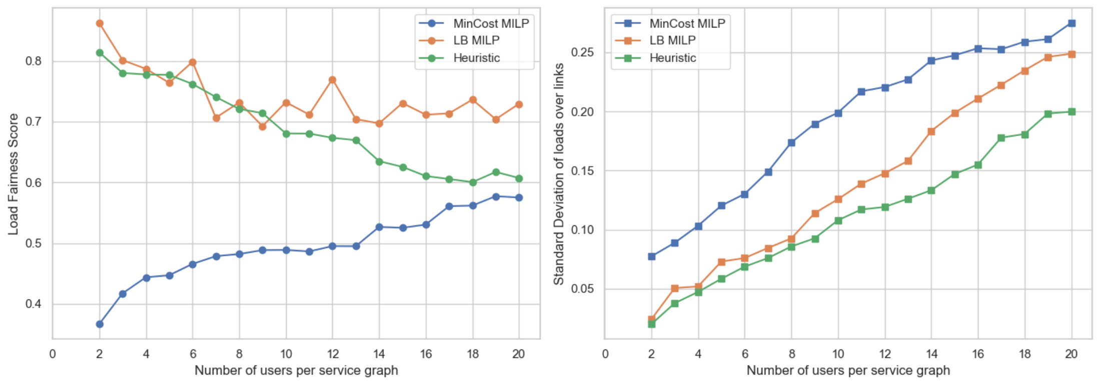


Figure 9.4. Fairness and Standard Deviation of load comparison for the 4-node CNG

The last pair of plots in Figure 9.4 assesses how evenly traffic is distributed in the 4-node CNG, using two complementary metrics:

- Load-fairness score (left): the closer to 1, the more equal the link loads (Jain index).
- Standard deviation of link loads (right): the lower, the more uniform the utilization.

Load-fairness (left panel)

- LB MILP achieves the highest fairness across the entire demand range, rarely falling below 0.70 and peaking around 0.90. This is consistent with its objective of explicit load balancing.
- MinCost MILP exhibits the poorest fairness. In the 2-to-12-user regime, its score ranges steadily up to 0.50, indicating that one or two links carry a disproportionate share of the traffic. Fairness improves slightly beyond 12 users as the model is forced to activate additional paths, but it never approaches the LB curve.
- The heuristic starts close to LB for 2 users (≈ 0.80) but declines steadily as demand increases, falling around 0.77 for 11–13 users before recovering to ≈ 0.60 for 20 users. In other words, it sacrifices some equity compared with LB yet remains markedly more fair than MinCost for most loads.

Standard deviation of link loads (right panel)

- Heuristic is the clear winner here. Up to 15 users its σ values remain below 0.12, roughly one-third of the LB values and one-half of the MinCost values; only at the very largest load (20 users) it approaches 0.20.
- LB MILP keeps the variability moderate (≤ 0.24) but cannot match the Heuristic because the MILP still allows some links to become noticeably busier than others when it must respect capacity constraints.
- MinCost MILP shows the largest dispersion, climbing from ≈ 0.07 at 2 users to $>$ 0.26 at 20 users. Its cost-driven path selection repeatedly pushes traffic onto the cheapest links, amplifying imbalance as the system fills.

9.2.2 Experiment 2: Performance Evaluation on CNG with 8 Nodes

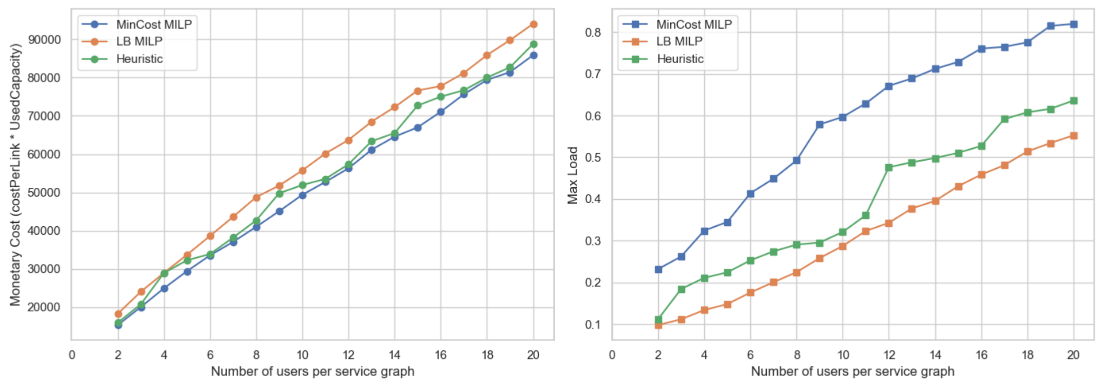


Figure 9.5. Cost-load trade-off for the 8-node Cloud Network Graph (CNG).

The monetary-cost panel shows the same qualitative trend as in CNG-4, but the numerical gaps are wider:

- **Linear Growth** MinCost MILP and Heuristic remain within 5% of each other, while LB-MILP is consistently 10 to 15% more expensive because it sacrifices cheap links to favor load spread.

The Max-Load panel reveals why: MinCost again saturates a single link (≥ 0.67 from 12 users on, up to 0.82 for 20 users), whereas LB caps the peak utilisation at ≈ 0.55. The Heuristic limits its worst-case load to ≈ 0.63 at the highest demand, providing a balanced compromise between cost and congestion.

On average, the Heuristic is **1.65**$\times$ faster than MinCost and **7**$\times$ faster than LB. MinCost overtakes the heuristic only in the smallest two instances; from 4 users onward, the MILP runtimes grow super-linearly, while the heuristic remains nearly linear. The

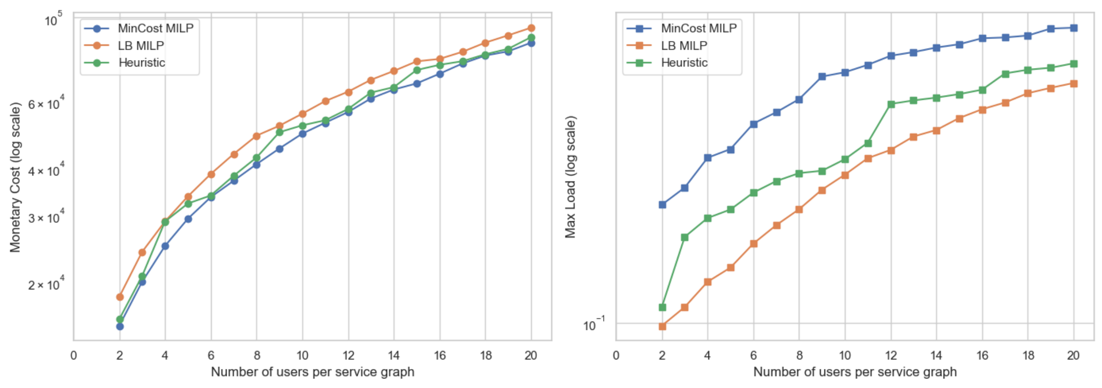


Figure 9.6. Cost-load **log scale** trade-off for the 8-node Cloud Network Graph (CNG)

#	MinCost	LB	Heuristic	MinCost/Hr	LB/Hr
2	0.055	0.458	**0.043**	x1.28	x10.56
3	0.079	**0.606**	0.079	x1.00	-
4	0.111	0.897	**0.033**	x3.40	x27.47
5	0.167	0.503	**0.089**	x1.87	x5.63
6	0.269	0.634	**0.134**	x2.01	x4.72
7	0.363	1.356	**0.300**	x1.21	x4.52
8	0.327	0.765	**0.264**	x1.24	x2.90
9	0.616	3.024	**0.227**	x2.72	x13.34
10	0.535	2.745	**0.298**	x1.80	x9.21
11	0.706	1.289	**0.366**	x1.93	x3.52
12	0.753	4.875	**0.378**	x1.99	x12.91
13	0.747	1.690	**0.350**	x2.14	x4.84
14	0.817	1.793	**0.358**	x2.28	x5.01
15	1.181	3.896	**1.162**	x1.02	x3.35
16	1.130	2.406	**1.019**	x1.11	x2.36
17	1.378	3.047	**1.289**	x1.07	x2.36
18	1.498	4.856	**1.327**	x1.13	x3.66
19	1.381	5.074	**1.306**	x1.06	x3.88
20	1.767	7.954	**1.593**	x1.11	x4.99
Average	0.731	2.519	**0.559**	x1.65	x6.99

Table 9.4. CPU-runtime comparison for the 6-node CNG.

LB-MILP is an order of magnitude slower for many instances due to its larger constraint set.

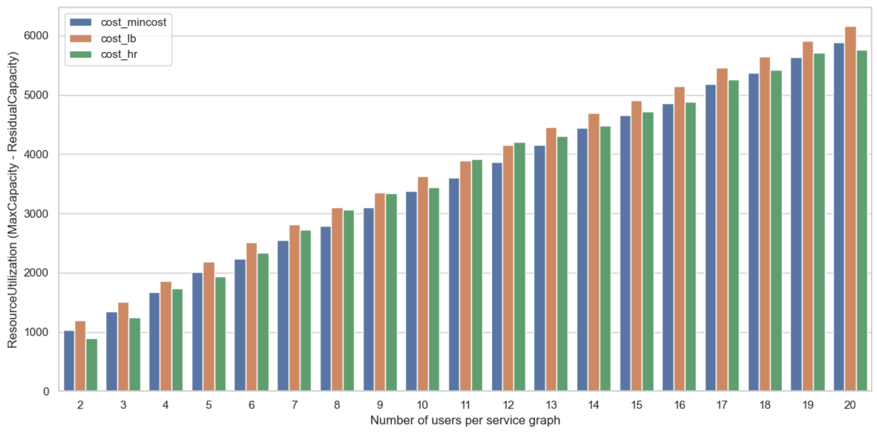


Figure 9.7. Resource Utilization comparison for the 8-node CNG

LB again activates the most capacity (≈ 20% above MinCost), with the heuristic sitting in between. From 16 users on, MinCost becomes the largest second consumer as its cheapest links fill up, and by 20 users it reaches 5900 units versus 6100 units for LB and 5700 units for the heuristic. Overall, the heuristic conserves 8–15% more capacity than MinCost in the high-load regime while never exceeding LB.

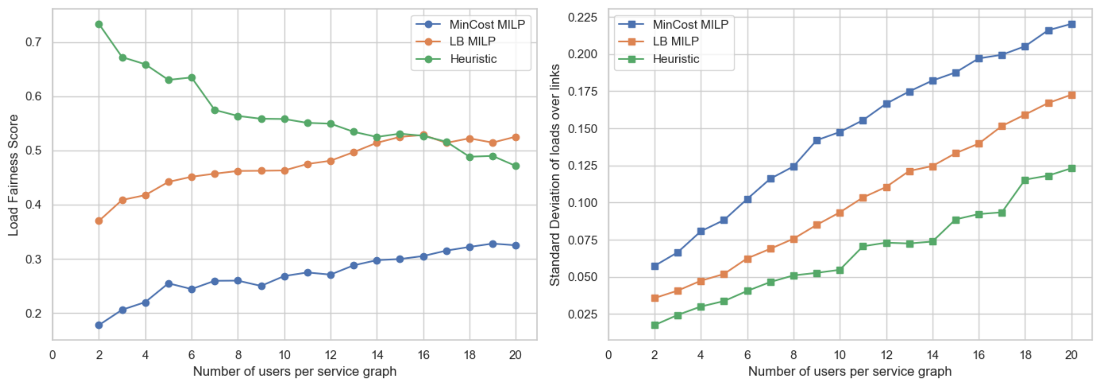


Figure 9.8. Fairness and Standard Deviation of load comparison for the 8-node CNG

- Fairness: The heuristic starts highest (0.72) but gradually declines to ≈ 0.46; LB is more erratic but overtakes the heuristic at 16 users and ends around 0.52; MinCost never exceeds 0.32.
- Dispersion: The heuristic keeps the standard deviation below 0.125 for all users, whereas MinCost rises to 0.225 and LB to 0.175.

Thus, even though LB scores slightly better on the fairness metric beyond midload, the heuristic still produces the most tightly clustered link loads, reflecting its modest but consistent balancing ability.

9.2.3 Experiment 3: Performance Evaluation on CNG with 12 Nodes

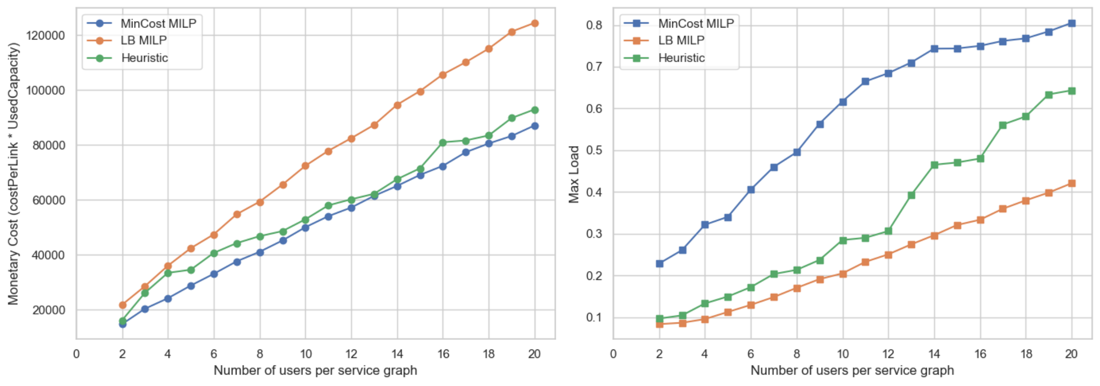


Figure 9.9. Cost-load trade-off for the 12-node Cloud Network Graph (CNG).

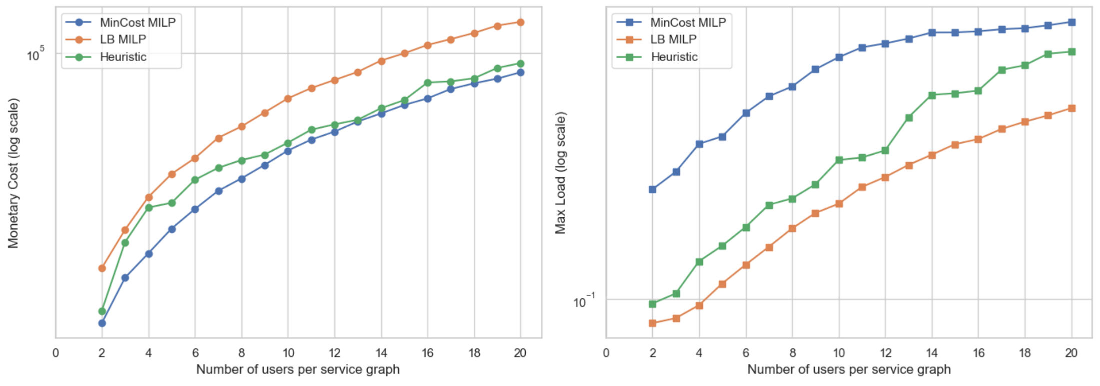


Figure 9.10. Cost-load **log scale** trade-off for the 12-node Cloud Network Graph (CNG)

From 2 to 10 users the heuristic never exceeds 10% of the optimal solution. Starting at 12 users the LB-MILP becomes markedly costlier, ending more than 40-50% above MinCost at 20 users. The heuristic tracks MinCost closely (within ±6 % for every instance) while staying well below LB for high loads.

As in earlier experiments, MinCost rapidly saturates one link (≈ 0.8 by 14 users). LB constrains the peak utilization below 0.45. The heuristic follows LB up to 11 users, then rises to ≈ 0.65 at 20 users, again striking a middle ground between cost efficiency and congestion control.

#	**MinCost**	**LB**	**Heuristic**	**MinCost/Hr**	**LB/Hr**
2	**0.082**	4.236	0.093	-	x45.38
3	0.136	2.237	**0.064**	x2.11	x34.80
4	0.163	5.260	**0.037**	x4.46	x144.07
5	0.279	21.683	**0.014**	x19.42	x1503.23
6	0.406	11.647	**0.090**	x4.50	x129.33
7	0.462	29.668	**0.091**	x5.07	x325.29
8	0.956	14.740	**0.070**	x13.70	x210.99
9	1.313	24.237	**0.056**	x23.49	x431.16
10	1.520	12.444	**0.082**	x18.44	x152.02
11	1.573	24.765	**0.078**	x20.14	x318.43
12	1.585	31.520	**0.912**	x1.74	x34.57
13	1.261	53.958	**0.719**	x1.75	x75.07
14	1.853	48.025	**0.088**	x21.06	x547.23
15	1.795	105.009	**0.894**	x2.01	x117.52
16	1.798	32.007	**0.875**	x2.06	x36.57
17	2.534	23.182	**0.935**	x2.71	x24.78
18	2.716	45.610	**1.002**	x2.71	x45.46
19	4.745	50.449	**1.042**	x4.56	x48.45
20	4.457	115.188	**1.039**	x4.29	x110.86
Average	1.560	34.519	**0.431**	x8.14	x227.87

Table 9.5. CPU-runtime comparison for the 8-node CNG.

The heuristic is on average **eight times faster** than MinCost and more than **two hundred times** faster than LB. MinCost only beats the heuristic at the smallest instance; starting with 3 users the MILP search tree explodes, and LB becomes prohibitive (4 s at 20 users vs. ≈ 1 s for the heuristic).

LB consistently activates the most capacity. MinCost remains the algorithm that provides lowest resource utilization. The heuristic ends at 6600 units, a saving of ≈ 22% overall against MILP-LB.

- Fairness: The heuristic starts at 0.63 and degrades slowly to 0.46; LB fluctuates between 0.35 – 0.45; MinCost never exceeds 0.25.

- Standard deviation: The heuristic keeps $\sigma < 0.07$ up to 11 users, then increases but stays below LB and well below MinCost (which peaks near 0.20).

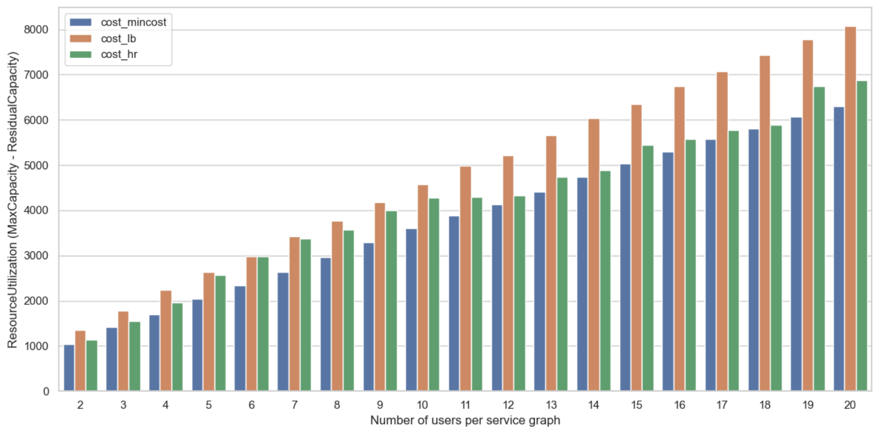


Figure 9.11. Resource Utilization comparison for the 12-node CNG

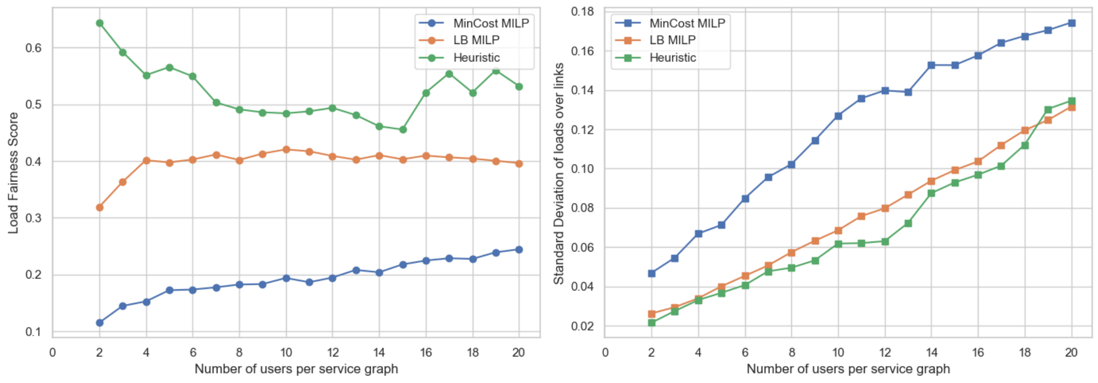


Figure 9.12. Fairness and Standard Deviation of load comparison for the 12-node CNG

9.2.4 Experiment 4: Performance Evaluation on CNG with 16 Nodes

- Cost: The Heuristic stays within $\approx 7\%$ of the MinCost curve for the entire demand range while remaining $10 - 15\%$ cheaper than LB in every high-load instance (≥ 12 users). Hence, the marginal monetary premium of the Heuristic over MinCost is minor, yet it already yields appreciable savings against the load-balancing MILP.

- Load: Relative to the Heuristic, MinCost drives peak utilization up by a factor 1.4-to-1.8 once the traffic exceeds 12 users, hitting 0.75–0.80 on several occasions. LB keeps the peak 35 – 45 % below the Heuristic, but, as shown above, at a noticeable cost surplus. In other words, the Heuristic achieves roughly two-thirds of LB's congestion relief while preserving near-optimal cost.

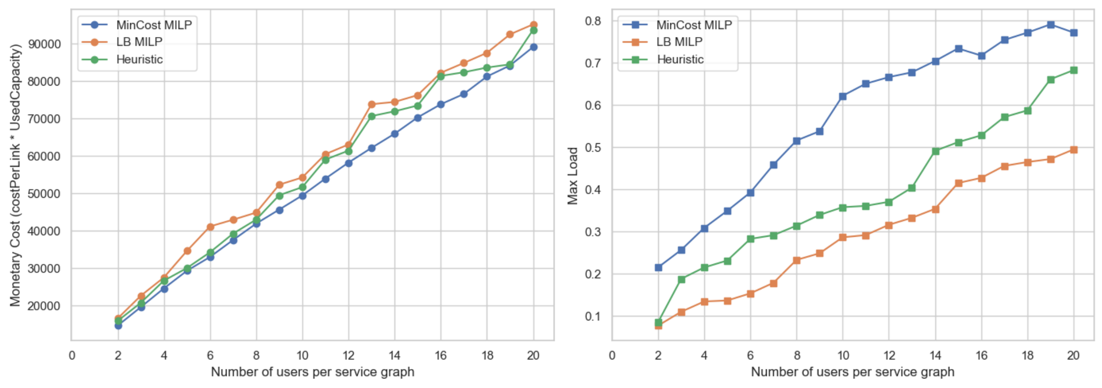


Figure 9.13. Cost-load trade-off for the 16-node Cloud Network Graph (CNG).

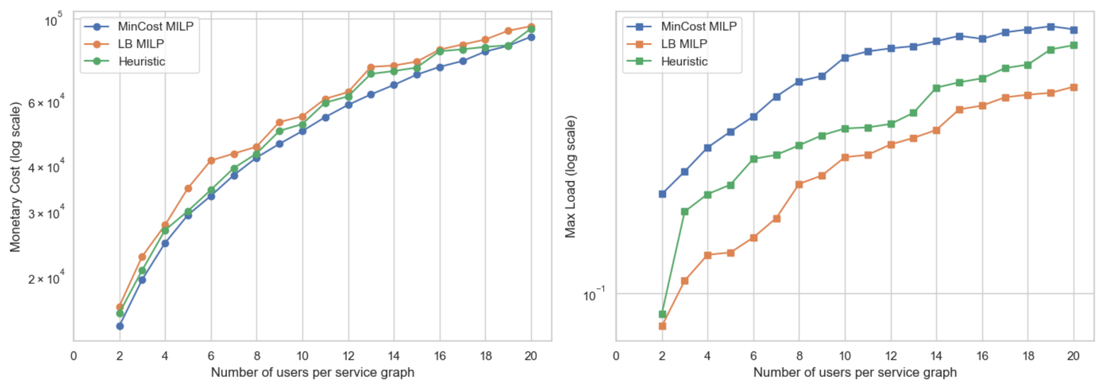


Figure 9.14. Cost-load **log scale** trade-off for the 16-node Cloud Network Graph (CNG)

Even at only two users the Heuristic runs twice as fast as MinCost and 36× faster than LB. By 15–20 users the speed-up jumps to ≈ 12 − 29× over MinCost and above 250× over LB. Therefore, **for a 16-node topology the MILP solvers become prohibitive for on-line use**, whereas the Heuristic still completes every instance within ≈ 1 s.

Across the entire range from 2 to 20 users, the heuristic consistently drives the **lowest overall substrate consumption** of all three methods. At very light loads (2–5 users), it cuts total capacity usage by around 15-20% compared with the Min-Cost MILP and by 5–10 percent compared with the Load-Balanced (LB) MILP. As the number of users grows and the network fills up, the gap naturally shrinks. However, even at 20 users, the heuristic still uses about 1–2 percent less capacity than the LB-oriented MILP and 3 percent less than the Min-Cost MILP.

Although the Min-Cost MILP is tuned to minimize a cost metric, those cost weights

#	MinCost	LB	Heuristic	MinCost/Hr	LB/Hr
2	0.403	7.795	**0.219**	x1.8	x35.6
3	0.479	1.674	**0.205**	x2.3	x 8.2
4	0.964	9.224	**0.369**	x2.6	x25.0
5	1.332	16.414	**0.218**	x6.1	x75.2
6	1.652	71.173	**0.232**	x7.1	x306.9
7	1.734	40.344	**0.305**	x5.7	x132.4
8	1.470	30.605	**0.324**	x4.5	x 94.3
9	3.108	41.316	**0.428**	x7.3	x 96.7
10	3.296	48.943	**0.453**	x7.3	x108.1
11	3.408	56.050	**0.571**	x6.0	x 98.0
12	4.037	28.151	**0.638**	x6.3	x 44.1
13	9.003	62.135	**0.645**	x14.0	x 96.3
14	11.221	127.846	**0.679**	x16.5	x188.4
15	8.785	554.794	**0.736**	x11.9	x753.7
16	9.967	212.429	**0.763**	x13.1	x278.4
17	11.672	93.140	**0.881**	x13.2	x105.7
18	24.738	159.234	**0.960**	x25.8	x165.9
19	26.265	119.060	**0.922**	x28.5	x129.2
20	29.250	294.627	**1.015**	x28.8	x290.2
Average	8.041	103.945	**0.556**	x11.0	x159.6

Table 9.6. CPU-runtime comparison for the 16-node CNG.

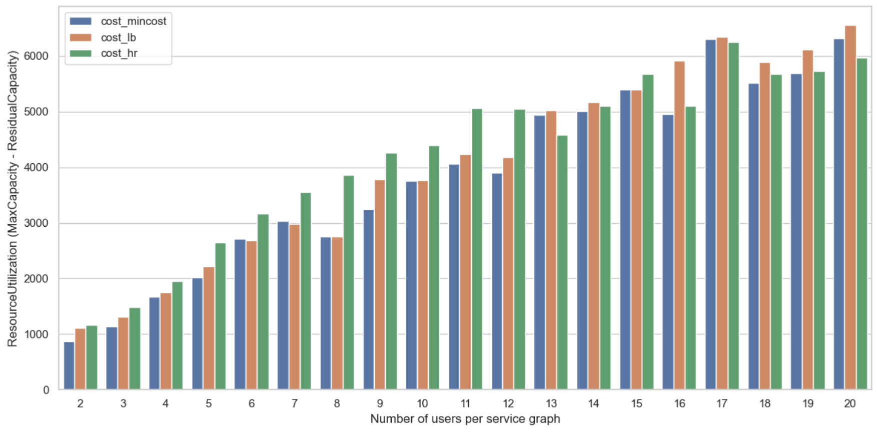


Figure 9.15. Resource Utilization comparison for the 16-node CNG

don't perfectly align with raw capacity consumption. In practice it over-allocates resources by up to 20 percent under light demand, and by roughly 3 percent under heavy demand, compared to our heuristic.

The LB-MILP was never designed to minimize resource usage (it focuses purely on fairness), yet the heuristic outperforms it on both fronts, achieving equal or better load balance and reducing total capacity consumption by up to 8 percent when the network is under-utilized, and still by a couple of percent at peak loads.

As user demand ramps from 2 to 20 concurrent service graphs, the absolute differences between methods narrow simply because there's less "slack" left in the substrate. However, the heuristic never relinquishes its lead: it scales smoothly, always making the most of available capacity.

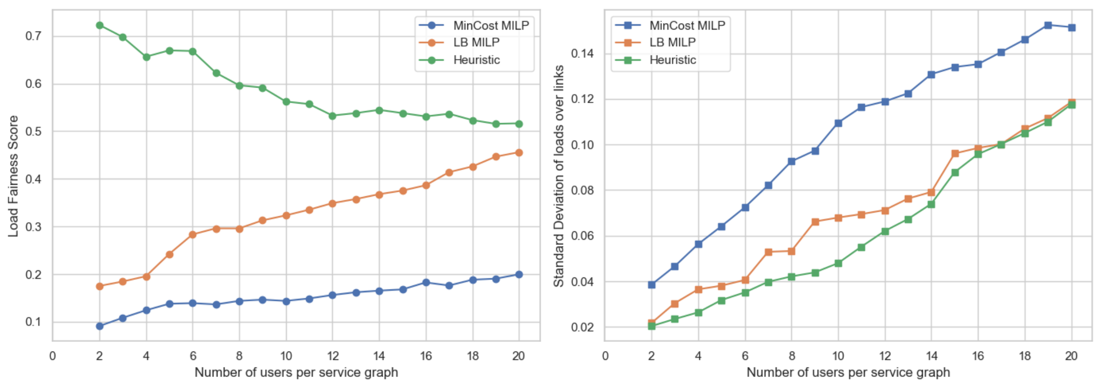


Figure 9.16. Fairness and Standard Deviation of load comparison for the 16-node CNG

- Fairness: MinCost MILP stalls near 0.18-0.15. LB MILP steadily climbs from 0.35 to 0.45. Our heuristic starts at 0.72 and, despite a gradual decline, remains above 0.50 at 20 users outperforming LB MILP by 0.10–0.18 up to 15 users and then nearing parity.

- Standard deviation: The heuristic yields the lowest deviation throughout (0.02-0.12), compared to LB MILP's 0.04-0.17 and MinCost s 0.06-0.22.

9.3 Outcomes Analysis

This chapter benchmarked the proposed HEURISTIC algorithm, against two exact MILP baselines on cloud–network graphs of 4–16 nodes and for 2–20 concurrent service graphs. The evaluation covered monetary cost, peak link load, total activated capacity, fairness, and run-time.

- **Cost**: The heuristic remained within **5 %** of the global optimum (MinCost MILP) for every instance.

- **Run-time**: Median speed-ups over MinCost grew with graph size: $\times 1.5$ (4 nodes), $\times 8$ (8 nodes), and $\times 11$ (16 nodes). Against the load-balancing MILP the gain was always at least two orders of magnitude.
- **Congestion**: Peak link utilization was $\approx 40\,\%$ lower than MinCost and within 15–20 percent of the LB MILP, but without its cost and run-time penalties.
- **Capacity**: Under heavy load the heuristic activated 8–15 % less total capacity than MinCost, and close to LB.
- **Fairness**: Load-fairness scores were consistently higher than MinCost and close to LB, while the standard deviation of link loads stayed below 0.15 in all scenarios.

Conclusion: The proposed algorithm offers a practical trade-off: near-optimal cost, substantial reductions in computation time, and acceptable (often superior) network utilization metrics. It is therefore the preferred option for real-time service-graph embedding on medium to large cloud–network topologies.

Chapter 10

Conclusion and Future Work

Conclusion

This thesis thoroughly investigated the complex challenges and innovative solutions required to meet stringent communication demands of emerging applications, within the forthcoming the ”Musical Metaverse.” By critically analyzing traditional Virtual Network Embedding (VNE), we identified that conventional models, treating virtual links as independent point-to-point demands without awareness of content or flow interdependencies, are insufficient for supporting real-time interactive musical collaboration requiring stringent Quality of Service (QoS), ultra-low latency, and optimal Quality of Experience (QoE).

In response, we introduced the **MusMOPT framework**, a novel optimization approach representing services as directed acyclic graphs (DAGs) within an augmented cloud-network graph. This method significantly enhances traditional VNE by unifying service placement, routing, and resource allocation into an integrated, **information-centric optimization problem**, effectively addressing the unique network orchestration challenges in the Musical Metaverse.

Furthermore, this research proposed a **two-layer abstraction model** for Musical Metaverse service design: a dynamic graph-layer that accommodates evolving participation and collaboration demands, and a data-layer that precisely captures real-time information flow behaviors, explicitly addressing network impairments such as congestion, jitter, and packet loss.

To concretely demonstrate our theoretical advancements, we implemented a representative virtual concert service graph, showcasing MusMOPT’s practical applicability and efficacy in modeling real-world interaction scenarios. Additionally, the **SiMusMet simulator**, a modular Python-based tool specifically developed in this thesis, provides a robust environment for dynamic service embedding, real-time traffic simulation, and comprehensive evaluation of QoS and QoE across diverse network conditions, thereby significantly aiding future research endeavors.

Finally, this thesis introduced two Mixed-Integer Linear Programming (MILP)-based optimization methods Min-Cost and Load-Balancing, and **a novel** Topology-Aware

Cost–Load **heuristic**, tailored to overcome scalability constraints associated with real-time, interactive performances. Simulation results confirmed the heuristic's practical value, achieving computational efficiency improvements of 70–80% in small to medium-scale networks, and approximately 65% in larger networks, while closely approximating optimal solutions at varied load levels.

Collectively, these contributions: the MusMOPT framework, two-layer modeling, SiMusMet simulator, and efficient heuristic algorithm, mark substantial progress towards realizing next-generation immersive, interactive digital experiences, significantly advancing the field of communication networks.

Future Work

The research presented in this thesis opens promising avenues for future exploration to enhance the robustness, scalability, and applicability of solutions in the Musical Metaverse.

Firstly, extending the MusMOPT framework to handle more dynamic and complex service graphs is crucial. Adaptive algorithms leveraging reinforcement learning for predictive and real-time resource allocation are particularly promising.

Incorporating security and privacy mechanisms, including blockchain-based solutions and secure embedding strategies, is another critical direction to address user data and intellectual property protection.

Enhancing SiMusMet by supporting diverse network topologies (e.g., 5G/6G, satellite networks), realistic user-behavior traffic models, and an intuitive graphical user interface would greatly increase its usability and accuracy.

Future work should optimize the Topology-Aware Cost–Load heuristic using advanced metaheuristics such as genetic algorithms or particle swarm optimization, alongside hybrid techniques, to further improve efficiency and performance. Incorporating explainability methods will also provide essential insights into model behaviors.

Beyond the Musical Metaverse, the MusMOPT approach can be adapted for broader latency-sensitive applications, including remote surgery, autonomous vehicles, and industrial automation.

Finally, empirical validation through real-world testbeds, possibly in collaboration with telecommunications providers, is essential to demonstrate practical feasibility and effectiveness.

Bibliography

[1] G. P. Alkmim, D. M. Batista, and N. L. Da Fonseca. Mapping virtual networks onto substrate networks. *Journal of Internet Services and Applications*, 4:1–15, 2013.

[2] D. G. Andersen. Theoretical approaches to node assignment. 2002.

[3] M. T. Beck, A. Fischer, J. F. Botero, C. Linnhoff-Popien, and H. de Meer. Distributed and scalable embedding of virtual networks. *Journal of Network and Computer Applications*, 56:124–136, 2015.

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[5] A. Boem, M. Tomasetti, and L. Turchet. Issues and challenges of audio technologies for the musical metaverse. *J. Audio Eng. Soc*, 73(3):94–114, 2025.

[6] R. Buyya, M. Pathan, and A. Vakali. *Content delivery networks*, volume 9. Springer Science & Business Media, 2008.

[7] Y. Cai, J. Llorca, A. M. Tulino, and A. F. Molisch. Compute- and data-intensive networks: The key to the metaverse. In *2022 1st International Conference on 6G Networking (6GNet)*, pages 1–8, 2022.

[8] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 70(6):066111, 2004.

[9] M. Dorigo. Optimization, learning and natural algorithms. *Ph. D. Thesis, Politecnico di Milano*, 1992.

[10] A. Fischer, M. T. Beck, and H. De Meer. An approach to energy-efficient virtual network embeddings. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1142–1147. IEEE, 2013.

[11] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach. Virtual Network Embedding: A Survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013. Conference Name: IEEE Communications Surveys & Tutorials.

[12] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[13] M. He, L. Zhuang, S. Yang, Z. Xu, W. Li, and J. Lu. An energy-efficient vne algorithm based on bidirectional long short-term memory. *Journal of Network and Systems Management*, 30(3):45, 2022.

[14] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press, 1992.

[15] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. ieee, 1995.

[16] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[17] V. Lira, E. Tavares, M. Oliveira, E. Sousa, and B. Nogueira. Virtual network mapping considering energy consumption and availability. *Computing*, 101:937–967, 2019.

[18] S. B. Masti and S. V. Raghavan. Vna: An enhanced algorithm for virtual network embedding. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2012.

[19] M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[20] E. Rodriguez, G. P. Alkmim, N. L. da Fonseca, and D. M. Batista. Energy-aware mapping and live migration of virtual networks. *IEEE Systems Journal*, 11(2):637–648, 2015.

[21] C. Rottondi, C. Chafe, C. Allocchio, and A. Sarti. An overview on networked music performance technologies. *IEEE Access*, 4:8823–8843, 2016.

[22] A. Satpathy, M. N. Sahoo, L. Behera, and C. Swain. Rematch: An efficient virtual data center re-matching strategy based on matching theory. *IEEE Transactions on Services Computing*, 16(2):1373–1386, 2022.

[23] A. Satpathy, M. N. Sahoo, L. Behera, C. Swain, and A. Mishra. Vmatch: A matching theory based vdc reconfiguration strategy. In *2020 IEEE 13th international conference on cloud computing (CLOUD)*, pages 133–140. IEEE, 2020.

[24] A. Satpathy, M. N. Sahoo, A. K. Sangaiah, C. Swain, and S. Bakshi. Comap: An efficient virtual network re-mapping strategy based on coalitional matching theory. *Computer Networks*, 216:109248, 2022.

[25] A. Satpathy, M. N. Sahoo, C. Swain, M. Bilal, S. Bakshi, and H. Song. Gamap: A genetic algorithm-based effective virtual data center re-embedding strategy. *IEEE Transactions on Green Communications and Networking*, 8(2):791–801, 2023.

[26] K. K. TG, S. K. Addya, A. Satpathy, and S. G. Koolagudi. Nord: Node ranking-based efficient virtual network embedding over single domain substrate networks. *Computer Networks*, 225:109661, 2023.

[27] K. K. Tg, A. Srivastava, A. Satpathy, S. K. Addya, and S. G. Koolagudi. Matchvne: a stable virtual network embedding strategy based on matching theory. In *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 355–359. IEEE, 2023.

[28] A. Tizghadam and A. Leon-Garcia. On robust traffic engineering in transport networks. In *IEEE GLOBECOM 2008-2008 IEEE Global Telecommunications Conference*, pages 1–6. IEEE, 2008.

[29] V. A. Traag, L. Waltman, and N. J. Van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):1–12, 2019.

[30] L. Turchet. Musical metaverse: vision, opportunities, and challenges. *Pers Ubiquit Comput 27, 1811–1827*, 2023.

[31] L. Turchet, C. Fischione, G. Essl, D. Keller, and M. Barthet. Internet of musical things: Vision and challenges. *IEEE Access*, 6:61994–62017, 2018.

[32] L. Turchet, M. Lagrange, C. Rottondi, G. Fazekas, N. Peters, J. Østergaard, F. Font, T. Bäckström, and C. Fischione. The internet of sounds: Convergent trends, insights, and future directions. *IEEE Internet of Things Journal*, 10(13):11264–11292, 2023.

[33] L. Turchet, B. O'Sullivan, R. Ortner, and C. Guger. Emotion recognition of playing musicians from EEG, ECG, and acoustic signals. *IEEE Transactions on Human-Machine Systems*, 2024.

[34] T. Wood, P. J. Shenoy, A. Venkataramani, M. S. Yousif, et al. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, volume 7, pages 17–17, 2007.

[35] H. Xu and B. Li. Anchor: A versatile and efficient framework for resource management in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1066–1076, 2012.

[36] H. Yoo, Y.-C. Lee, K. Shin, and S.-W. Kim. Directed network embedding with virtual negative edges. In *Proceedings of the fifteenth ACM international conference on web search and data mining*, pages 1291–1299, 2022.

[37] Y. Zhou, X. Yang, Y. Li, D. Jin, L. Su, and L. Zeng. Incremental re-embedding scheme for evolving virtual network requests. *IEEE communications letters*, 17(5):1016–1019, 2013.