Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electrical, Electronics, and Communications Engineering
(36^{th}cycle)

Enhancing Performance in Programmable 5G Networks, Blockchains, and Cloud Gaming

By

Iman Lotfimahyari

******

Supervisor(s):
Prof. Paolo Giaccone, Supervisor
Prof. Vittorio Curri, Co-Supervisor

Doctoral Examination Committee:
Prof. Stefano Ferretti, Referee, University of Bologna
Prof. Marco Savi, Referee, University of Milano-Bicocca

Politecnico di Torino
2024

Declaration

I hereby declare that the contents and organization of this dissertation constitute my own original work and do not compromise in any way the rights of third parties, including those relating to the security of personal data.

Iman Lotfimahyari
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to my loving wife and my parents

Acknowledgements

I would like to express my deepest gratitude to the numerous individuals who have played a pivotal role in completing this Ph.D. thesis.

Foremost, my heartfelt appreciation goes to my thesis advisors, Prof. Paolo Giaccone and Prof. Vittorio Curri, whose unwavering support, guidance, and mentorship have been instrumental throughout this journey. Your wisdom and encouragement have been a constant source of inspiration.

I am also immensely grateful to my thesis committee, Prof. Stefano Ferretti and Prof. Marco Savi, for their valuable insights, constructive feedback, and dedication to refining the quality of this work. To my colleagues and fellow researchers, I extend my thanks for the stimulating discussions, collaborative endeavors, and the strong sense of camaraderie that made this academic pursuit not only enlightening but also enjoyable.

I must acknowledge my loving wife, *Mahboobeh*, who has been my pillar of strength and persistent support throughout this journey. Her understanding, patience, and encouragement have sustained me through the challenges and triumphs of this endeavor. To my parents, I owe a debt of gratitude that can never be fully repaid. Their endless belief in my abilities and the sacrifices made to provide me with education opportunities have been the foundation of my academic achievements.

I would also like to express my appreciation for the financial support provided by *Bechain srl*, without which this research would not have been possible. This thesis represents the collective effort, encouragement, and sacrifices of all these individuals, and I am truly grateful for their pivotal roles in my academic journey.

Abstract

The research described in this dissertation explores three distinct technological domains: programmable networks, network-wide distributed blockchain technologies, and cloud gaming. Nevertheless, these domains may appear distinct, the first two often coexist and interact closely in modern distributed applications, and the last one is an example of a modern distributed application. In the context of enhancing overall performance, it's crucial to recognize that each component can potentially serve as a bottleneck that needs improvement. The primary research question underlying this work is centered on optimizing network-wide distributed applications to minimize latency, reduce resource consumption, and alleviate network overhead. Through tailored solutions for each domain, such as novel mechanisms for state replication in 5G networks, optimization of blockchain internal processes for latency-sensitive applications, and development of efficient algorithms for game engine module placement in distributed gaming environments, this dissertation aims to address these challenges comprehensively. By acknowledging the interconnectedness of these technological landscapes and their impact on performance, this research contributes to a holistic approach to improving network-wide distributed applications.

As the first research domain, we consider modern 5G networks capable of providing ultra-low-latency and highly scalable network services by employing modern networking paradigms, such as Software-Defined Networking (SDN) and Network Function Virtualization (NFV). The latter enables performance-critical network applications to be run in a distributed fashion directly inside the infrastructure. Being distributed, those applications rely on sophisticated state replication algorithms to synchronize states among each other. In such a scenario, we propose STARE, a novel state replication system tailored for 5G networks. At its core, STARE exploits stateful SDN to offload replication-related processes to the data plane, ultimately reducing communication delays and processing overhead for virtual network functions. We provide a detailed description of the STARE architecture

alongside a publicly available P4-based implementation. Furthermore, our evaluation shows that STARE is capable of scaling to large networks while introducing low communication overhead.

Regarding the second research domain, blockchains offer trust and immutability in untrusted environments, but they are typically not fast enough for latency-sensitive applications. Hyperledger Fabric (HF) is a common enterprise-level platform offered as a fast Blockchain-as-a-Service (BaaS) by cloud providers. In HF, every new transaction requires a preliminary endorsement by multiple mutually untrusted organizations, contributing to the delay in storing the transaction in the blockchain. The endorsement policy is specific for each application and defines the required approvals by the Endorser Peers (EPs) of the involved organizations. Given an input endorsement policy, we study the optimal choice to distribute the endorsement requests to the proper EPs. We propose the OPtimal ENdorsement (OPEN) algorithm, devised to minimize the latency due to network delays and the processing times at the EPs. Through extensive simulations, we show that OPEN can reduce the endorsement latency by up to 70% compared to the state-of-the-art solutions and can approximate well the optimal policies while offering a negligible implementation overhead compared to them.

In the third research domain, online gaming has seen a significant surge in popularity, becoming a dominant form of entertainment worldwide. This growth has necessitated the evolution of game servers from centralized to distributed models, leading to the emergence of distributed game engines. These engines allow for the distribution of Game Engine Modules (GEMs) across multiple servers, improving scalability and performance. However, this distribution presents a new challenge: the game engine module placement problem. This problem involves strategically placing GEMs to maximize the number of accepted placement requests while minimizing the delay experienced by players, a critical factor in enhancing the gaming experience. We show that the problem can be formulated as an Integer Linear Programming (ILP) model. It provides an optimal solution but suffers from high computational complexity, making it impractical for real-world applications. To address this challenge, our research introduces two novel heuristic algorithms, MAP-MIND and MAP-MIND*. The MAP-MIND algorithm demonstrates superior performance, achieving near-optimal delay and more than 92% GEM request acceptance in the worst heterogeneous scenarios. The MAP-MIND* algorithm, while slightly under-performing MAP-MIND in terms of delay, proves to be significantly

faster, making it a viable alternative for real-world applications with equal GEM request acceptance. The trade-off between the two algorithms offers a flexible approach to GEM placement, balancing performance and computational efficiency.

Contents

Chapter 1

Introduction

Nowadays, modern network technologies are evolving more towards increasing performance and functionality. The integration of novel technologies into this dynamic world is more and more promising as it introduces new use cases and opportunities. Nevertheless, to achieve these advances, some new challenges must be addressed through carefully advised solutions. In other words, the modern infrastructure of the current fast-evolving systems (e.g., distributed applications) needs to be optimized to enhance performance. This would need some advancements designed thoughtfully for these demands, which may require the integration of some cutting-edge technologies to meet the specific needs of this digital age.

In modern distributed applications, managing state sharing across distributed components is critical for maintaining consistency and preventing data inconsistencies, often achieved through techniques like distributed locking, replication, and event sourcing. Trusted decision-making and data integrity are safeguarded through robust authentication and authorization mechanisms, encryption techniques, digital signatures, and consensus mechanisms, fostering trust and reliability in distributed environments.

Efficient performance in modern distributed applications hinges on minimizing latency, a task that necessitates the utilization of streamlined communication protocols, edge computing, caching mechanisms, and asynchronous processing models. Researchers have delved into numerous strategies to optimize latency, including refining communication protocols, harnessing edge computing resources, implementing caching systems, and embracing asynchronous processing techniques to reduce

round-trip times and enhance user satisfaction. Our efforts in this domain have been marked by thorough research and innovative proposals aimed at addressing key challenges such as latency reduction while facilitating secure state-sharing among potentially untrusted parties, or simultaneously optimizing user experience by expediting request acceptance and minimizing latency in applications like cloud gaming. Through our relentless pursuit of optimized solutions, we aspire to establish resilient, high-performing distributed systems capable of meeting the demands of modern computing environments while prioritizing trustworthiness and integrity.

We start our research by deepening into two different and distinct areas of modern technology, namely programmable networks, and blockchains which are undoubtedly intersected by the ever-growing networking field. Finally, we considered cloud gaming, as an obvious example of a next-generation distributed application with a new paradigm in the entertainment industry, to be investigated.

Each of these works provides unique facilities in terms of solutions for the recent challenges we face day-by-day in this dynamic landscape.

1.1 5G Network Evolution: NFV-SDN Synergy with P4 Switches

The evolution of modern networking infrastructures is characterized by transformative changes driven by Software Defined Networking (SDN) [1], Network Function Virtualization (NFV) [2], and Programmable Data Planes, particularly with the emergence of technologies like Programming Protocol-independent Packet Processors (P4) [3]. These domains collectively represent a paradigm shift in how networks are designed, managed, and operated, offering unprecedented flexibility, agility, and efficiency in meeting the evolving demands of modern communication environments. Rapid growth in demand for fast and reliable services has made NFV and SDN the main pillars of modern 5G management infrastructures, as emphasized by ETSI [4].

SDN revolutionizes network management by decoupling the control plane from the data plane, enabling centralized control and programmability of network behavior through software-based controllers. SDN introduces flexibility and scalability by abstracting network control functions from the underlying hardware, allowing for dynamic provisioning, traffic engineering, and policy enforcement across heterogene-

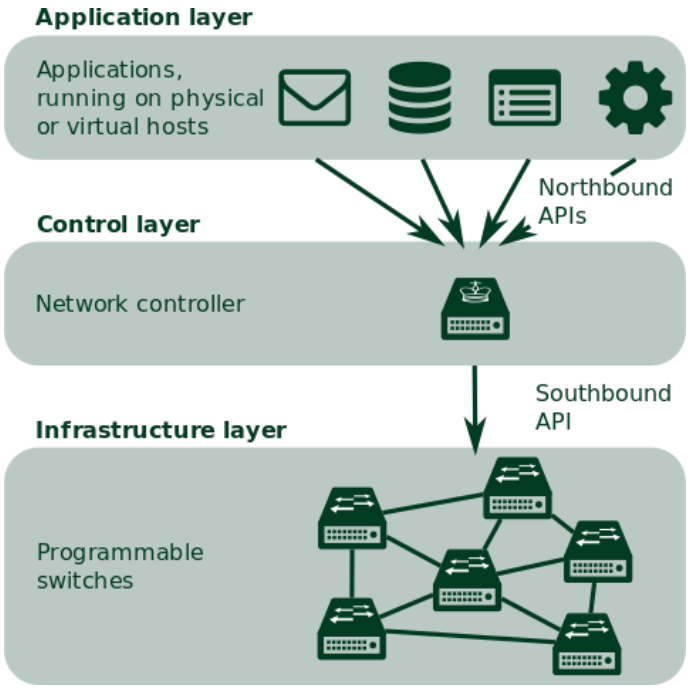


Fig. 1.1 SDN model [1].

ous network environments. This architectural shift is depicted in Fig. 1.1, illustrating the separation of control and data planes and the centralized control provided by the SDN controller [1]. SDN controllers, such as OpenDaylight [5] or ONOS [6], provide a centralized point of control for managing network resources and implementing network policies.

Network Function Virtualization (NFV) transforms traditional network architectures by virtualizing network functions previously implemented in dedicated hardware appliances. NFV replaces proprietary hardware with software-based virtualized network functions (VNFs) running on commodity servers, enabling on-demand service deployment, scaling, and chaining to meet changing service requirements. Fig. 1.2 showcases the concept of NFV, illustrating the virtualization of network functions [2].

Furthermore, Programmable Data Planes, exemplified by technologies like P4, extend the programmability and customization of network devices to the packet processing level. P4 provides a high-level language for defining packet processing logic, allowing network operators to tailor packet forwarding behavior based on specific applications, protocols, or policies. P4 allows for the definition of flexible

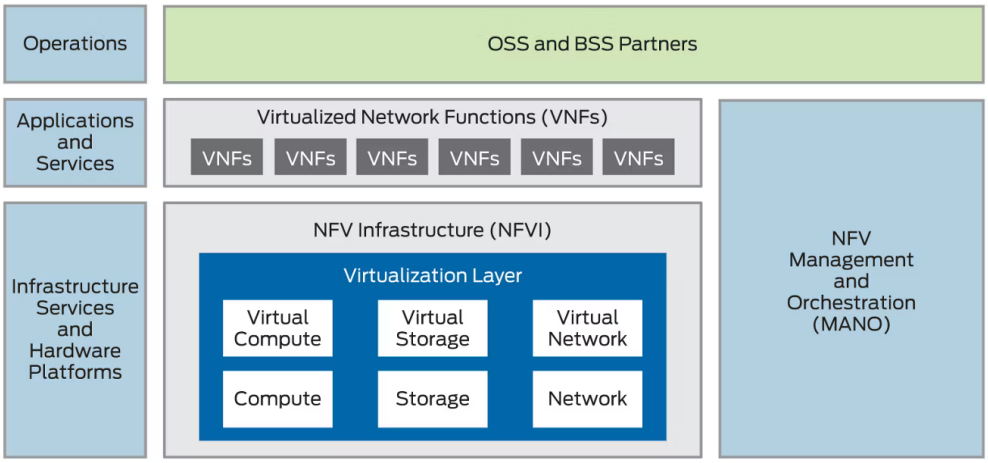


Fig. 1.2 NFV model [2].

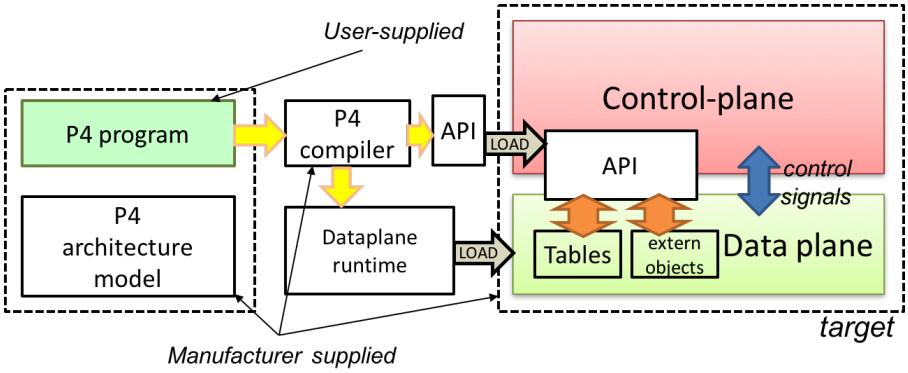


Fig. 1.3 The P4 abstract switch model. [8].

and customizable packet processing pipelines, enabling the implementation of complex forwarding behaviors tailored to specific network requirements. Fig. 1.3 demonstrates a general P4 workflow showcasing the customizable packet processing pipeline and the ability to define packet forwarding rules based on application requirements [7]. Moreover, this programmability at the data plane level complements the centralized control provided by SDN, allowing for fine-grained control and optimization of packet forwarding behavior.

Network operators now face a scenario where general-purpose devices are widespread and easily programmable, opening opportunities for deploying new control and management applications and running applications close to end customers.

However, new challenges arise related to the necessity of richer coordination schemes among network applications, which rely on shared global states across multiple VNFs. Additionally, while speed and availability remain crucial, application transparency becomes fundamental in modern infrastructures, demanding a simple, adaptable system for facilitating data exchange across various VNFs with minimal modification to their fundamental setup.

Advancements in SDN have impacted SDN switches' architecture and greatly improved SDN controllers' scalability and performance. NVF enables efficient resource utilization and reduces operational costs compared to traditional hardware-based network deployments. Programmable data-planes provide a novel methodology that is changing next-generation SDN switches, enabling customers to incorporate custom code directly into switches for customized packet processing without introducing latency. Technologies like P4 describe how data plane devices process packets, promoting creativity in applications for traffic control and monitoring [9].

These technological advancements offer unparalleled flexibility and control over network operations, empowering organizations to build agile, efficient, and scalable networks capable of supporting a diverse range of applications, services, and deployment scenarios.

1.2 Network in Blockchains: The Hyperledger Fabric Endorsement case

Nowadays, Blockchains have become an important asset across a wide range of ICT applications, serving as a foundational pillar that fosters trust and reliability among various entities or parties amidst uncertainties in untrusted environments. Beyond their fundamental role, Blockchains offer a multifaceted enhancement to the ICT domain by addressing crucial aspects of security and privacy while promoting a decentralized organizational structure. The immutability intrinsic to Blockchain technology ensures not only the integrity of stored data but also provides unique visibility and traceability, attributes particularly beneficial in sectors such as banking, supply chain management, IoT, healthcare, and the energy industry [10–12].

A Blockchain is essentially a digital ledger that securely and transparently records transactions across a network of computers. Each transaction is bundled into a

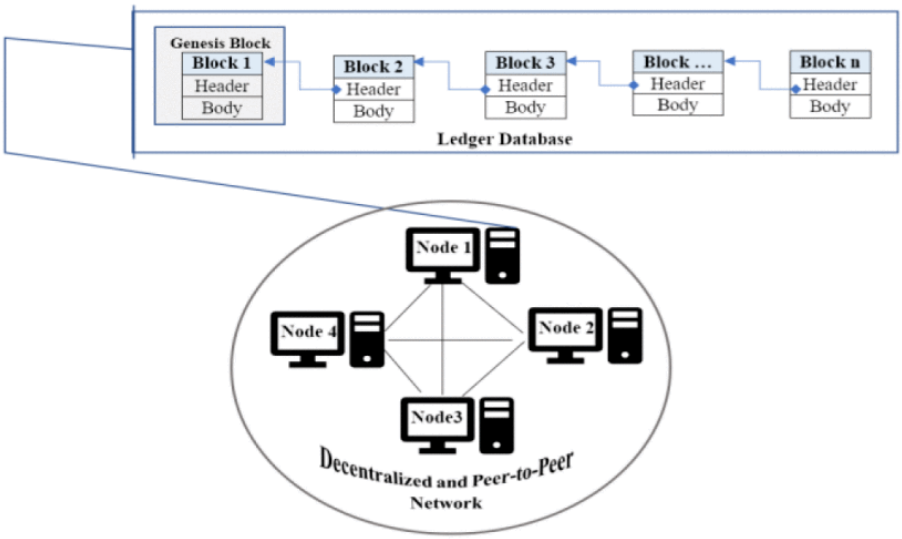


Fig. 1.4 Blockchain structure. [13].

"block" and added to a chain of previous transactions, creating a chronological and unalterable record (see Fig. 1.4). In simple terms, think of a Blockchain as a shared, tamper-proof database that enables secure and transparent transactions without the need for intermediaries.

The applications of Blockchain technology are diverse and far-reaching. One of its primary use cases is in financial transactions, where it enables secure and efficient peer-to-peer transfers of digital assets such as cryptocurrencies like Bitcoin. Beyond finance, Blockchain has been applied to supply chain management, ensuring the traceability and authenticity of goods from origin to destination. It also holds promise in fields like healthcare, where it can securely store and share patient records while maintaining privacy and security. Essentially, Blockchain technology offers a decentralized solution to various challenges associated with trust, security, and transparency, making it a powerful tool for innovation across industries.

The classification of Blockchains introduces a spectrum of accessibility and permissions. A Blockchain is named public when its ledger is open for everyone to inspect, whereas it will be considered private when access is restricted. Additionally, the main distinction between permissioned and permissionless Blockchains revolves around the participation requirements for validating transactions, a nuanced consideration underscored by the literature on the subject [14]. Unlike public permissionless Blockchains like Bitcoin [15], the realm of enterprise applications demands performa-

nce metrics that often surpass the capabilities of their permissionless counterparts. This is especially evident in scenarios where use cases dictate the necessity of knowing the identities of involved participants.

Looking ahead, the future of blockchains holds immense promise for further innovation and adoption. With ongoing research and development efforts focused on improving scalability, privacy, and usability, blockchain technology is poised to revolutionize various industries, including finance, supply chain management, healthcare, and identity verification. As the technology continues to mature and overcome existing limitations, it has the potential to reshape the global economy and empower individuals and organizations with unprecedented levels of trust, security, and transparency.

For instance, in the complex landscape of financial transactions, where adherence to notary service regulations is paramount, a delicate approach becomes indispensable. It is in response to these intricate demands that private-permissioned blockchains come to the fore. Platforms such as Hyperledger Fabric (HF) [16] and Corda [17] appear as tailored solutions, adeptly meeting the exact regulatory and performance requirements of these specialized domains. These private-permissioned blockchains not only assure compliance but also offer a flexible and customizable framework, aligning seamlessly with the complex needs of current ICT applications.

HF blockchain serves as the framework for constructing modular apps or solutions. It supports replaceable components, such as consensus and membership services, allowing for a plug-and-play environment. It is also intended to fulfill the needs of a wide range of industries. Scalability and speed are two main drawbacks of most blockchains which mostly relate to their consensus. In contrast, HF provides a novel way to consensus that allows for scalable performance while protecting anonymity.

Smartly, HF uses an architecture called Execute-Order-Validate for transactions, enabling the definition of endorsement policies. The endorsement policy is a logical combination of the participants that based on it a transaction should be approved by participants (e.g., from four participants namely A, B, C, and D, we need A and one among B, C, and D). This design in which HF executes transactions before reaching a final agreement on their order departs completely from the formal order-execute way (e.g., Proof of Work (PoW) consensus). During the execution phase, the client sends the transaction to some nodes called Endorser Peers (EPs), based on the user's specified endorsement policy. Each EP processes the transaction by only simulating

it without applying the results on the blockchain. The simulation result, denoted as "endorsement", is signed by the EP and returned to the client. Finally, if the endorsement policy is satisfied through the participants that calculated and signed the transaction, the signed and endorsed proposal of the transaction will be sent to the blockchain nodes to be stored. This important but smart change led to a massive increase in the scalability of HF.

Hyperledger Fabric deserves to be on any pragmatic assessment of blockchain systems' shortlist. When combined, Fabric's unique features allow it to be a highly scalable permissioned blockchain system with flexible trust assumptions. This allows the platform to support a wide range of industry use cases, including supply-chain logistics, government, banking, healthcare, and much more [18].

1.3 Gaming Evolution: From Online Gaming to Cloud Gaming

In recent years, a notable trend in the world of video games has been observed. They are increasingly evolving into online services. In an online game, the gaming experience mostly relies on a server that provides a shared exchange point for many players. This central point may provide a single-user (i.e., single-player) experience as well as a shared virtual environment for the users to interact with each other. In any case, the server is responsible for efficiently managing the shared resources among the players, independently of the game mechanics.

Online games' evolution has seen many stages. In the beginning, a centralized (physical) server was in charge of managing all gaming sessions, with obvious scalability issues. In the second generation, with the increasing availability of cloud computing, gaming servers have been distributed (replicated) in the cloud and run as a combination of virtualized software services. Even though the adoption of cloud computing to implement partial cloud gaming allowed for significant cost reduction and a scalability increase, this paradigm may require special proprietary hardware and/or software from the user side to use the gaming platform.

In the last generation, complete cloud gaming has been introduced. In complete cloud gaming (or just cloud gaming which is also known as game-streaming), the computation is completely offloaded to the remote (cloud-based) server, and a video

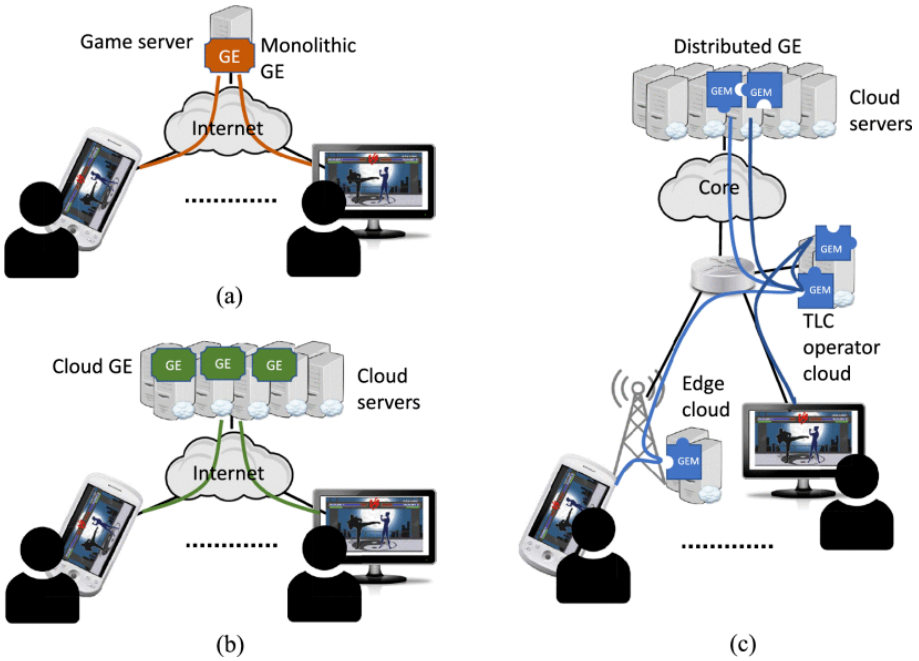


Fig. 1.5 Game Engine (GE) evolution (a) Monolithic GE. (b) Cloud GE. (c) Distributed GE. [23].

stream is sent to the player. A client handles the player's inputs, which are sent back to the game server. This paradigm shift eliminates the need for gamers to acquire and maintain cutting-edge hardware while making high-quality gaming experiences more accessible than ever before. As a result, the number of players using game streaming is ramping up, opening up new business opportunities for modern game companies. Services like Google Stadia [19] (now discontinued), Amazon Luna [20], GeForce Cloud Gaming [21], and Microsoft Xbox Cloud [22] exemplify this model, where the player's device is only responsible for displaying the game scene streamed from private data centers equipped with dedicated hardware.

Hopefully, as discussed in [23], the new way to provide gaming services will move millions of existing and next-generation players from legacy platforms to cloud gaming (see Fig. 1.5.

Anyway, such a large-scale migration might generate a workload able to overtax even hyper-scaling cloud architectures. While the capability to scale up in terms of raw computation power can be given for granted, modern cloud architectures are

not designed to organize internal modules while taking into account strict Quality of Service (QoS) requirements.

1.4 Organization of the thesis

This thesis focuses on addressing additional research questions motivated by the incorporation of programmable data planes into modern 5G use cases (Chapter 2). In Chapter 3, we analyze the role of the network in the distributed world of blockchain applications, exploring more specifically the platforms such as Hyperledger Fabric. We also discuss the growing area of cloud gaming (Chapter 4).

We carefully investigate the ways that programmable data planes can improve the efficiency of data flows over wide-area networks. Through our shift to rapid and effective resource use in blockchain settings, we want to find solutions that improve the overall effectiveness of these decentralized systems. Ultimately, we go into the complexities of effectively allocating resources for distributed game engines, focusing particularly on coordinating the procedure with users' perceived Quality of Experience (QoE).

Chapter 2

Data-plane assisted state replication

A part of the work presented in this chapter has been published in [24] and is reported here for the reader's convenience:

- I. Lotfimahyari, G. Sviridov, P. Giaccone and A. Bianco, "Data-Plane-Assisted State Replication With Network Function Virtualization," in IEEE Systems Journal, vol. 16, no. 2, pp. 2934-2945, June 2022.

2.1 Background and Context

Rapid growth in demand for fast and reliable services has made NFV and SDN the main pillar of modern 5G management infrastructures, as emphasized by ETSI [4]. SDN facilitates centralized network management by separating the data-plane in switches from the control plane in a network controller, allowing for precise control over network operations. NFV transforms the network infrastructure by replacing dedicated hardware devices with general-purpose machines like commodity servers, resulting in the virtualization of legacy devices as VNFs. This shift enables diverse applications, such as IoT and real-time applications, and the deployment of control and management applications for detailed network statistics. However, as new applications emerge, challenges arise in coordinating between VNFs. Network applications like DDoS detection algorithms, stateful load balancers, and user mobility services require communication and cooperative decision-making, relying on shared global states across multiple VNFs.

In modern infrastructures, alongside speed and availability, application transparency is crucial. The current implementation of VNFs deeply impacts the existing production network infrastructure, necessitating a flexible scheme for data sharing across VNFs with minimal modification to their original structure. The possibility of replicating local states on remote devices is the main pillar of most modern distributed applications. It allows systems to efficiently scale to large sizes, thus providing higher service availability and resilience.

The replication procedure typically involves a set of *states*, defined as generic data structures that are to be shared among different devices. Noteworthy, an efficient replication algorithm must guarantee that all shared states are kept "fresh" with respect to one another, i.e., all replicas should have the same value. An important example of such a necessity comes from distributed database systems, in which data is located in different regions. In the case of real-time services, freshness becomes of paramount importance, as otherwise it may introduce ambiguity or preclude the correct functionality of the service. To this end, multiple-state replication schemes, tailored for different applications, have been proposed.

Traditionally, simple gossip protocols have been widely employed to disseminate information among different sites by optimistically propagating information in the network [25]. Although easy to implement and requiring low computational and storage overhead, this family of algorithms inevitably incurs practical limitations related to data freshness and resilience to faults [26]. To solve these issues, modern services rely on a broad variety of *replication algorithms* with specific algorithms being chosen to satisfy particular service requirements [27]. Such requirements are summarized by the CAP theorem [28] which states that for a replication algorithm, out of Consistency, Availability, and Partition-tolerance only two properties can be chosen at the same time. More formally, the CAP theorem defines three fundamental properties a replication algorithm should satisfy:

1. *Consistency*: Any read performed on any replica of the state will always return the most (globally) recent value of such a state, or eventually an error. It follows that all non-faulty replicas will always contain the same value for a given state.

2. *Availability*: The availability property guarantees that any read performed on any replica will yield a non-error result independently from the freshness of the state value. It follows that no service disruption is possible in the case of

faulty replicas. Yet, if consistency property is not satisfied, the system must be designed in such a way as to guarantee correct functionality in the presence of out-of-date state values.

3. *Partition-tolerance*: Partition-tolerance defines the reliability property of the replication algorithm. If this property is satisfied, the replication system is guaranteed to operate correctly even in the case of an arbitrary number of state-update messages being dropped or delayed by the network.

Being reliability a crucial factor in modern infrastructures, it follows that no practical replication algorithm can operate without satisfying the partition-tolerance property. This reduces the freedom of picking the possible property combinations down to Availability-Partition-tolerance and Consistency-Partition-tolerance, which leads to *strong consistency*[1] and *eventual consistency*[2] models for replication algorithms, respectively.

In the rest of this chapter, we discuss the issues of state sharing especially for synchronizing them across various Virtual Network Functions (VNFs). We provide an overview of how existing technologies are based on stateful data-planes and highlight possible future directions. Then, we propose our STAte REplication (STARE) mechanism and explain its implementation details alongside its operation modes. Subsequently, we evaluate STARE in an emulated test-bed to validate the approach, showing that it is feasible to be implemented and run in a real P4-based network. Alongside the evaluation, we quantify the resource occupancy to understand the scalability of the approach. Then, we compare it against alternative solutions, either centralized or distributed ones. Finally, we draw our conclusions.

2.2 Related works

The work in E-State [30] presents a state management framework to share the states between VNFs by creating a logically distributed state memory. It provides fast reads

[1]The strong consistency in distributed computing, guarantees that all data replicas provide the same value for a given operation at any time. It ensures immediate and uniform agreement across all replicas, achieved through mechanisms like distributed transactions or consensus protocols [28].

[2]The idea of eventual consistency, pivotal in distributed computing, suggests that data replicas might briefly diverge but will eventually align to a consistent state. It recognizes the complexities of distributed setups, prioritizing availability and resilience while ensuring eventual harmony among all replicas [29].

and writes to flow-related states on the local VNF instance, and when global reads are used, it provides an eventual consistency model, exactly as STARE assumes. In E-State, replication is managed directly within the application layer, thus reducing the net resources devoted to the VNF, especially when the replication is directed to a very large number of VNFs. On the contrary, in STARE, the replication is offloaded into P4 switches through a middleware running outside the VNF application. This reduces the resources required within each VNF, which is not involved anymore in the replication process. Furthermore, replication latency in STARE is mainly affected by network congestion and communication details, not by the CPU load on the VM or container running the VNF. Moreover, STARE is providing a simple interface to the middleware to facilitate VNF development.

The work [31] addresses the problem of live-migrating VNFs. The migration requires to transfer of the internal state of the original VNF to the new one in real time. Thus, the problem is similar to our state-sharing problem between VNFs, but with the peculiarity that the replication process occurs just once and in one direction. The proposed framework SHarP separates the state migration problem from the traffic steering scheme that reroutes the flows from the old VNF to the new one. By decoupling the state replication by the traffic management, the framework is compatible with any replication scheme, and in particular, it is compatible with our STARE scheme, offloading the state replication into the data-plane.

There is a substantial ongoing effort to investigate application-level acceleration and state replication via programmable data-planes. This is enabled by multiple stateful switch architectures capable of holding persistent states and performing custom packet processing, as proposed in the last few years. OpenState [32], which introduces a minor architectural extension to the OpenFlow data-plane and control plane, is capable of supporting custom persistent states inside the switch, which can be interacted with by custom routines upon packet arrival, internal switch signals, or timers. Open Packet Processor (OPP) [33] extended OpenState by adding additional features that allow the execution of Extended Finite State Machines (EFSM) directly in the data-plane. Similarly, P4-enabled switches [34] switches are capable of performing the same tasks as OPP-based switches, yet they also provide a comprehensive high-level programming language for the definition of custom packet processing routines.

In NetPaxos [35] the authors propose to move part of the traditional Paxos consensus protocol into the network to accelerate its application-layer performance. Being one of the most deployed protocols in distributed systems and a fundamental building block for several distributed applications [36–38], NetPaxos showed the potential benefit programmable data-planes can bring to the application level. On the contrary, in SwingState [39] a first attempt has been made to perform state migration entirely in the data-plane. The authors were able to dynamically migrate an in-switch internal state across different switches but assumed a single copy of the state that is on-demand migrated across the network. A step forward towards full data-plane solutions has been made in [40], which proposed LOADER, a programming abstraction for defining distributed network applications based on replicated states. Yet, similarly to [39], the authors of LOADER focused just on internal states available at the switches, thus without providing any interaction with other elements of the network such as servers or VNFs. STARE, instead, closes this gap by providing a comprehensible middleware between the application layer and the network. The consistency model for the state replication adopted in STARE is the same as the one considered in [40], for which the P4 implementation on the data-plane provides a prototypal idea for the implementation of the publish-subscribe scheme in STARE.

More related to our work, the work in [41] proposed to use the concept of group tables introduced in OpenFlow 1.3 to implement a publish-subscribe scheme to increase the performance in terms of data delivery and notification process. While it takes a step towards providing application-layer acceleration using SDN, it remains highly dependent on the remote controller for any change in the subscriptions, while STARE manages the subscription directly on the data-plane. The idea of [42] which is source-routing pubsub with P4, is that the publisher sends the notification packet containing a stack of headers similar to MPLS after the Ethernet header. Each stack has two fields containing the switch ID and a bit-mask of the ports of the mentioned switch that the copies of the Publish packet (notification) should be sent out of.

The authors of [42] propose a content-based publish-subscribe based on source-routing multicast which exploits P4 programmable data-planes. They show that their approach is better than competing schemes, including different implementations of Application Layer Multicast (ALM) with software brokers, unicast, and broadcast while having a similar structure to OpenFlow Multicast (OFM) [43]. In our work, we compared STARE with the same approach. We show that our solution behaves similarly to [42] in terms of the number of packets and network overhead during

the notification publish. Furthermore, apart from lacking scalability due to the absence of a complete decoupling of publishers and subscribers, the authors of [42] do not clarify how the subscription problem is handled in dynamically changing scenarios. Indeed, dynamic subscriptions and publishes are completely supported by construction in STARE, where they are handled without interaction with the SDN controller. The work in [44] has addressed the problem of how to optimally place replicated states within a programmable data-plane-enabled network. This has been performed by taking into consideration both the data traffic and the replication traffic overheads. The work has proposed a framework to optimize the number of replicas and their placement within the network, taking into account the main trade-off between data traffic and replication traffic. Differently from [44], STARE does not optimize the placement of the VNFs within the network. Yet, we foresee optimizing VNF placement as a potential future work.

2.3 State sharing

Recently, different proposals have emerged as substitutes for classic replication algorithms. Motivated by the ever-rising number of connected devices and an ever-growing number of different applications common to many devices, schemes such as *publish-subscribe* [45] have been proposed.

Traditional publish-subscribe algorithms work in three phases. i) A given subscriber can express their interest in a particular topic by specifying it to the broker. ii) The broker builds and keeps track of a map between specific topics and the subscribers interested in those topics. iii) Each time a publisher sends an update on a specific topic to the broker, the broker forwards the update to all of the subscribers to that topic. This scheme effectively enables one-to-many communication while decoupling in time and space the publishers from the subscribers. An example of a publish-subscribe model is demonstrated in Fig. 2.1.

2.3.1 Publish-Subscribe for VNF state synchronization

Low latency is among the main requirements for modern real-time network applications as it affects system reactivity and dramatically impacts user satisfaction. The latency requirement is made even more crucial for sharing application-critical states

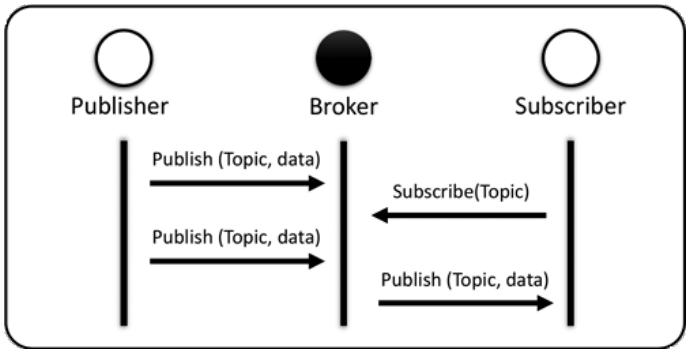


Fig. 2.1 Publish-Subscribe model [46].

across different applications as it may affect the correct functionality of applications relying on those states. From the point of view of a replication scheme, this translates into the necessity of providing availability. Furthermore, since the majority of modern distributed systems, such as IoT or cellular networks, operate in a loosely coupled or fully autonomous way, supporting partition tolerance becomes fundamental. Indeed, no network is immune to faults, so algorithms that implement some degree of resilience to network failures are required. As a consequence, the Consistency-Partition-tolerance model is typically deployed.

Publish-subscribe has been introduced as a possible solution for such kinds of environments by closely mimicking the eventual consistency models, thus providing fast and reliable state replication. Publish-subscribe protocols provide a message exchange mechanism between the two main types of actors involved in the algorithm, namely the publishers and the subscribers. The peculiarity of such schemes is that, differently from the traditional replication algorithm, most of the complexity is moved out of the endpoints and pushed inside a central entity, namely the *broker*, which is responsible for collecting the data from the publishers and distributing it to all the relevant subscribers.

The publisher is responsible for generating and sending messages (or events) to the messaging system. These messages contain information that the publisher wishes to distribute to interested parties. Publishers do not need to know who the subscribers are. They simply publish messages to a topic or a channel without concerning themselves with the specific recipients. Subscribers are entities that express interest in receiving specific types of messages or events from the messaging system. Subscribers subscribe to topics or channels, indicating which messages they

are interested in receiving. When a publisher sends a message to a topic/channel, the messaging system ensures that all interested subscribers receive the message. Subscribers can receive messages in real-time or retrieve them from a message queue, depending on the messaging system's capabilities.

The broker, which is sometimes referred to as the channel or messaging middleware, acts as an intermediary between publishers and subscribers. Its primary role is to receive messages from publishers and deliver them to the appropriate subscribers based on their subscription preferences. The broker manages the routing and delivery of messages, ensuring that messages are sent to all interested subscribers efficiently and reliably. Additionally, the broker may provide other features such as message filtering, transformation, and persistence, depending on the requirements of the messaging system.

Utilizing a publish-subscribe scheme proves advantageous in environments comprising diverse devices, as it necessitates minimal integration within the devices while providing flexibility in managing various states and implementing the safety properties inherent in traditional replication algorithms based on eventual consistency. However, unlike traditional gossiping algorithms, publish-subscribe schemes may encounter notable latency issues when the broker experiences excessive overload. Consequently, the applicability of such mechanisms is constrained, particularly for ultra-low-latency applications, such as those commonly targeted by 5G networks.

We opted for the publish-subscribe model for state sharing owing to its simplicity and efficiency. This model operates asynchronously, enabling components to function independently, thus streamlining implementation efforts. Additionally, it facilitates effective communication by empowering publishers to dispatch updates without the direct knowledge of subscribers. This characteristic renders it particularly suitable for distributed systems where state sharing transpires across multiple nodes or services.

In fact, in a publish-subscribe system which typically has a software-based broker, there are added constraints that can worsen latency issues during heavy loads. Software brokers face limitations in processing power due to hardware constraints, making it challenging to efficiently handle a surge in message volume. This can lead to delays in message delivery as the broker struggles to keep up with the workload. Additionally, scalability becomes an issue, especially if the broker is deployed on a limited number of servers, hindering its ability to scale up to meet increased demand. Resource contention further compounds the problem, as the broker competes for

resources with other processes running on the same hardware during peak periods, slowing down message processing and delivery. Moreover, the inherent overheads in the software's design and architecture, such as context switching and memory allocation, add to latency under heavy load conditions. Optimizing the software for varying load levels also poses a challenge, requiring careful configuration of parameters like thread pool sizes and network buffers to minimize latency. Effective management of these limitations is crucial for maintaining satisfactory performance in pub-sub systems [47, 48].

2.3.2 Programmable data-planes in networks

Recent advances in the field of SDN led to considerable improvements not only in the scalability and performance of SDN controllers but, most importantly, in the architecture of SDN switches.

Programmable data-planes emerged as a novel paradigm for the next generation SDN switches. Differently from traditional data-planes, programmable data-planes such as Programming Protocol-Independent Packet Processors (P4) [34] and Flowblaze [49] introduce the possibility of embedding user-defined programs directly inside switches, thus enabling the possibility of executing custom code during the packet processing pipeline at zero increased latency cost. Although commercial products employing programmable data-planes still have numerous limitations in terms of resources and programming semantics, the degree of programmability offered by programmable switches remains sufficiently high. This flexibility is mainly because, alongside programmability, programmable switches offer the possibility of keeping *persistent states* inside switches thanks to the presence of stateful elements such as registers and counters; hence, they are also usually referred to as *stateful data-planes*. At the same time, numerous programmable switch architectures offer the opportunity to define custom packet headers by directly programming the packet parser, which was widely employed to develop novel traffic management schemes [50] and monitoring applications [51]. The combination of all of the novel features introduced by programmable data-planes makes them a perfect foundation to build upon for novel algorithms targeting the in-network acceleration of application-layer services.

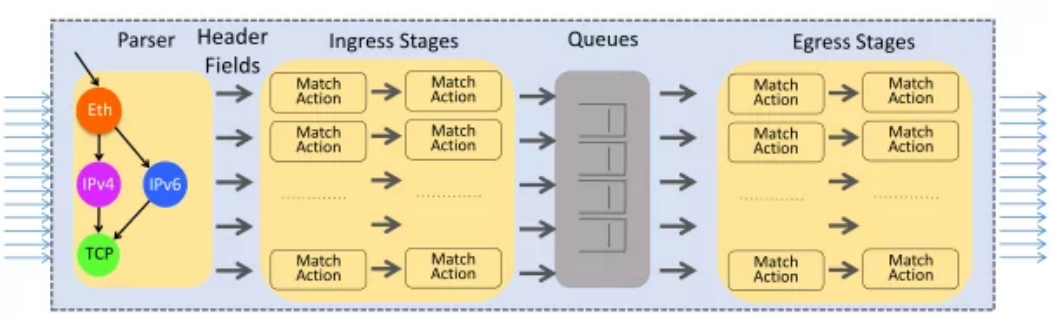


Fig. 2.2 The P4 abstract switch model [8].

P4 stands out as a revolutionary programming language designed specifically for specifying packet processing in networking devices like switches, routers, and NICs. Its protocol-agnostic nature empowers users to create highly customized packet processing pipelines, offering unparalleled flexibility and efficiency in network management. P4 benefits from a header parser based on a state machine that parses any customer header defined by the user and operates on a match-action paradigm, where packets are matched against predefined criteria, triggering corresponding actions. This approach allows for fine-grained control over packet forwarding, filtering, and modification, leading to optimized network performance and resource utilization. An abstract of the P4 switch model is demonstrated in Fig. 2.2.

One of P4's key advantages lies in its programmable data-plane elements, which enable users to define packet processing behaviors directly in hardware. By moving packet processing tasks closer to the network edge, P4 reduces latency and improves overall network responsiveness. Additionally, P4's target independence ensures compatibility with a wide range of hardware platforms, including ASICs, FPGAs, and software-based switches, offering unprecedented deployment flexibility. This versatility allows organizations to leverage existing infrastructure while seamlessly integrating P4-based solutions to meet evolving network requirements.

Moreover, P4 serves as a catalyst for innovation and research within the networking community. Its open-source nature encourages collaboration and experimentation, driving advancements in network programmability, security, and performance. Researchers and developers leverage P4 to explore novel packet processing algorithms, protocols, and architectures, leading to breakthroughs in areas such as programmable networks, network function virtualization (NFV), and software-defined networking

(SDN). In essence, P4 represents a paradigm shift in network programming, offering unmatched flexibility, efficiency, and innovation in packet processing.

In this work, we propose STARE, a mechanism that mitigates the aforementioned issues by providing an application layer-transparent method for state sharing in 5G networks. The replication in STARE is complementary to eventual replication schemes running within clusters of distributed SDN controllers. STARE provides seamless sharing of application-level states by partially replicating those states across different VNFs. This is achieved by exploiting advances in the field of SDN, specifically in the field of programmable data-planes. While in traditional SDN, switches are left with little to no decision-making capabilities, thus fully relying on the controller for any control plane-related operation, recent advances in the field of SDN, introduced the concept of *programmable data-planes* by enhancing switches with powerful packet processing pipelines and support of stateful operations.

To this end, STARE exploits these advances by defining a custom *publish-subscribe* protocol run directly in the data-plane, thus not requiring any additional hardware. By fully exploiting the potential of the stateful operations and programmable pipelines present in programmable switches, STARE can achieve high scalability and no additional latency while leading to memory-efficient *state replication* across multiple VNFs. Furthermore, STARE simplifies the development process of the VNFs by pushing the replication complexity down from each VNF to the network. This is done by providing a middleware shared across all of the VNFs hosted on a single server, which simplifies the design and provides a more transparent way for developing state-sharing procedures.

The actual benefit of STARE is both toward the network and VNFs. Indeed, reducing the memory consumption for the matching tables allows us to exploit the available matching tables better. Those are typically implemented with expensive and limited-size TCAMs and extend the applicability of network applications running in P4 switches. Furthermore, STARE is beneficial for VNF developers since they are exempt from the burden of managing the real-time replication protocols by being provided with standard STARE libraries.

Our main contributions are twofold: (1) validate the approach and show that STARE can run in a real P4-enabled network, and (2) evaluate the resource occupancy of the proposed solution to better understand the scalability of the approach.

2.3.3 State sharing in operational scenarios

This work has been motivated by an operational use case in the context of the European 5G-EVE project[3] targeting the tracking of users' mobility in smart cities-[52]. Mobility tracking enables many new applications: on-demand public transportation, crowd management, mobility planning, social distance monitoring, etc. All these applications fit very well with the pervasive nature of mobile networks. User mobility is of particular relevance to telecom operators. Its general knowledge permits us to perform network planning and provisioning to guarantee continuous and high-quality customer service. Nevertheless, in the case of novel scenarios targeted by 5G (e.g., connected vehicles, UAV, and smart manufacturing), such coarse-grained knowledge becomes insufficient, thus requiring more fine-grained measurements that would not only give insight into the number of users in a given cell but would also be able to capture the dynamics of the users' flow.

As shown in Fig. 2.3, the use case comprises a set of WiFi scanners deployed in an area around eNodeBs[4] used in the testbed. The WiFi scanners capture the probe request messages periodically sent by the smartphones, advertising the list of the WiFi access points to which they have been connected in the past. A tracking-mobility VNF runs in each edge cloud and processes the anonymized MAC addresses of the mobile devices observed by each WiFi scanner. This enables the possibility of tracking the device's mobility in a completely transparent way for the users. More details are available in [53]. To capture the mobility across an area spanning multiple eNodeBs, it is necessary to correlate the presence and the coverage time of any device across multiple VNFs. This requires sharing the internal states of each VNF. Indeed, whenever a previously detected MAC address by a given VNF is later detected by another VNF, it is possible to infer the spatial trajectory of the device and its speed. In the example scenario of Fig. 2.3, we have two VNFs (VNF_x and VNF_y) that exchange their internal states (S_x and S_y) with the timestamped list of all the observed MAC addresses. Thanks to this state sharing, each VNF can individually evaluate the direction of the path (either $x \rightarrow y$ or $y \rightarrow x$) and the corresponding average speed.

[3] https://www.5g-eve.eu/

[4] eNodeB is the short form for Evolved NodeB which is the evolved version of the Node B in UMTS. It serves as the base station that directly communicates with mobile devices, acting as their access point to connect to the LTE network.

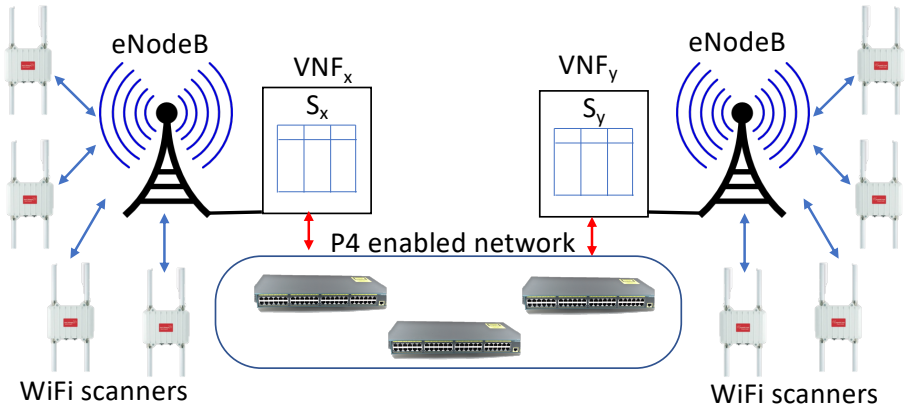


Fig. 2.3 Urban mobility tracking application with multiple WiFi scanners, leveraging a programmable data-plane connecting the different eNodeB for replication of global states between VNFs.

To support such an application, a centralized approach can be employed. Such an approach would require all the WiFi scanners to send their data to a single VNF which would aggregate the data from different eNodeB. This solution, although feasible in many scenarios, has limited scalability and does not exploit the natural spatial correlation of the mobile nodes. Trajectory and speed are intrinsically local properties from a spatial point of view and can be easily evaluated through a distributed approach. STARE finds excellent applicability in such a scenario as it enables direct state sharing across multiple VNFs through a programmable data-plane (e.g., based on P4 switches) connecting the different eNodeBs. Contrary to the centralized solution, STARE is capable of achieving this goal without incurring scalability issues, even whenever the monitored area becomes very large.

2.4 Accelerating state replication with STARE

Motivated by the flexibility of programmable data-planes and the potential of achieving ultra-low latency, we propose STARE, a state-sharing protocol to replicate the states between VNFs, able to fill the performance gap of traditional publish-subscribe protocols by performing acceleration of the protocol via data-plane operations. The high-level communication pattern of the proposed approach is depicted in Fig. 2.4.

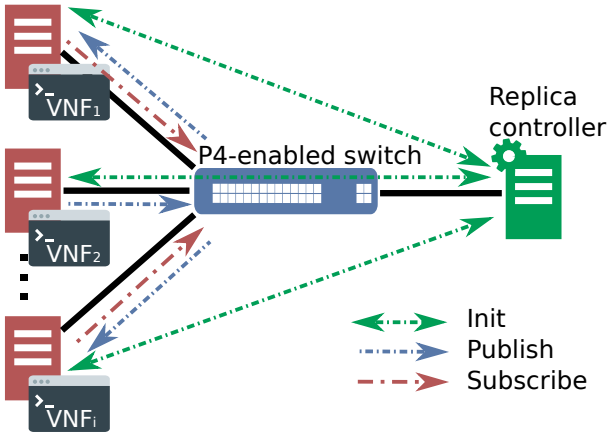


Fig. 2.4 Overview of STARE communication modes.

STARE removes the inevitable drawbacks that come from employing a software broker by delegating its functionalities to the data-plane. Switches act as distributed publish-subscribe brokers for inter-VNFs state replication while relying on a centralized controller only for the initialization phase and in case of critical events. All publish-subscribe messages are processed directly by programmable switches, thus allowing packets to always follow the shortest path to their destination without any need to be detoured to a middlebox first. At the same time, such an approach does not introduce any processing latency, since, as previously discussed, programmable switches can process packets at the line rate. While reducing communication latency, STARE also provides a means for effortless integration in existing and new VNFs. In the following section, we will discuss in detail the implementation and operation of STARE.

At its core, STARE exploits three main components to provide fast and application layer-transparent publish-subscribe service: i) a custom communication protocol that is easily interpretable both by the VNFs and by the switches, ii) a middleware running at each machine hosting VNFs that exposes clean and easily accessible communication hooks to each VNF and iii) a data-plane algorithm running inside the programmable switches responsible of performing broker's tasks. In the following section, we closely analyze the implementation and design choices behind each of the three components.

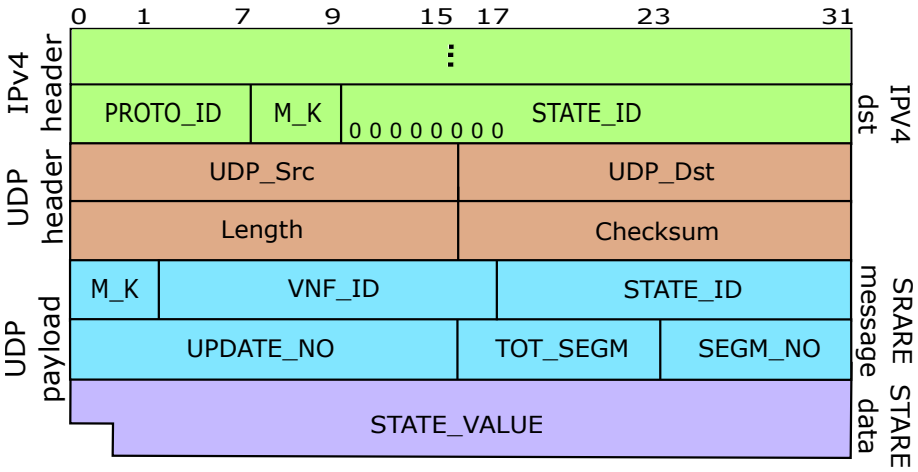


Fig. 2.5 STARE message format

Table 2.1 Coding and definition of different *Message_kind*s.

Message_kind	Definition	Message_kind	Definition
00	Publish	01	SUB-ack
10	SUB-remove	11	SUB-register

2.4.1 STARE communication protocol

STARE heavily exploits features provided by modern programmable data-planes. Notably, it relies on the possibility of defining custom protocol headers and the possibility of taking switch-local decisions based on them.

STARE uses the IPv4 *Administratively Scoped* IP Multicast range (239.0.0.0 - 239.255.255.255) as the destination IP address to disseminate the publish-subscribe messages. STARE exploits this 32-bit length field by including it within the essential data required to guarantee correct message identification and forwarding. Specifically, three custom fields are encoded by STARE within the IPv4 destination field that is summarized in Fig. 2.5: i) a constant 8 bit-long protocol_ID(`PROTO_ID`) equal to (`11101111`), which is required to correctly identify STARE packets, ii) a 2 bit-long Message_kind(`M_K`), which permits to disambiguate among different kind of messages present in STARE whose values are summarized in Table 2.1, and iii) a 22 bit-long field for state_ID(`STATE_ID`) to uniquely identify the ID of the state for which the message has been generated, corresponding to about 4 10^6 maximum number of states that could be enough for many applications. All of the above information, combined with a dedicated destination UDP port (number

```
1     // ### Example of STARE logic written in P4 language
2     apply {
3       if (!hdr.ipv4.isValid()){ // Drop non-IPv4 packets
4         drop(); return;
5       }
6       // Process the IPv4 destination field and
7       // Check if a STARE message has been received
8       ipDstCheck();
9       if (hdr.udp.isValid()) {
10        if (loc_metadata.ipDstProtoId == IP_STARE
11          && hdr.udp.dstPrt == PROTO_STARE) {
12          checkMsgKindAndInputPortMask();
13          if (local_metadata.STARE_msgKind == 0) {
14            setMcastGrp();
15            publish();
16          }
17          else if (local_metadata.STARE_msgKind == 1) {
18            sendToCpuPort();// Embedded controller-based
19            Drop();// Register-based
20          }
21          else if (local_metadata.STARE_msgKind == 2) {
22            updateUnsubscribe();
23          }
24          else if (local_metadata.STARE_msgKind == 3) {
25            updateSubscribe();
26          }
27        else {
28          drop(); return;
29        }
30      }
31      else { // process non-STARE packets
32        ipv4_lpm.apply();
33      }
34    }
35
```

Listing 2.1 STARE message processing example in P4 language

65432 in our evaluation), ensures the correct identification of STARE packets inside switches and endpoints. Since the IPv4 header is discarded at the endpoints once it is received, an application layer message is placed on top of the UDP header including: i) message_kind(`M_K`), ii) VNF_ID(`VNF_ID`), iii) state_ID(`STATE_ID`), iv) update_number(`UPDATE_NO`), v) the update total_segments(`TOT_SEGM`), and vi) segment_number(`SEGM_NO`). Although the definition of the format for each field can be arbitrarily decided by the programmer, in our implementation we employed a format represented in Fig. 2.5.

By construction, independently from the size of the network and the number of states, the message overhead is fixed (i.e., 36 bytes for each subscription and publish) and the number of exchanged messages is minimal, thanks to the use of

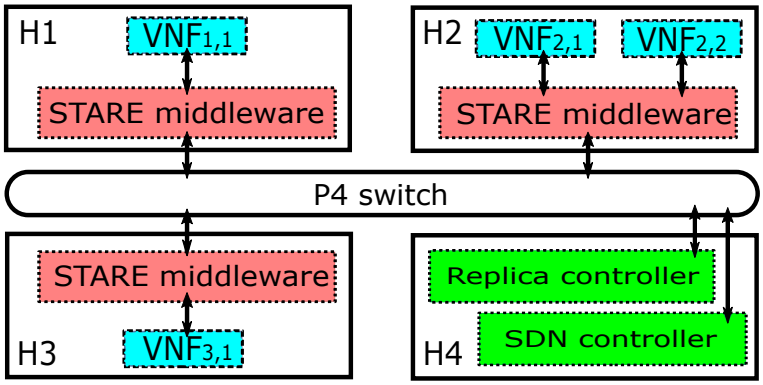


Fig. 2.6 VNFs, middleware and replica controller running in a STARE scenario.

spanning trees connecting the subscriber and publisher nodes. Notably, the use of UDP poses hard constraints on the maximum size of the states to be replicated. For this reason, STARE exploits some semi-application layer segmentation to support large states, thus appearing in a completely transparent way to the data-plane. A general sample implementation of the switch logic for it, alongside the definition of the IPv4 destination type in P4 language, is depicted in the Listing 2.1.

2.4.2 STARE middleware

As previously anticipated, STARE is transparent to the actual implementation of the core logic of each VNF. This is achieved thanks to the fact that STARE incorporates flexible middleware which is responsible for managing the overall process of state replication. Thus, as the replication protocol runs in the network, the VNFs are not directly involved in it. This approach permits the reduction of the computation overhead incurred by the management of complex replication algorithms at the VNF level and further simplifies both the development of new VNF and integration of STARE in legacy ones. The proposed middleware acts as a single endpoint of the replication scheme within each server, thus it is agnostic to the number of VMs and VNFs running on each server. The main role of the middleware is de facto to act as a proxy between local VNFs and other servers comprising the network, as shown in the example of 4 servers depicted in Fig. 2.6.

Fig. 2.7 depicts an overview of the main components of the sample middleware we implemented. The black lines are the *regular data-paths* carrying the STARE data, the red lines are the possible *Publish distribution paths* which can be either

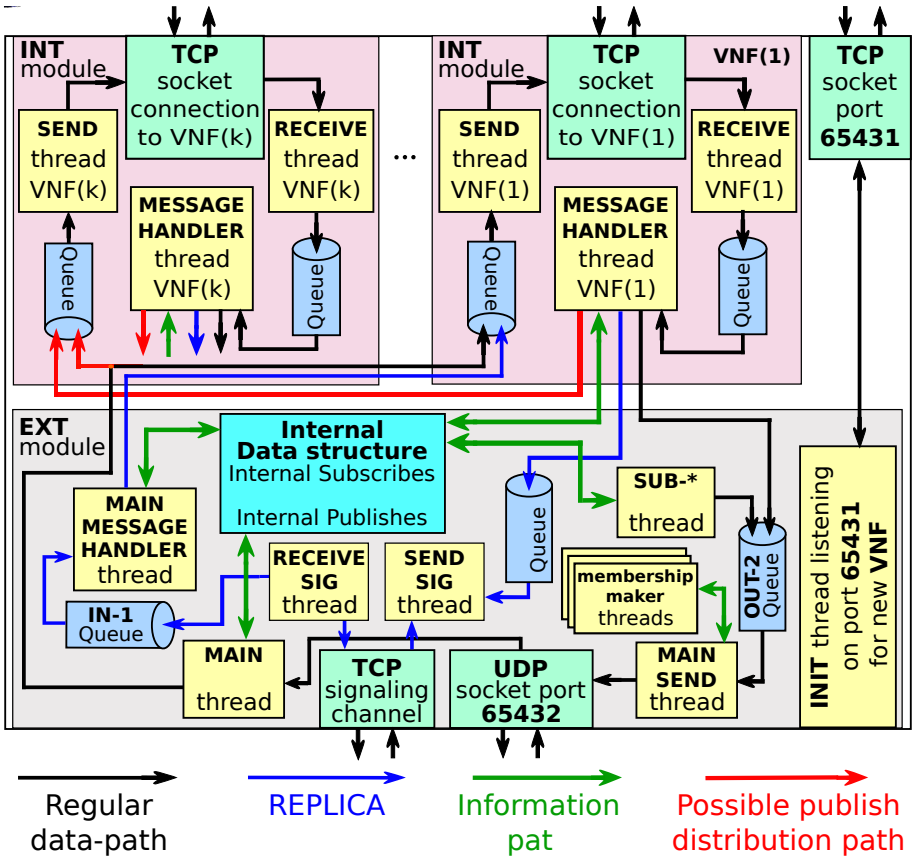


Fig. 2.7 The software architecture for STARE middleware

intra-server or inter-server, the green lines define the *Information paths* used to keep the control information updated inside the middleware, and the blue line defines the *replica controller message path*.

To achieve our final goal, we implemented a simple protocol to permit the interaction between the VNF and the STARE middleware. The architecture is made clearer by mimicking all messages exchanged inside the middleware to the behavior of the publish-subscribe protocol adopted at the network level by the programmable switches.

The middleware is composed of two macro elements: i) an external module *EXT module* which is responsible for managing communication between the server hosting the VNFs and the external network and ii) a dedicated internal module *INT module* which is responsible for managing communications between individual VNFs and the EXT module.

The main components of the EXT module can be summarized as follows:

1. *INIT thread:* The main entry point to the middleware is defined by the *INIT thread* which is responsible for opening a new socket and spawning a listener on a specific TCP port (65431 in our case). This listener provides the main means of communication between the VNFs and the STARE middleware. Whenever a new VNF is instantiated it will try to reach the middleware by binding to this particular port (we assume the port number to be known in each VNF).

2. *SIG SEND and SIG RECEIVE threads:* These processes implement a TCP connection with the replica controller and are responsible for managing all of the control information between the replica controller and the VNFs. This control information, implemented via a set of replica controller messages, includes connection initialization of each VNF and the management of the publish messages losses.

3. *MAIN MESSAGE-HANDLER thread:* It is responsible for reading and rebuilding the signaling messages queued in the incoming *IN-1 Queue*. Whenever a message from the replica controller is received, it will update the *Internal data structure* and will forward the message to the input queue of the proper VNF.

4. *INTERNAL DATA-STRUCTURE:* The internal data structure is responsible for keeping track of the publish and SUB-register requests made by the VNFs connected to the middleware. This component helps the EXT module with the distribution of the publish packets received from the network to the proper VNFs. Furthermore, whenever required, the *MESSAGE-HANDLER thread* of the INT modules can access this component for information about the local subscribers, if any, for their publishes, to avoid extra circulation of their published data using the possible *Publish distribution path*s.

5. *MAIN SEND thread:* All the outgoing messages generated by the VNFs, except the replica controller messages, are managed by the *MAIN SEND thread*. This process provides connectivity to the external network by writing data present in the *OUT-2 Queue* on a dedicated UDP socket. Depending on the message kind it will take care of specifying a proper destination for the IP layer, set the UDP layer destination port to another special port number (65432 in our case),

and send the message to the network. Whenever a *SUB-register* message is sent to the network, the employed IPv4 address is recorded in the *IP-multicast membership* module of the OS. This ensures that the OS will be able to deliver the published messages related to the sub-register to the STARE middleware. A similar procedure will be done to remove the registration in the membership module of the OS in the event of a SUB-remove message. To address the current challenges, we integrate information from all of the DC networks. In the case of receiving a subscription message, the actual IP destination is informed to the membership module, which is updated with a new IP-multicast group whenever a new subscription message is received internally from a VNF.

6. *MEMBERSHIP-MAKER threads:* These threads permit overcoming the limitations of the UNIX-based OSs for multicast-group memberships, which have a limited membership size. Following an attempt to send a new message, an instance of such a thread is spawned whenever the IP-multicast registration in the IP-multicast membership module of the OS fails.

7. *SUB-* RESEND thread:* As a primitive practice for ensuring the delivery of the SUB-register and SUB-remove messages to the network, this process emulates an ARQ protocol. It will periodically check the *Internal data structure* for the SUB-register messages that have been sent but that did not receive a *SUB-ack* message from the network. It will then add them to the *OUT-2 Queue* to be resent. It loosely ensures the delivery of the subscription messages to at least one of the switches participating in our protocol.

8. *MAIN RECEIVE thread:* Similarly to the *MAIN SEND thread*, the *MAIN RECEIVE thread* is responsible for managing all of the incoming messages except for the *replica controller* messages. Upon message reception, this thread will forward the message to the proper *INT module* according to the information stored in the *Internal Data Structure*. Additionally, if the processed message is an *SUB-ack message*, the thread will update the information associated with the waiting list of the non-acknowledged *SUB-remove* or *SUB-register* messages. The list will then be checked by the *SUB-* RESEND thread* periodically.

For each VNF the STARE middleware creates a dedicated instance of an INT module that provides message connectivity to the rest of the middleware. The architecture of such a module closely mimics that of the EXT module: two different threads (SEND and RECEIVE threads) are responsible for reading and writing from/on a dedicated socket which is continuously listened to by the VNF, thus providing the last hop to the actual implementation of the VNFs. Analogously to the EXT module, input and output queues are used to buffer messages with the latter being read by a dedicated message handler which is responsible for forwarding the message to the EXT output queue.

2.4.3 STARE replica controller

Although STARE is highly decoupled from centralized entities, it still requires minor intervention from a central controller. In particular, during network setup, a *replica controller* provides unique identifiers for each VNF, which will be later exploited by STARE to avoid ambiguity in the communication protocol. Furthermore, these so-called globally unique IDs (i.e., an ID that is unique among all VNFs participating in each implementation of this scenario) are assigned to each state in an analogous way to what already happens in classic publish-subscribe schemes. Those IDs permit discrimination among different publish-subscribe messages and, as we will show later, are exploited to perform message forwarding in the network. Regarding the scalability of the proposed solution, theoretically it supports up to 4 million different unique states, limited by the length of the IPv4 destination field. The same approaches used for IPv4 scalability could be applied to our solution. However, adapting our solution to the IPv6 header can significantly increase scalability.. Additionally, there is a need to keep a backlog of published messages on the network, which must be used to recover from message losses. The replica controller may be implemented as a dedicated server or as an integrated process running inside the SDN controller.

Also, To address the risk of a single point of failure, particularly regarding the replica controller, it is possible to implement a standard replica scheme for fault tolerance. This approach involves having multiple copies (replicas) of critical components, like the replica controller, to ensure redundancy and minimize the impact of potential failures. It's crucial to keep these replicas synchronized to maintain consistency across the system. Finally, since using a master node for system coordination introduces centralization, we need robust failover mechanisms

and redundancy. Techniques like leader election algorithms or standby replicas are essential for ensuring continuous operation and fault tolerance, even in the face of failures. By employing these strategies, the system can effectively mitigate the risks associated with single points of failure, thereby enhancing its overall reliability and resilience in distributed environments.

2.4.4 Message loss recovery

In the case of a loss of a publish message, the event can be easily detected inside the subscriber VNFs. This is performed by checking the application layer message information regarding the continuity of the segment number in relation to the state update sequence number and the total segments of the update. Whenever a VNF detects a loss, it will notify the replica controller through a RECOVER message containing the related information from the application message layer, in turn, and the replica controller will provide the lost message by looking for it in the stored backlog. The use of the replica controller for such tasks significantly reduces the resource overhead on each server running STARE. Indeed, all of the backlogs of the published messages are stored at the controller instead of being distributed on every single server. This approach reduces the overall memory requirement for the deployed servers and considerably simplifies the design of the STARE middleware. To keep the memory utilization low, the backlog must be periodically truncated. This can be achieved by actively querying single VNFs for the maximum segment numbers received so far for any given state. This information can then be easily exploited to perform backlog truncation by considering the minimum among the received segment numbers and by truncating the backlog up to that point.

We address the recovery of the SUB-remove and SUB-register messages on both the: i) VNF-to-switch/switch-to-VNF and ii) switch-to-switch segments. As explained in Sec. 2.4.2, we delegate the reliable delivery of these messages to the first P4 switch (VNF to switch segment) through a simple ARQ algorithm and by employing the *SUB-* RESEND* process for both solutions. Employing such a simple ARQ protocol implementation inside a lightweight embedded controller on each P4 switch effectively permits support for reliable delivery of these messages between switches.

Table 2.2 Example of a state forwarding table and the corresponding match-action rules for messages forwarding

State forwarding table		Port forwarding table	
State_ID | i	register[i]	**Match** Dst. bitmask	**Action** Forward on ports
0	00011110	00011110	{4,5,6,7}
1	10001010	10001010	{1,5,7}
2	10000000	10000000	{1}

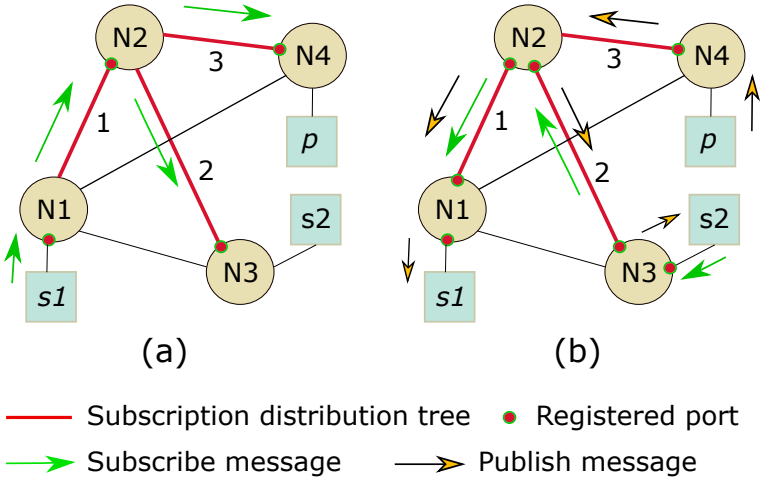


Fig. 2.8 The general subscription distribution tree and the possible paths traversed by the subscription messages.

2.5 Implementation with P4-enabled dataplanes

Concerning the data-plane implementation, we consider P4-enabled devices [34] as the main candidate for the implementation of STARE functionalities. This, however, does not limit the generality of our approach since most of the modern programmable switch architectures, as we discussed in Sec. 2.3.2, offer similar capabilities. Nevertheless, in the case of computationally or resource-limited devices, some of the requirements of the STARE data-plane implementation may be unfeasible to implement. For this reason, we propose two alternative solutions, the first (denoted as "Register-based") being a pure data-plane implementation, while the second (denoted as "Embedded controller-based") being a hybrid data-plane/CPU implementation.

For further discussion, we assume that the SDN controller has already set up a subscription distribution tree spanning all the switches to which the servers running the VNFs are connected. An example of this tree is highlighted with the red lines in Figs. 2.8(a) and 2.8(b). The subscribe (i.e., SUB-register and SUB-remove) and publish messages will only be forwarded on this tree or a subset of it. Details will be provided in the following sections.

2.5.1 Register-based implementation with P4

A pure data-plane implementation requires a considerable amount of stateful resources inside the switch to support the broker functionalities. Analogously to a traditional software broker, the switch must be able to correctly process three main kinds of messages: the *subscribe*, the *unsubscribe*, and the *publish* messages. To do so, switches must keep track of a data structure mapping particular state_IDs specified in the Sub-register messages to a set of output ports on which VNFs requiring updates of that particular state are reachable. This is achieved thanks to a dedicated data structure, namely the *state forwarding table* implemented with an array of registers. Analogously to a hashmap, given a state_ID x, the state forwarding table stores a bitmask B_x of output ports inside the register indexed by index x. Noteworthy, such implementation implies that the number of bits to store each bitmask is equal to the number of ports in the switch. Table 2.2 depicts an example of a state forwarding table for an 8-port switch storing 3 state_IDs alongside the corresponding match-action port forwarding table responsible for multicasting the message on given ports.

Notably, although the distribution tree spans all of the available endpoints, to reduce the total data overhead in the network, only a subset of the links is used each time. Each switch forwards publish messages on a given port only if an active subscriber is reachable through that port (yellow line in Fig. 2.8a). Similarly, whenever a subscribe message floods the distribution tree, all ports leading to an active publisher are ignored (green line in Fig. 2.8a) since the publish connectivity is already guaranteed up to that point.

We report here the pseudocode of the STARE procedures for handling the Sub-register and Sub-remove messages in the P4 switches demonstrated in Fig.2.9. Through these procedures, the switch processes the messages and updates two flags,

"drop" and "publish". The "drop" flag represents whether the message should be dropped or forwarded, while the "publish" flag value determines whether the message should be forwarded by the publishing rules (similar to what happens in the case of a publish message) or simply be flooded on the subscription distribution tree. Those procedures permit a reduction of the total data overhead in the network by avoiding flooding the SUB-register and Sub-remove messages on the whole distribution tree. Those procedures are run locally at each switch and operate on three main inputs: i) the state_ID x carried by each packet; ii) the arrival port p of the packet; and iii) the bitmask B_x describing the forwarding rules for publish-subscribe messages. At the end of the procedures, each message is either dropped or forwarded following the distribution tree by respectively setting either of the two flags "drop" or "publish" to *True*. The "publish" flag value is used if the message should not be dropped (i.e., the "drop" flag is set to *false)* and the default value for both of the flags is initialized to *false*. The message kind is checked by the switch by parsing the STARE header.

Handling of SUB-register messages

In the case of a SUB-register message for state S_x arriving on port p (line 1), the switch checks whether p is already registered in B_x. If p is not registered, registration is found (line 3), the switch will register p (line 6) and check whether other prior subscriptions from other ports exist (line 4). If no prior registrations exist, the switch will flood the distribution tree (except for p) to ensure that the subscription message reaches the related publishers, thus activating the distribution tree up to that switch. This is shown by the green arrows in Fig. 2.8(a). If, instead, other registrations are found (line 7), the switch will treat the message as redundant and drop it (line 8). If, instead, prior subscriptions were found in B_x (scenario in Fig. 2.8(b)), the publish flag will be set to *True* (line 5). This will result in the SUB-register message being forwarded on a publish message path (red links) from the previously registered ports. This procedure will force the SUB-register message to traverse only the required links and avoid flooding the entire network.

Such a mechanism permits the dynamic build of a global port mapping for all of the state_IDs present in the network. Once built, this mapping permits switches to unambiguously process the publish messages by simply indexing the bitmasks by the value of state_ID contained in the messages and forwarding them to the corresponding ports.

Input: x: state_ID carried by the message
Input: B_x: bitmask of port-registration for state S_x, equal to Register[x] for register-based approach and equal to L2_publish[x] for embedded controller-based approach
Input: p: input port of the received message

```
 1: procedure UPDATESUBSCRIBE(x,B_x,p)
 2:     Publish ← False, Drop ← False                    ▷ Initializing the flags
 3:     if bit(B_x, p) = 0 then                          ▷ Check if the port is NOT in the bitmask
 4:         if B_x ≠ 0 then                              ▷ Prior subscribed ports exists on the switch
 5:             Publish ← True                           ▷ Publish Subscribe message
 6:         B_x ← or(B_x, p)                             ▷ Update subscription bitmask with incoming port p
 7:     else
 8:         Drop ← True                                  ▷ No forward anymore
 9: procedure UPDATEUNSUBSCRIBE(x,B_x,p)
10:     Publish ← False, Drop ← False                    ▷ Initializing the flags
11:     if bit(B_x, p) = 1 then                          ▷ Check if the port is in the bitmask
12:         B_x ← or(B_x,not(p))                         ▷ Removing the port from subscription bitmask
13:         if B_x ≠ 0 then                              ▷ Still subscribed port/s exists on the switch
14:             Drop ← True                              ▷ Do not forward the Unsubscribe message
15:     else
16:         Drop ← True                                  ▷ No forward anymore
```

Fig. 2.9 Pseudocode of STARE procedures for subscribing and unsubscribing

Handling of SUB-remove messages

Analogous processing occurs for a SUB-remove message, but instead, B_x is updated in such a way as to remove the original input port p of the received message, as depicted in the procedure *"UpdateUnsubscribe"* (line 9). Upon packet reception, the switch will check whether p is registered in B_x (line 11). Based on the outcome, it will then proceed by unregistering p from B_x if present (line 12) or by considering that packet as redundant and dropping it (line 16). If, after removing p from B_x, B_x still contains other ports, the unsubscribe message is dropped (line 14) and the distribution tree is maintained up to that point. On the contrary, an empty B_x would mean that there are no active subscribers connected to that switch. In such a case, the unsubscribe message floods the distribution tree (Publish and Drop flags set to *False*) to destroy the now-inactive branch of the tree up to the first switch, leading to more than one subscriber.

Finally, to support advanced signaling, all other kinds of messages used in STARE protocols (e.g., Recover message, etc.) are sent to the port corresponding to the replica controller. A partial implementation of this solution with P4 is available on `github` in [54].

2.5.2 Embedded-controller-based implementation with P4

This second solution is motivated by the fact that the scarce availability of registers can pose hard constraints on the feasibility of the register-based solution due to the impossibility of storing the state forwarding table. Yet, the match-action tables are plentiful even in low-end devices which provides an alternative for building the state forwarding table. Nevertheless, commercial switches are typically equipped with an onboard CPU that can operate as a local embedded controller, although without the global visibility of the SDN controller, but at a much-reduced latency. To build the state forwarding table, The switch exploits the arrival port_ID of the Sub-register messages and uses the state_ID as a key in the state forwarding table to retrieve and eventually update the current destination bitmask.

This can be done by using a key-value structure, mapping the state_ID as the key to a list of forwarding ports of the message. This implies that the possible received subscribe-ack message will be sent to this controller to remove the input port of this message from the mentioned list. The SUB-ack message will be dropped at the controller. Furthermore, the controller sends back a copy of the SUB-register or SUB-remove message to the switch adding in the *Packet-out* header the original input port of the message, the *Publish*, and the *Drop* flags as well, to inform the switch about the decision over dropping the message or how to forward it to the proper ports if should not be dropped. In any case, if message forwarding is needed, the *Packet-out* header will be removed before forwarding the message by the switch.

Unfortunately, the current implementation of the P4 programming abstraction does not provide a means of direct match-action table manipulation, thus relying on the controller for such a task. The Embedded controller-based implementation of STARE, exploits an Embedded-controller using the onboard CPU, which, analogously to the register-based solution, modifies the entries in the switch match-action table with respect to the procedures reported in Fig. 2.9. In our specific implementation, this is achieved by exploiting the P4-Runtime API [55].

Anytime a Sub-register or Sub-remove message arrives at the switch, it is sent to the Embedded-controller through a *Packet-In* message, whose header contains the incoming port of the packet. The controller will follow the exact procedures of Fig. 2.9, and if needed, updates its local version of the state forwarding table and send the corresponding instructions to update the switch *L2_publish* match-action

table as well. The switch then acknowledges the configuration to the embedded controller.

Based on the procedures reported in Fig. 2.9, if the resulted value for the "drop" flag becomes true, the controller will drop the message and will not send it back to the switch. In the other case, if the resulted value for the "drop" flag is false, it is needed to forward a copy of the message to other switches, and the controller will send the original message to the switch through a *Packet-In*, whose header contains the resulted "publish" flag value and the original input port of the message. This informs the switch about the decision on how to forward the message to the proper ports. In any case, if message forwarding is needed, the *Packet-out* header will be removed before forwarding the message by the switch.

Based on the *Packet_out* header flags defined by the controller, the switch will check the "publish" flag value and after removing the *Packet_out* header, if the value is false, the switch will flood the message to the ports related to the subscription distribution tree. But, If the mentioned value is true, the switch will forward copies of the message, to the ports subscribed for this state_ID, except to the original input port of the message. This is similar to when a Publish message arrives at the switch.

The embedded controller can keep track of the delivery for Sub-register and Sub-remove messages to the next step P4 switches by following a simple ARQ protocol, like *stop-and-wait* between adjacent switches as well. The state_ID is used to identify uniquely the message for the ARQ. In this case, the controller can update the subscription's delivery status to each of the next switches as undelivered, waiting for a reply.

2.5.3 Comparison among the two implementations

The register-based implementation relies on a static number of registers, defined at compilation time. This implies that whenever the number of different states present in the publish-subscribe scheme exceeds the available ones, it is necessary to perform switch reconfiguration. While, in theory, this reconfiguration can be operated live on specialized hardware, in practice it may introduce a transient service outage. On the other hand, the embedded controller-based solution can overcome this issue, as match/action tables are plentiful in modern switches, but at the cost of increased complexity and new rule installation latency [56].

Indeed, the processing time of STARE subscribe messages for the *Embedded controller-based* solution can be slightly higher than that for the *Register-based* solution due to the time required to modify the match-action tables. More on that, the interaction required with the embedded controller can introduce a very small latency, and such latency is still considerably lower in comparison with the interaction with a remote controller. Due to its general-purpose nature, the embedded controller can implement additional capabilities, such as secure communications, e.g., by supporting MACsec to protect network links between P4-based SDN switches, as described in [57]. The advantages of the local controller-based solution over the register-based one are: i) the table size, in terms of stored states, is independent of the width used to represent the state_ID, so eliminates the need for a recompile and reinstall the P4 STARE program again; ii) the state_IDs can be chosen in an arbitrary range due to the similarity of a match-action table to a hash table, thus giving a higher level of flexibility; while in the register-based implementation, the state_IDs are bound to the register indexes and will need a hash mechanism implementation mapping the real state_IDs to the 0 to $n-1$ (where n is the number of registers defined in the P4 switch) the switches or in the middleware.

On the contrary, in the register-based application, the STARE-related table size is always equal to the number of defined registers for this purpose, independently from the number of recorded states. Notably, the maximum number of registers depends on the particular hardware architecture and is typically limited, whereas the size of the match-action table depends on the actual use of the available memories. In terms of performance, the processing time of STARE messages of the local controller solution can be a little higher with respect to the register-based solution due to the time required to interact with the embedded controller; nonetheless, this is considerably lower in comparison with the interaction with a remote controller. This may introduce a higher degree of unpredictability in comparison with a pure hardware-based solution and may deteriorate performance when match-action tables are close to their maximum capacity [56]. It is worth highlighting that both schemes do not require the interaction of the switch and the VNF with a software broker or with the SDN controller, since all the related operations are offloaded directly to the data-plane. In conclusion, the register-based is suitable for the cases that have a limited number of states on smaller scales but they need a guaranteed range of ultra-low latency in sharing the states. On the other hand, for a more scalable solution

with a large number of states that can tolerate some small degree of unpredictability in latencies, the embedded-controller-based solution is the perfect choice.

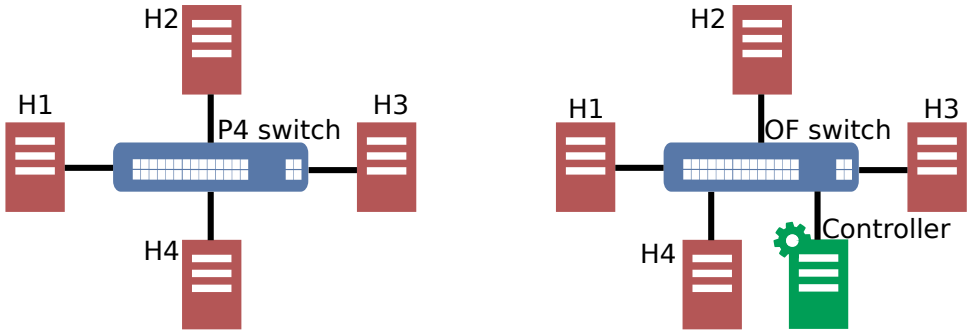
2.6 Experimental evaluation

We implemented the STARE framework and performed a set of experiments to compare it with alternative solutions. We used Mininet [58], a network emulator that provides software models for vanilla SDN switches and P4 switches, and BMV2 [59] software switch to run the experiments based on P4 switches.

First, we tailored our experiments to the 5G use case described in Sec. 2.3.3, where a VNF for mobility tracking is available at each edge cloud. Fig. 2.10a depicts the testbed topology, which includes 4 VNFs and one P4 switch. The VNFs are emulated through Mininet "hosts" and each of them runs the STARE middleware. The replica controller runs in H4, and STARE middleware runs in H1, H2, and H3, while to mimic a more heterogeneous scenario, we consider one VNF running on H1 and H3 and two VNFs running on H2. The VNFs track the users' mobility according to the considered 5G use case. To measure the resources required in the P4 switches and compare the different solutions, we evaluated the memory occupancy in terms of the Resident Set Size (RSS) of the process running the virtual switch.

We run an alternative scenario based on a pure OpenFlow approach. Ryu [60] is the SDN controller interacting with the OpenFlow switches and P4 switches, chosen thanks to its simplicity of deployment and its wide support of OpenFlow standards. The STARE middleware has been developed in Python and implemented through the standard Python socket library. For the register-based solution, we have pre-allocated the switch resources by defining 2048 registers inside the P4 program at the compilation time.

This metric allows us to evaluate how many resources, in terms of memory, are associated with each process. To mimic the 5G EVE platform, each VNF generates and receives the feed of the WiFi sensor in a JSON format every two minutes and updates a local table mapping each listened MAC address and the corresponding timestamp to a WiFi sensor ID. To accurately emulate the arrival process of messages generated by the WiFi scanners, every 20 seconds, a report message is generated by each scanner. Each message carries a list of MAC addresses (corresponding to the



(a) STARE scenario (b) Pure OpenFlow scenario

Fig. 2.10 Topologies employed in the experimental testbed.

detected mobile devices) with the corresponding timestamps, both encoded in JSON format. The number of detected MAC addresses varies randomly between 10 and 50 to mimic the experimental conditions observed in the 5G EVE use case during rush hours in the area between our university and the main train station of our city. Anytime the table is updated (typically with multiple entries added to the table), the new state must be replicated across all the other VNFs. To reduce communication overhead, updates are incremental, so only the new entries are sent within each Publish message.

As a term of comparison, we considered a pure OpenFlow implementation for the adopted publish-subscribe protocol, based on OpenFlow 1.3. In particular, we use Open vSwitch [61] software switch, as the replacement of the P4 software switch (BMV2), for the experiments based on standard OpenFlow switches. Similar to the previous test-bed with P4, we consider one VNF running on H1 and H3 and two VNFs running on H2 and the replica controller is placed in H4. A simple topology for this scenario is shown in Fig. 2.10b.

In such a scenario the SDN controller takes the responsibility of storing the state forwarding table, which forces the switch to interact with the SDN controller anytime a new subscription occurs. Thus, this scenario is representative of a centralized solution alternative to STARE. Also, the initialization phase, required to assign the VNFs with their corresponding unique IDs and to know the state IDs used to publish and subscribe, is managed centrally by the SDN controller. All the subscription messages received by a switch are sent to the SDN controller, which in turn programs the switch flow tables to associate the corresponding incoming port of the switch as the forwarding port for all the corresponding publish messages. This permits

avoiding additional interaction with the SDN controller whenever an already-seen publish message arrives since the packet will be directly sent to all the switch ports corresponding to subscribing VNFs.

We do not report any results regarding latency measurements, since Mininet is an emulated network environment and does not allow us to evaluate meaningful statistics regarding the delays. Nevertheless, it is worth recalling that, by design, P4 switches perform packet processing at the line rate, thus we expect minimum processing delays (which depends on the internal hardware pipeline and the organization of the programmable matching tables of the P4 switch). This implies that we expect the gain of STARE in terms of latency to be significant in realistic implementations, and in large networks connecting the different VNF instances, STARE outperforms other centralized solutions in terms of latency. This is because STARE removes the necessity of the switch to interact with a remote controller (replica or SDN) anytime a publish message is received.

2.6.1 Resource consumption

We compare the performance of the STARE with the solution based on OpenFlow switches in terms of internal resource consumption within the switch. The next three figures illustrate the overall memory usage of the software-switch processes.

Registered-based with P4 solution

In the register-based solution, the memory occupied by the software switch remains constant regardless of the operations within the registers. Therefore, we equated the memory needed by a rule to that of one register in our setup. Instead of writing rules, we compiled our P4 program for varying numbers of registers multiple times to determine the memory occupied by the software switch and its confidence interval. Fig. 2.11 reports the average RSS memory, including the 95% confidence intervals and a linear regression over the average values. The achieved accuracy is very high since the relative width of the confidence interval is around 0.2%. From the figure, the average amount of memory is 119.0 bytes for each installed rule.

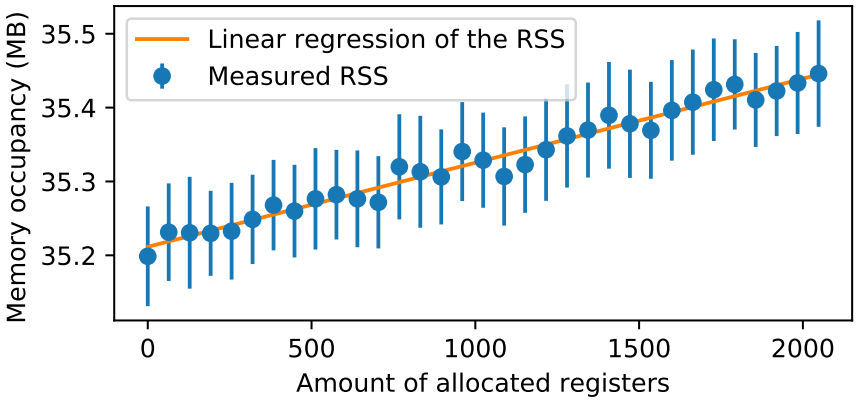


Fig. 2.11 Register-based solution.

Embedded controller-based with P4 solution

In the adopted match-action table, we evaluate the width of the key. In the flow match-action table, we use an exact matching on the IP address, so 32 bits are needed, and the corresponding action is coded as a 32-bit number representing the multicast group, thus we can expect a minimum of 64 bits for each rule. The result of the measurement is shown in Fig. 2.12, which shows that the memory occupancy increases linearly with the number of installed rules and that the average memory occupancy is 184 bytes per installed rule. It's important to note that, for the Embedded controller-based solution, we did not report a confidence interval. This is because the increase in memory occupied by one rule remained constant throughout testing, irrespective of the initial memory occupied by the software switch. For simplicity, we reported one of the test results.

OpenFlow-based solution

We evaluated the RSS of the OVSK process, and the measurements were collected by the remote SDN controller. The result of the measurements is depicted in Fig. 2.13, which shows a step-wise increasing function. Such behavior is due to the internal memory allocation scheme, which employs a batch allocation process in memory. The average occupancy is 586 bytes per installed rule.

In essence, Table 2.3 outlines the average memory occupancy per rule across the three mentioned scenarios. This calculation involves subtracting the initial

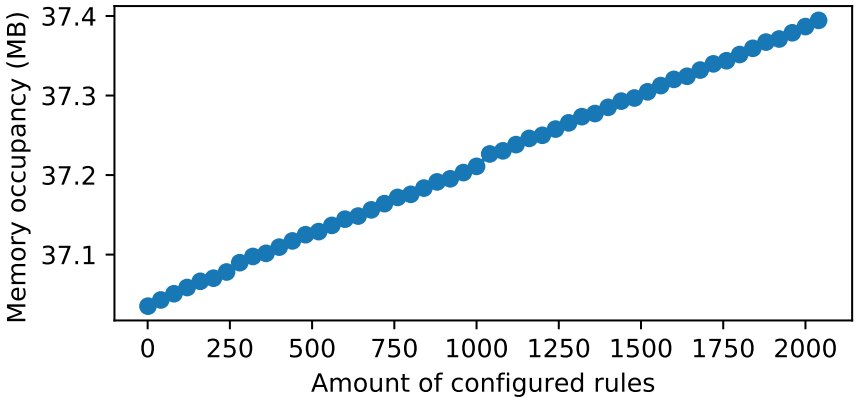


Fig. 2.12 Embedded controller-based solution.

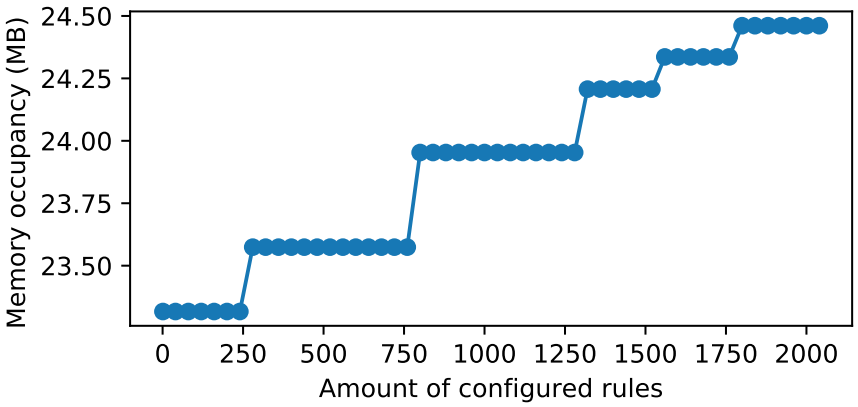


Fig. 2.13 Remote Controller-based solution.

Table 2.3 Average memory occupancy per rule for the three scenarios.

Scenario	Average memory per rule
Register-based P4 solution	119 bytes
Embedded-controller P4 solution	184 bytes
OpenFlow-based solution	586 bytes

measurement values of software-switch memory occupancy from the final ones and then dividing the result by the number of rules. The most efficient solution in terms of memory is based on a register-based STARE implementation, whereas the least efficient solution is the one based on a standard OpenFlow switch. The differences are due to the internal memory management process internal to the BMV2 software

switch (for both P4-based solutions) and the Open vSwitch software switch (for the OpenFlow-based solution).

2.6.2 Network load and traffic overhead

We considered a full tree graph with three generations after the parent node, where at each generation there exist four nodes, and the clients are connected to the graph leaves. The graph comprises 84 links, and each of the nodes is considered a switch. Then, we compare the register-based solution of STARE in terms of network traffic in terms of the number of packets, protocol overhead, and the number of transmitted bytes with the following five alternative approaches.

- **Unicast**: In the *Unicast* approach, the publisher sends one publish message individually to each subscriber.
- **Broadcast**: In the *Broadcast* approach, the publisher sends one publish message in the whole distribution tree.
- **Application-Layer Multicast (ALM)**: ALM is based on a broker for each switch, which is aware of the switch ports that are on the distribution tree towards the subscribers. This broker will receive the Publish packet from the switch and return the packet to the switch with a header containing the corresponding multicast group.
- **OpenFlow Multicast (OFM)** [43]: OFM is an OpenFlow-based approach that uses an IP address and a UDP port number for addressing a multicast tree stored on the switches. We assume that all these OFM trees have already been configured through related flow rules from the SDN controller.
- **P4 source routing**: This approach was recently proposed in [42] as a centralized approach for publish-subscribe based on source-routing multicast implemented in P4. It uses a stack of headers added to the MAC header containing the switch identifiers of the path to the subscribers and the corresponding multicast address for each switch. The SDN controller is responsible for receiving the subscription and informing the publisher on how to generate this header stack.

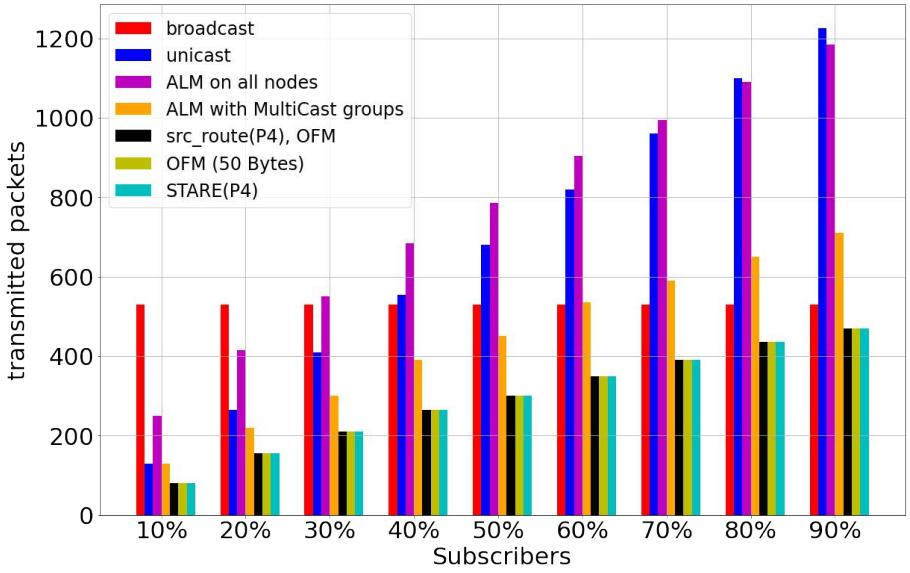


Fig. 2.14 Comparing the transmitted packets of the STARE with other approaches.

Transmitted packets

Fig. 2.14 demonstrates the number of packets transferred in the network for each approach concerning the number of subscribers. The minimum number of transmitted packets has mutually belonged to STARE, source-route P4-pubsub, and OFM. During the distribution phase, none of these approaches uses the links except for forwarding the published notification to the subscribers. The results of the ALM approaches are worse because they need to forward the published notifications to the software broker and receive the packet duplicates from them, while the ALM with MC-groups performs better as it sends only one duplicate back to the switch rather than multiple packets in the ALM on all nodes, and it has fewer brokers connected to the switches as well. The ALM with MC-groups is the closest in behavior to the STARE; it only differs in the fact that the software broker in the STARE has been merged into the P4 switches. The unicast and the broadcast perform well when the subscribers are a few or almost all, respectively. STARE, source-route P4-pubsub, and OFM converge to the broadcast when more of the subscribers are involved and converge to the unicast when the involved subscribers are smaller.

Table 2.4 Network messages to deliver a publish message.

Approach	In-net(B)	Out-net(B)	In-net(W)	Out-net(W)
Unicast	32	17	96	17
Broadcast	84	85	84	85
ALM	20	59	26	67
OFM	20	17	26	17
P4 source routing	20	17	26	17
STARE	20	17	26	17

B: Best case W: Worst case

Traffic overhead

The traffic overhead depends on the network topology. In our evaluation, we considered a symmetric tree topology connecting the switches with 63 leaves, distributed in 4 layers (comprising root and leaves). All nodes (except the leaves) have 4 children. The total number of links is 84. The subscribers are connected to the leaf switches, while the publisher is connected to an arbitrary switch through a dedicated link. We calculated the best and the worst case with respect to selecting any set of leaves for the subscribers and the position of the publisher. Since subscriptions are transient and are defined only in publish-subscribe schemes, for a fair comparison, we consider only the publish phase since it is well-defined for all of the considered approaches. Thus, we evaluate the traffic in terms of "in-net" traffic, i.e., across links in the considered topology, and in terms of "out-net" traffic, i.e., across the links that will connect the publisher and subscribers VNFs (not considered in the above topology).

Table 2.4 depicts the in-net and out-net messages needed for delivering a Publish message to 16 subscribers in the network for the different approaches. The minimum number of in-net traffic is achieved by STARE, P4 source-routing, and OFM since each link in the distribution tree is used just once for each Publish message. The results of ALM are worse since it requires each switch to forward the Publish messages to the software broker and to receive the corresponding multicast group from it. ALM is similar to STARE, except for the fact that in STARE the switch does not need an external interaction. Unicast is inefficient since the same publish message may be sent multiple times on the same link, and it would be a reasonable solution for a few subscribers. Instead, broadcast results are the least inefficient since each publish message is flooded across the whole distribution tree, independently

from the subscribers. Thus, it could be used only when the number of subscribers is maximum. Although it is not depicted in the table, the amount of traffic generated by STARE, P4 source-routing, and OFM converges to the amount of traffic generated by broadcast whenever all nodes act as subscribers and it converges to the unicast case whenever there is only one subscriber.

After considering the traffic overhead in terms of the number of packets, we consider the traffic overhead in terms of bytes, for each protocol message. All of the considered approaches adopt standard protocol headers at layers 2, 3, and 4, except for the P4 source routing. [42] adopts a stack of sorted labels that enable simple processing at each switch, yet is it required to be transmitted across links even if not required. Each label is 3 bytes long and includes the switch identifier and the bitmask with the local destination ports. In the considered scenario with 16 subscribers at the leaves of the topology, it can be shown that, in the best case, the stack comprises between 16 and 20 labels, depending on the level in the topology, with an average of 18 labels per publish message (54 bytes total). In the worst case, The stack comprises between 16 and 24 labels, depending on the level in the topology, with an average of 18.75 labels (56.25 bytes). Note that such overhead may not be negligible in the case of updates corresponding to "small" states, e.g., integer counters.

Transmitted bytes

Figs. 2.15-2.16 show the comparison results for 50-bytes-length published notifications, where all the other approaches are implemented using the MAC header and IP/UDP header, respectively. For the IP/UDP implementation of the other approaches, it is obvious that STARE is outperforming all of them except for the case of OFM, in which the amount of traffic is slightly lower than our solution. We did not implement the source-route P4-pubsub approach with the IP/UDP header based on its authors' implementation goals. Even in the case of implementing other approaches with only MAC headers, STARE with IP/UDP and fixed 28 bytes more headers in each packet is outperforming the unicast and ALM on all nodes for any number of participants. It is better than broadcast for the subscriber numbers of less than half of the hosts, and almost equal to ALM with MC-groups in all the cases. The results of the source-route P4-pubsub and OFM are slightly better than STARE in a small number of subscribers, and the gap becomes greater as the number of subscribers increases. This is due to shrinking the header of the notifications alongside the path to subscribers and the

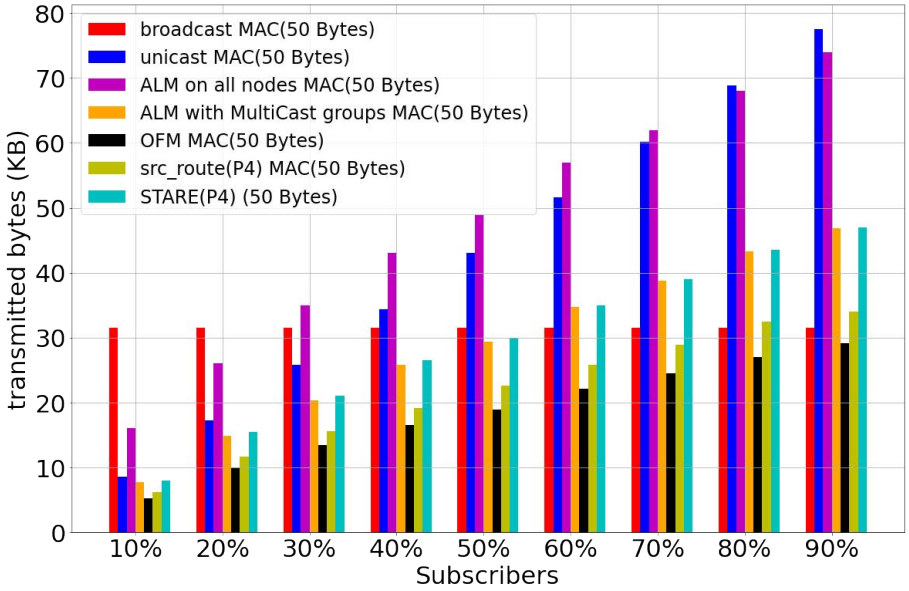


Fig. 2.15 Comparing the transmitted bytes of the STARE with other approaches using MAC.

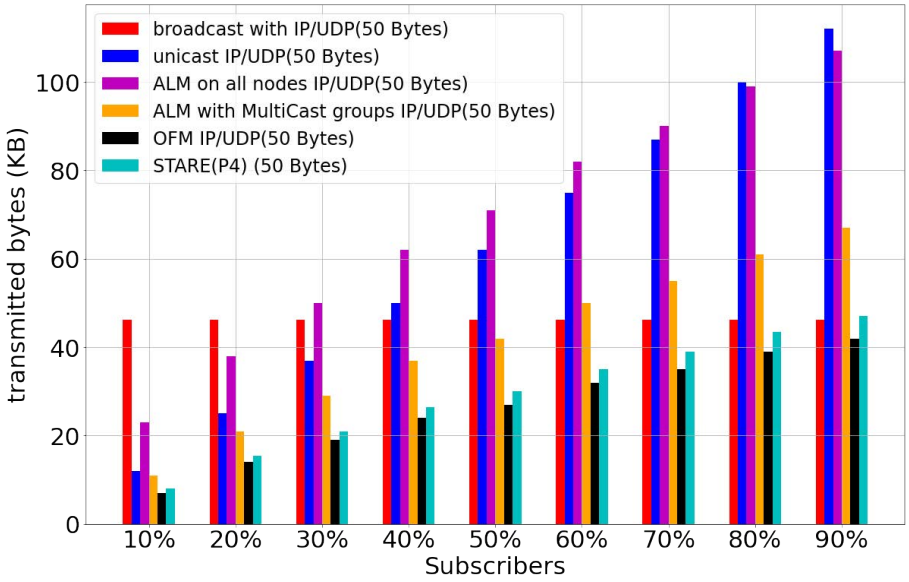


Fig. 2.16 Comparing the transmitted bytes of the STARE with other approaches using IP/UDP.

constant small MAC header in the case of the source-route P4-pubsub approach and OFM approach, respectively.

2.7 Discussion

In this work, we propose STARE, a low-latency publish-subscribe architecture for NFV and SDN-enabled networks. STARE aims at achieving fast state replication among different VNFs in 5G networks. To do so, it exploits recent advances in the field of programmable data-planes by offloading the functionalities of a traditional publish-subscribe broker to the programmable switches. At the same time, STARE provides easily deployable middleware that permits rapid integration of new and existing VNFs by exposing simple APIs that can be accessed easily inside the source code of each VNF. We validate our approach by employing an emulated, yet realistic, testbed network with both P4-enabled switches and vanilla SDN based on OpenFlow switches. We also compared our solution with some other approaches for the distribution of the published notifications using different technologies.

Our experiments show that using STARE, in combination with P4-enabled switches, leads to completely homogeneous traffic in the network by using fixed-length headers that use the least possible number of packets for delivering the published notifications to the subscribers. It also remarks that, as a trade-off, by sending in a total of a few larger amounts of bytes than one or two approaches, the approach will not need to interact with a software broker nor map the total path between publishers and subscribers inside the packet, which leads to a complete decoupling of the publishers and subscribers. At last, it incurs a smaller overhead in terms of memory in comparison with the traditional approaches based on vanilla SDN. Furthermore, thanks to the limited interaction with a centralized controller and removal software brokers, STARE is expected to lead to significantly lower state replication latency compared to the traditional publish-subscribe schemes. The deployment of STARE on a real testbed, along with a detailed evaluation of the gain in terms of latency, is left for future work.

Chapter 3

Optimal endorsement for network-wide distributed blockchains

A part of the work presented in this chapter has been published in [62] and is reported here for the reader's convenience:

- I. Lotfimahyari and P. Giaccone, "Optimal Endorsement for Network-Wide Distributed Blockchains," in IEEE Systems Journal, vol. 17, no. 3, pp. 4775-4785, Sept. 2023.

3.1 Background and Context

By observing the success of the decentralized finance (DeFi) applications alongside notable advancements in blockchain technology, the increased interest in merging decentralized trust with distributed applications is sensible, driving innovation and reshaping traditional business models.

Blockchains and Distributed Ledger Technologies (DLTs) have evolved from humble beginnings into revolutionary tools for decentralized data management and secure transactions. Originating from the concept of a decentralized digital currency, Bitcoin [15], introduced in 2008, blockchain technology provided the foundational framework for DLTs. Bitcoin's blockchain served as a public ledger for recording all transactions in a transparent and immutable manner, demonstrating the potential of

decentralized systems to enable trustless transactions without the need for intermediaries.

Following Bitcoin's emergence, subsequent developments in blockchain and DLTs introduced new features and functionalities. In 2013, Vitalik Buterin proposed Ethereum [63], a decentralized platform that extended the capabilities of blockchain technology by introducing smart contracts. Ethereum's launch in 2015 marked a significant milestone, enabling developers to create decentralized applications (DApps) with programmable logic executed on the Ethereum Virtual Machine (EVM).

The distinctiveness of Blockchain lies in its decentralized structure, which doesn't hinge on a sole central authority for transaction validation. Rather than depending on a single entity, numerous nodes within the network autonomously verify and validate each transaction through a mechanism known as consensus. In brief, consensus protocols ensure that all nodes in a distributed network agree on the validity of transactions before they are added to the blockchain. This decentralized consensus mechanism plays a critical role in blockchain networks which ensures the integrity and immutability of the ledger, making it extremely difficult for any single entity to manipulate or corrupt the data.

Different consensus schemes have been used in blockchains [13], and a few examples of them are provided here. Proof of Work (PoW) involves miners competing to solve complex puzzles, requiring significant computational resources and energy consumption. Proof of Stake (PoS) reduces energy usage by selecting validators based on their cryptocurrency holdings, aiming for faster transaction confirmations. Delegated Proof of Stake (DPoS) further enhances scalability and efficiency by allowing token holders to vote for delegates responsible for transaction validation.

On the other hand, Proof of Authority (PoA) relies on a limited number of trusted validators in private blockchains, ensuring consensus among known entities. Proof of Burn (PoB) introduces a mechanism where participants burn coins to demonstrate commitment to the network, receiving mining or validation rights in return. Additionally, Proof of Space (PoSpace) and Proof of Capacity (PoC) leverage storage space allocation instead of computational power for consensus, providing an alternative approach. Meanwhile, Practical Byzantine Fault Tolerance (PBFT) targets permissioned blockchains, ensuring consensus despite potential faults or malicious nodes. Finally, Directed Acyclic Graphs (DAGs) [64] like IOTA's Tangle [65]

form consensus by referencing multiple previous transactions, offering a different paradigm from traditional blockchains.

All consensus mechanisms have their strengths and weaknesses, making them suitable for different use cases and environments. Consensus protocols play a critical role in blockchain scalability and speed. However, achieving consensus can be a time-consuming process, especially in networks with a large number of nodes or when using complex consensus mechanisms. As blockchain technology continues to evolve, new consensus mechanisms may emerge to address scalability, security, and decentralization challenges.

On the other hand, data dissemination mechanisms are crucial for distributing information among nodes in distributed systems. They can be centralized, relying on a central server, or decentralized, where nodes directly communicate with each other. In blockchains, data dissemination protocols play a vital role in propagating transaction information and maintaining the integrity of the distributed ledger. They facilitate the propagation of transactions and blocks across the network and are tightly coupled with consensus. Inefficient data dissemination can lead to delays in reaching consensus, thereby impacting scalability and speed. The gossip protocol is a decentralized approach where nodes exchange information with random neighbors, quickly disseminating data throughout the network. This protocol is highly scalable and fault-tolerant, making it ideal for large-scale distributed systems including blockchains. Gossip protocols or similar decentralized mechanisms ensure all nodes have consistent copies of the blockchain, preserving its decentralized nature and security.

In the world of real-time Distributed Applications (DApps) integrated with blockchains, particularly when implemented in a decentralized manner, speed is crucial. To this end, we considered Hyperledger Fabric (HF) [16], an open-source and widely used enterprise-level blockchain that is suggested by major cloud service providers. HF is a permissioned blockchain framework designed for enterprise use, known for its scalability and modular architecture. One key feature that contributes to its scalability is its support for various consensus mechanisms, including PBFT, which is more efficient than the ones such as PoW that are used in public blockchains.

Additionally, HF enhances consensus speed by adopting an "execute-order-validate" execution model instead of the traditional "order-execute-validate" model. In this approach, transactions are first executed in parallel across Endorsing Peers (EPs),

and then the resulting endorsements are ordered and validated by the ordering service, streamlining the consensus process and improving overall transaction throughput. This alteration in the execution model significantly reduces latency and enhances the scalability of HF [66]. To further improve its end-to-end latency for use cases such as real-time and distributed applications, we studied all of its phases. HF employs its own gossip protocol for data dissemination in various aspects of its operation. However, it's crucial to note that this gossip mechanism isn't utilized for the endorsement phase due to inherent structural and logical disparities, as well as the unique role played by the endorsement process.

The endorsement process involves designated peer nodes executing a transaction (i.e., simulating the execution without recording results on the blockchain) and then providing a proposal response to the client application. This response encompasses the execution response message, results (both read and write sets), events, and a signature serving as evidence of the peer's execution [67].

The endorsement delay experienced by a client is affected mainly by two components: i) the network delay between the client and the Endorser Peers (EPs) and ii) the processing delay at each EP. The network delay mainly depends on the network congestion and the propagation delays, whereas the processing delay depends on both the CPU capability and the computation load of each EP. Because network and processing delays are time-varying and hence difficult to predict, optimally selecting EPs is hard. Note that a selection algorithm choosing just the best EP based on the minimum experienced delays will concentrate the endorsement requests to the same EPs, increasing the network congestion and the processing load, thus increasing the overall endorsement delays.

In this research, we propose an *optimal EP selection policy* minimizing the endorsement delays. The main idea is to send redundant endorsement requests to multiple EPs. The adopted spatial diversity increases the chance of having the best EPs among the selected ones. The benefit of the proposed approach can be captured by a simple queueing model in which a task is sent in parallel to multiple servers, each with its queueing system, to minimize task completion time.

In our work, the novel contributions are as follows. (i) We highlight the role of the network and processing delays in the overall endorsement delay. (ii) We refer to a simple analytical model, based on the classical theory of queueing systems, to evaluate the effect of redundancy in selecting the EPs and to compute the optimal

number of EPs. (iii) We propose an optimization approach denoted as *OPtimal ENdorsement (OPEN)* based on the analytical results, leveraging the history of endorsement delays. (iv) We demonstrate through extensive simulations that OPEN outperforms the state-of-the-art solution and accurately approximates other optimal policies while having a much lower implementation overhead compared to them.

The rest of this chapter is structured as follows. In Sec. 3.2 we discuss the related work. Sec. 3.3 describes the HF architecture and then focuses on the endorsement phase delay by introducing the network model and the EP selection problem. Sec. 3.4 explains a simple analytical model, derived from classical results on queueing theory, to find the optimal number of EPs in a simplified scenario. In Sec. 3.5, we propose an EP selection algorithm based on the optimal replication factor computed analytically in Sec. 3.4, able to operate in a generic scenario. In Sec. 3.6, we assess by simulation the performance of our proposed approach and compare it with the alternatives proposed and with the state-of-the-art solution. Finally, we draw our conclusions in Sec. 3.8.

3.2 Related works

Different works modeled analytically the endorsement process in HF. [68] modeled the EPs as M/M/1 queues and considered the propagation delays in the network model, coherently with our work. It showed that using a pure "AND" endorsement policy, compared to "OR" or "k-OutOf-Q" policies, significantly increases the endorsement delay by increasing the number of organizations. Similarly, [69] showed the same results by modeling HF using stochastic reward networks. They also observed that for "OR" and "k-OutOf-Q" policies the latency decreases by increasing the number of EPs within the same organization, similar to the effect of increasing R in our work.

[70] modeled HF using Generalized Stochastic Petri Nets and showed that for high request arrival rates, the endorsement phase is a performance bottleneck of HF. This is coherent with the motivation of our work, focusing on optimizing the endorsement phase. [71] showed that using "k-OutOf-Q" policy, increasing k decreases the throughput and increases the latency. This is coherent with our system model since the endorsement latency will be the maximum among k request delays. [72] optimized the HF configurations to improve the throughput and reduce

the delays. Coherently with our results, they showed the equivalence between the "1-OutOf-Q" policy and the "OR" among all organizations. Our results in Sec. 3.3.1 generalize such property.

Some works tried to improve endorsement phase of HF. [73] proposed a way to select the best EP for "1-OutOf-Q" endorsement policy in HF v1.4. They introduced an algorithm running in each EP, called DSLM, to calculate the EP's load by considering multiple resource metrics within an EP. For each request, only half of the EPs are probed to get their actual load, coherently with $R = Q/2$ adopted in OPEN. A version of DSLM tailored to our system model has been considered in Sec. 3.6 as an alternative approach to be compared with OPEN. [74] showed that the failed transactions due to timeouts are affected by the number of statements within the "AND" operator defined in the endorsement policy. Such failures increase the latency and waste of resources due to re-transmissions at the application level.

[75] suggested a way to reduce the possibility of endorsing conflicting transactions. They proposed a cache mechanism inside the EPs to record some data of the recently endorsed transactions and drop the conflicting proposal before execution. Recall that, in the endorsement phase, no execution results will update the world state, so transactions with similar initial world states can propose different updates for the world state. This early drop of the proposal before execution will reduce the computing and network resources by reducing the chance of transaction failure at the validation phase. [76] removed unnecessary operations for pure read requests, by modifying the EPs algorithm to differentiate the process of pure read transactions from mixed read/write ones. This reduced the latency and resource consumption in the endorsement phase.

The main idea of OPEN is to send multiple replicas of the same request to multiple Endorser peers. This approach has been deeply investigated in the literature on queueing theory, motivated by the problem of optimal job assignment to servers. As the literature is huge, we focus just on a few papers for the sake of space. In the generic literature about distributed systems, several works [77–79] investigated the effect of sending replicas of a job to more than one randomly selected server and waiting for the first response to exploit redundancy, as in OPEN. These works introduced redundancy to reduce the job completion time and overcome server-side variability, where a server might be temporarily slow, due to many factors like garbage collection, background load, or even network interrupts. [80] showed that,

besides its simplicity, in many cases, redundancy outperforms other techniques for overall response time. [81], by decoupling the inherent job size from the server-side slowdown, described a more realistic model of redundancy and showed that increasing the level of redundancy can degrade the performance, coherently with our observations in Sec. 3.4. [82] showed that a major improvement results from having each job replicated to only two servers, coherently with our Fig. 3.3 which shows that for the 1-OutOf-k policy, the endorsement latency decreases mostly when varying R from 1 to 2. On the contrary, in our work, we have considered the optimal value of R that minimizes the endorsement latency, which may be greater than 2. [83] showed the reverse relation between the incoming load and the optimal number of replicas, coherently with (3.17), and experimentally obtained the optimal redundancy factor in different job arrival rates and for different service times. Also, [84] theoretically demonstrated that, when replicating the job to multiple servers, the best choice in case of low (or, high) loads is to replicate to all (or, only 1) servers, coherently with (3.17) and with the operations of OPEN, which adapts the replication factor to the instantaneous load.

3.3 Hyperledger Fabric architecture and endorsement

The Execute-Order-Validate approach enables the simulation of the transactions before the agreement of the participants on recording the results in the Hyperledger Fabric blockchain. We describe the role of the entities that are participating in the simulation phase of Fig. 3.1.

The *client* in the endorsement phase, is responsible for preparing the transaction proposal of the users' transactions and sending it to the Endorser Peers (defined below) based on the specified endorsement policy. If the client receives enough endorsements before a specific time-out, it forwards them to the ordering service; otherwise, the client can re-transmit the same proposal in the hope of receiving enough endorsements in time.

The *Peer* is the element responsible for the following tasks. The *Endorser Peer* (EP) simulates/executes the transaction received from the client application, based on the current values of the world state. The *Verifier/Committer* (VCP) receives a block of simulated transactions from the ordering service and verifies their legitimacy to mark them as validated or invalidated. Then it appends the verified

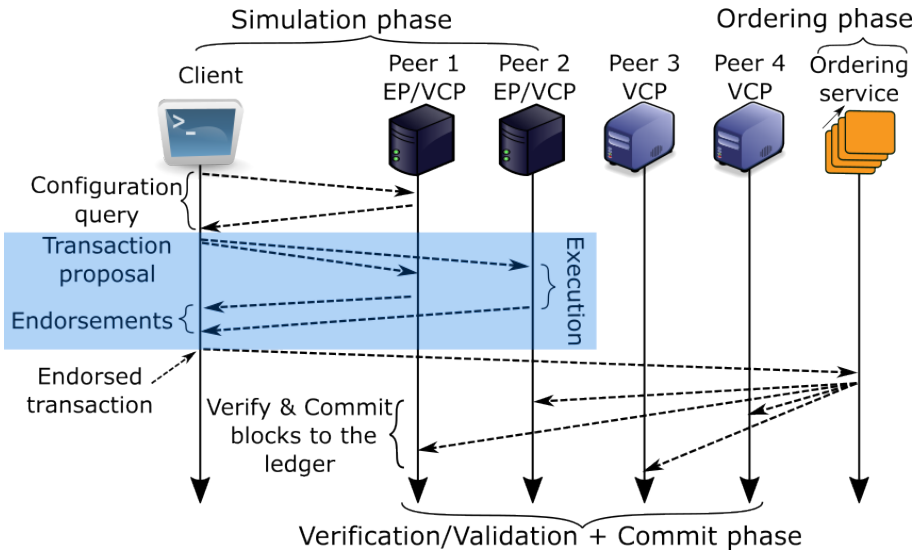


Fig. 3.1 Transaction processing phases in HF highlighting all the message interactions between the involved entities

block to the blockchain, comprising all the transactions (validated or invalidated). To update its copy of the ledger, an EP is typically a VCP at the same time. The peers are owned by various *organizations* that are blockchain members. An organization can be as small as individuals or as large as a multi-national corporation.

The *endorsement policy* defines the logical conditions to validate a transaction in terms of the EPs on a channel that must execute a transaction proposal. In Sec. 3.3.1, we will describe in detail the representation of the endorsement policy. The definition of an endorsement policy is at the organizational level, which means any EP of that organization can represent that organization in the endorsement policy. A transaction should pass three phases to be stored in the blockchain, as shown in Fig. 3.1.

3.3.1 Standard form of an endorsement policy

HF provides a very flexible way to define an endorsement policy. We will show that any endorsement policy, despite its complexity, can be reduced to a standard form. In HF, the definition of an endorsement policy is based on a syntax that allows the operators "AND", "OR" and "k-OutOf" to be applied to a set of organizations and nested expressions [85]. In particular, the operator "k-OutOf-E" returns true whenever at least k expressions within set E are satisfied. Despite the complexity of the policy expression, we prove that the following proposition holds:

Proposition 1. *Any endorsement policy obtained by combining arbitrarily "AND", "OR", and "OutOf" operators is equivalent to the policy:*

$$OR(St_1, St_2, ...) \tag{3.1}$$

where each St_i is either a single organization or the conjunction ("AND") of different organizations.

Proof. In the case of expressions based on only "AND" and "OR" operators, thanks to the distribution principle in logic expressions, we can transform the original expression into the target form (3.1). In the case of "k-OutOf$(e_1, e_2, \ldots, e_m)$ operator, where e_i is a single expression, by definition this holds:

$$k\text{-OutOf}(e_1, e_2, \ldots, e_m) = \text{OR}(\{\text{AND}(E)\}_{E \in \Omega})$$

being Ω the set of all $\binom{m}{k}$ combinations of k expressions from the set of m.

Now, since any expression with the "OutOf" operator is equivalent to one with only "AND" and "OR", by following the previous reasoning, such expression can be reduced to the expression (3.1). □

The policy, defined at the organization level, must be mapped into a policy defined at the EP level since the endorsement requests should be sent to the proper EPs. So, getting the endorsement from a specific organization requires receiving it from *any* of its EPs, which is equivalent to the policy 1-OutOf$(p_1, p_2, \ldots)$, where p_i are the EPs within the organization. Revisiting Proposition 1 applied at the policy expression at the EP level, we can claim:

Proposition 2. *Any endorsement policy defined at the organization level can be expanded into an endorsement policy defined at the EP level as follows:*

$$OR(St_1^{'}, St_2^{'}, ...) \tag{3.2}$$

where each of $St_i^{'}$ is either a single EP or the conjunction ("AND") of different EPs.

The result of Proposition 1 allows investigating only one standard form of endorsement expression, independently from the original expression complexity. Now, by using Proposition 2, we will have the endorsement expression extended

at the EP level. At this level, the final endorsement will be just in the form of the OR between the conjunction ("AND") of different EPs of different organizations, as in (3.2).

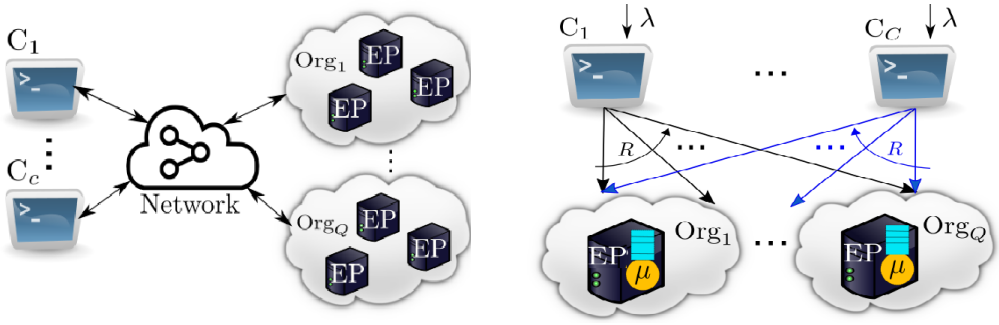
For example, consider a scenario with three organizations and two EPs in each of them. If the endorsement policy is "2-OutOf(o_1, o_2, o_3)", we can rewrite it as:

$$\text{2-OutOf}(o_1, o_2, o_3) \quad = \quad \text{OR}(\{\text{AND}(p_{ij}, p_{i'j'}), \forall i, \forall i' \quad \neq \quad i, \forall j, \forall j'\}) \quad (3.3)$$

where o_i is organization i, and p_{ij} is the EP j of organization i. The expanded version in (3.3) lists all the possible combinations of the EPs that can satisfy the endorsement policy according to the standard form.

3.3.2 Endorser peer (EP) selection algorithm

In our work we focus on the EP selection algorithm, starting from the standard form of the endorsement policy. The *endorsement delay* is the amount of time the client waits, from sending the endorsement request until receiving the first endorsement reply that satisfies the endorsement policy. The response delay from an EP is the sum of two components: the network delay and the processing delay at the EP. The *network delay* depends on the propagation delay and the queueing delay along the path to the EP, which is affected by the time-variant congestion conditions. The overall *processing delay* depends on the *queuing* at the EP before being served and the *computation time* at the EP, which depends on the CPU speed and the instantaneous CPU load and resource contentions. Because the standard form of any endorsement policy comprises an overall "OR" operator, as in (3.2), the endorsement latency corresponds to the *minimum* delay to get a valid statement. Also, each statement is based on an "AND" operator between EPs, so the delay of each statement depends on the *maximum* response delay of all EPs included in a statement. In summary, the endorsement latency depends on the "fastest" group of EPs forming a statement, while the delay of each group depends on the "slowest" EP within the group.



(a) General network model

(b) R requests for 1-OutOf-Q

Fig. 3.2 Network model and the endorsement policy sending each request to R organizations/peers in parallel.

3.3.3 System model for the endorsement phase

Without loss of generality, we consider a fixed network topology connecting C clients with Q organizations, each of them with a generic network connecting the internal EPs, depicted in Fig. 3.2a. We assume that all nodes in the system are always available, the routing is fixed, and the links have enough bandwidth to prevent network congestion caused by the endorsement protocol. Thanks to the service discovery process in HF, we consider only the most updated EPs.

3.4 Background on optimal replication in queueing systems

Now, we discuss an analytical model to compute the optimal number of EPs for each transaction, derived from classical results on task replication in a queueing system, as explained in Sec. 3.2. For the sake of readability, we report the adopted notation in Table 3.1.

We consider a simplified model as shown in Fig. 3.2b, with one EP in each organization. We assume 1-OutOf-Q as the endorsement policy, which corresponds to $\mathrm{OR}(p_1, p_2, \ldots, p_Q)$ in its standard form. For now, we neglect the network delays and concentrate just on processing delays. We suppose each client generates endorsement requests according to a Poisson process with rate λ. Each client selects at random R EPs to send the endorsement request. R will be denoted in the

Table 3.1 Notation

C	number of clients
Q	number of organizations
p	endorser peer (EP)
λ	arrival rate of new transactions to each client
μ	inverse of computation time for the EP server
U	utilization factor in each EP server
R	redundancy factor
W_i	waiting time needed for the ith request to be served
S_i	inter-arrival time between the ordered version of $\{W_i\}_i$
γ	normalized load factor for the worst case $R = Q$
$\hat{R}_k$	optimal R for policy k-OutOf-Q
L_k	endorsement latency for policy k-OutOf-Q
$\mathscr{P}$	set of all available EPs
$\mathscr{P}_e$	set of selected EPs
$\mathscr{P}_e^{\text{old}}$	set of previously selected EPs
T	probe sampling period
TX^n	transaction with local sequence number n
x_p^k	endorsement latency of TX^k for peer p
t_p^{resp}	virtual response delay of a peer for the new TX
t_p^{busy}	virtual time at which EP p is not busy anymore
τ_p^{proc}	processing delay for the current endorsement request
τ_p^{net}	network delay for EP p
τ_p^{queue}	queueing time experienced by the TX at EP p
d_{cp}	the network delay between client c and EP p

following as *redundancy factor*. To model the processing time variability at the EP, we assume an exponentially distributed processing time with an average $1/\mu$, coherently with past works [70, 68]. Thus each EP can be modeled as an M/M/1[1] queue with arrival rate $\lambda RC/Q$ and service rate μ. We define the utilization factor for each EP as:

$$U = \frac{\lambda RC}{\mu Q} \tag{3.4}$$

Thus, for the request traffic to be sustainable, $U < 1$ and the endorsement request arrival rate must satisfy:

$$\lambda < \frac{\mu Q}{RC} \tag{3.5}$$

[1] In classical queueing theory, an M/M/1 queue has a single server, arrivals follow a Poisson process and service times are exponentially distributed [86].

We now claim the following:

Proposition 3. *Under a sustainable arrival rate of endorsement requests and a random selection policy with R EPs, according to the endorsement policy* 1*-OutOf-Q, it holds for the endorsement latency* L_1*:*

$$E[L_1] = \frac{1}{\mu - \frac{\lambda RC}{Q}} \left(\frac{1}{R}\right) \qquad R \in [1, \ldots, Q] \tag{3.6}$$

Proof. From Fig. 3.2b, let λ' be the average incoming rate of the requests for the queue of each EP such that:

$$\lambda^{'} = \frac{\lambda RC}{Q} \tag{3.7}$$

We define W_i as the waiting time of a request to be served at the ith EP, which is the sum of queuing time and the serving time of the request in the ith EP. From M/M/1 well-known properties [86], W_i are i.i.d. and exponentially distributed with mean:

$$E[W_i] = \frac{1}{\mu - \lambda^{'}} \tag{3.8}$$

Observe that:

$$L_1 = \min(W_1, W_2, \ldots, W_R) \tag{3.9}$$

where W_i are i.i.d.. From basic properties of the exponential distribution, L_1 is exponentially distributed with mean:

$$E[L_1] = \frac{E[W_i]}{R} \tag{3.10}$$

and finally get (3.6). □

By computing the first derivative of (3.6) with respect to R, we can prove the following:

Proposition 4. *Let* $\hat{R}_1$ *be the optimal value of R that minimizes* $E[L_1]$ *for the policy* 1*-OutOf-Q.*

$$\hat{R}_1 = \frac{\mu Q}{2\lambda C} \tag{3.11}$$

In summary, the optimal number of EPs changes with λ. For low arrival rates, R must be large to exploit the spatial diversity, without incurring additional overhead

in the processing times. For high arrival rates, conversely, R is small to reduce the load on the EPs. Notably, for the sake of readability, we omitted from (3.11) the clipping to the interval $[1,Q]$ and the rounding procedure to find the optimal integer value of R. We can now extend the result of Proposition 3 to a generic OutOf policy.

Proposition 5. *Under a sustainable arrival rate of endorsement requests and a random selection policy with R EPs, according to the endorsement policy k-OutOf-Q, it holds for the endorsement latency L_k, for any $R \in [k,\ldots,Q]$:*

$$E[L_k] = \frac{1}{\mu - \dfrac{\lambda CR}{Q}} \left(\sum_{i=0}^{k-1} \frac{1}{R-i} \right) \tag{3.12}$$

Proof. Using the same definition of W_i as adopted in the proof of Proposition 3, we can define L_k as the endorsement latency for the policy k-OutOf-Q. Now L_k can be computed as the kth order statistic as follows:

$$L_k = (W_1, W_2, \ldots, W_R)_{(k)} \tag{3.13}$$

recalling the fact that W_i are i.i.d. and exponentially distributed, we can define S_i as the time interval between the ordered version of the W_i (i.e., $S_i = W_{(i+1)} - W_{(i)}$). Thanks to the theory of order statistics [87], S_i is exponentially distributed with average:

$$E[S_i] = \frac{E[W_i]}{R-i} \tag{3.14}$$

By combining (3.10) and (3.14), for any $R \in [k,\ldots,Q]$:

$$E[L_k] = \sum_{i=1}^{k-1} E[S_i] + E(L_1) = \sum_{i=0}^{k-1} \frac{E[W_i]}{R-i} \tag{3.15}$$

and we get (3.12). □

The optimal value of $\hat{R}$ can be computed analytically as well. We impose sustainable request arrivals, i.e., $U < 1$, *for any* R to guarantee sustainable arrivals also in the case $R = Q$, it must hold $\lambda < \mu/C$. Thus, we can set:

$$\lambda = \gamma \frac{\mu}{C} \tag{3.16}$$

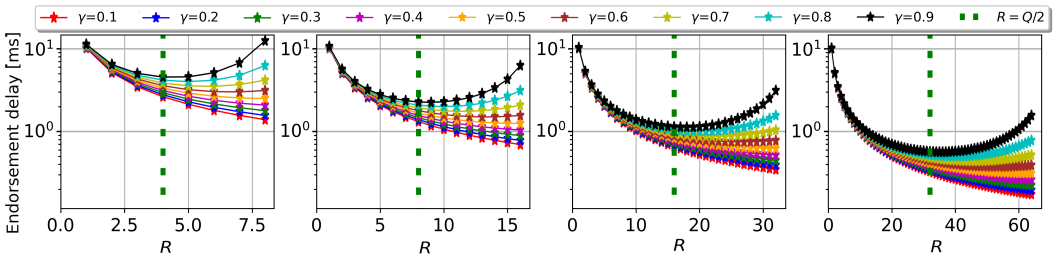


Fig. 3.3 Endorsement latency where $Q \in [8, 16, 32, 64]$ (left to right). The green line represents $R = Q/2$.

with $\gamma \in (0,1)$ being the load factor. By substituting (3.16) into (3.11), we can obtain the optimal number of EPs for 1-Out-Of-Q policy as:

$$\hat{R}_1 = \frac{Q}{2\gamma} \tag{3.17}$$

We can repeat the same derivation also for $\hat{R}_k$, i.e., for a generic k-OutOf-Q policy.

3.4.1 Numerical evaluation

In Fig. 3.3 we reported the endorsement latency computed in the function of γ and R, obtained by substituting (3.16) in (3.12). As expected, we observe a minimum endorsement latency obtained with $R = R_k$, as computed analytically, which depends on the load γ. Due to the difficulty of estimating the load in practical scenarios (which may not be stationary), for $k = 1$, we propose heuristically choosing $R = Q/2$ as a sub-optimal redundancy in our proposed approach, discussed in the following. This choice is robust since it is optimal at high load and at low load the latency increase is limited. Indeed, for $\gamma = 0.5$ the increase is no more than 8% compared to the optimal value, and for $\gamma = 0.1$ no more than 12%. Thus for $k = 1$, $R = Q/2$ appears to be a practical solution, which will be exploited when devising online EP selection algorithms in Sec. 3.5.

The redundancy effect can be limited due to the number of organizations/EPs or applied endorsement policies, as they affect the number of statements generated by Proposition 2. With fewer final statements, there would be less space for redundancy. Indeed, systems with less restrictive and less complex endorsement policies (e.g.,

majority policies) benefit more from redundancy, while organizations benefit from adopting more EPs to increase reliability.

3.5 Practical endorsers selection algorithms

Now, we concentrate on the 1-OutOf-Q policy, since it is coherent with the standard form of any endorsement policy. Without loss of generality, we assume just one client in the system ($C = 1$). We assume that the client is aware of all needed information including available/most-updated EPs, thanks to the configuration query request, as shown in Fig. 3.1, which leverages the available service discovery process.

We propose an optimization procedure to select the EPs, denoted as `OPEN`, whose main goal is to minimize the endorsement response delay. `OPEN` considers the past response delays experienced by the previously selected EPs and selects the EPs with the lowest delays. This choice is motivated by the high temporal correlation between the response delays of an EP, due to queueing in the network and in the EPs. Notably, the history is meaningful only for recently selected EPs, otherwise, it is obsolete. Therefore, it is possible that a highly loaded EP that was not recently requested becomes among the least loaded ones and is worth again sending the request to it. To address this, `OPEN` probes non-selected EPs by sending gratuitous endorsement requests, which are still considered in the evaluation of the endorsement policy. Furthermore, in `OPEN` pending requests are considered indicators of possibly congested EPs, which are chosen at a lower priority.

The pseudocode of `OPEN` is provided in Fig. 3.4. Let TX^n be the transaction with sequence number n, evaluated locally at the client. Let x_p^n be the measured response delays of TX^n for any EP $p \in \mathscr{P}$. Let $\mathscr{P}_e^n$ be the set of selected EPs for TX^n. For each transaction, we initialize all EPs as eligible to be selected (ln. 2). Just for the first transaction, `OPEN` initializes the history of response delays to a dummy value and selects all EPs as selected endorsers (ln. 3-6). For a generic transaction, all response delays are initialized to a dummy value (ln. 7-9). Then the EPs are selected based on a procedure described in the next paragraph (ln. 10). Now `OPEN` sends the endorsement request for TX^n to the computed set of EPs (ln. 11) and updates the measured delays (ln. 12). A new instance of the procedure would start if a new transaction TX^{n+1} is generated. Note that the procedure ends when all the responses are received.

```
1:  procedure OPEN(n)                                          ▷ Process TX^n
2:      e_p^n ← true, ∀p ∈ 𝒫                                   ▷ Init the eligibility vector for TX^(n)
3:      if n = 1 then                                          ▷ Just for the first transaction
4:          for p ∈ 𝒫 do
5:              x_p^0 ← x_p^1 ← −1                             ▷ Init the response delay history
6:          𝒫_e^1 ← 𝒫                                          ▷ Select all the available peers
7:      else                                                   ▷ Consider a generic transaction
8:          for p ∈ 𝒫 do
9:              x_p^n ← −1                                     ▷ Init the measured delays for TX^(n)
10:         𝒫_e^n ← Select-Endorsers()
11:     Send-Endorsement-Requests(TX^n, 𝒫_e^n)
12:     X^n ← Update-Response-Delays()
```

Fig. 3.4 Pseudocode of the `OPEN` algorithm for TX^n

We now discuss how `Select-Endorsers` function operates. Inspired by our previous result in (3.17), it selects $|\mathscr{P}|/2$ EPs chosen among the ones that experienced the lowest response delays, based on the measures for the last transaction TX^{n-1}. The choice is challenging when one or more responses are still pending for TX^{n-1}, and the algorithm key idea is that the corresponding EPs are considered as congested and thus should not be selected for the current transaction TX^n.

The pseudocode is reported in Fig. 3.5. It calculates the maximum delay measured for TX^{n-1} (ln. 2). For each EP in $\mathscr{P}_e^{n-1}$ such that the response is not received yet, we mark the corresponding EP as non-eligible (ln. 3-5). There are two cases. The first case is the special one in which no responses have been received for TX^{n-1}, thus the algorithm speculates the delay equal to the delay of TX^{n-2} (ln. 6-7). The eligibility assigned to the EPs will lead to selecting the other $|\mathscr{P}|/2$ EPs compared to the previous ones. The second case is the typical one in which at least some responses have been received for TX^{n-1} (ln. 8). For the EPs used in TX^{n-1} and for which no response has been already received, the speculated delay is equal to the maximum delay $d_{\max}$ plus some constant ε, chosen enough small to be negligible compared to the average network and processing delays (e.g., 1 ns) (ln. 9). This will model the fact that the actual delay is unknown, but for sure it is strictly larger than $d_{\max}$. Finally, for all the other EPs, not used for TX^{n-1}, the delays are speculated to be equal to X^{n-2} (ln. 10-11). Now, the EPs are sorted based on the X^{n-2} delay values and the half best will be selected (ln. 12). A random EP from not selected ones will be chosen as the gratuitous probe EP (ln. 13). The slowest

1: **procedure** SELECTENDORSERS( )
2: $d_{\max} = \max_{p \in \mathscr{P}_e^{n-1}} \{x_p^{n-1}\}$ ▷ Max measured delay for TX^{n-1}
3: **for** $p \in \mathscr{P}_e^{n-1}$ **do** ▷ For EPs used for TX^{n-1}
4: **if** $x_p^{n-1} = -1$ **then** ▷ Not yet response from EP p
5: $e_p^n \leftarrow$ **false** ▷ Make the EP Not-eligible for TX^n
6: **if** $d_{\max} = -1$ **then** ▷ No delay measured for TX^{n-1}
7: $x_p^{n-1} \leftarrow x_p^{n-2}$ ▷ Use past delays
8: **else**
9: $x_p^{n-1} \leftarrow d_{\max} + \varepsilon$ ▷ Speculate the delay
10: **for** $p \in \mathscr{P} \setminus \mathscr{P}_e^{n-1}$ **do** ▷ For EPs not used for TX^{n-1}
11: $x_p^{n-1} \leftarrow x_p^{n-2}$ ▷ Use past delays
12: $\mathscr{P}_e^n \leftarrow$ `Eligibile-EPs-with-min-delay`$(|\mathscr{P}|/2, X^{n-1})$
13: $p \leftarrow$ `Random-EP`$(\mathscr{P} \setminus (\mathscr{P}_e^n \cup \mathscr{P}_e^{n-1}))$ ▷ Select probe EP
14: $\mathscr{P}_e^n \leftarrow$ `Replace-slowest-EP`$(\mathscr{P}_e^n, p)$ ▷ Embed the probe EP
15: **return** $\mathscr{P}_e^n$ ▷ Selected EPs augmented with the probe EP

Fig. 3.5 Pseudocode for `SelectEndorsers`

EP from $\mathscr{P}_e^n$ will be replaced with the gratuitous probe EP (ln. 14), and $\mathscr{P}_e^n$ will be returned to the main OPEN process (ln. 15).

3.6 Performance evaluation

We developed an event-driven simulator using OMNeT++ [88]. We considered a scenario with $C = 8$ clients and $Q = 8$ organizations, each of them with 1 EP, thus $|\mathscr{P}| = Q$. The endorsement requests are generated according to a Poisson process at

Table 3.2 Experimental Implementation of HF use cases

Application	Paper	organizations	EPs per organization	clients
Healthcare applications	[89]	5	1	any
Construction management	[90]	6	4	any
Privacy-Preserving Healthcare	[91]	3	3	any
Drug traceability	[92]	6	scalable	any

each client and we set the normalized load $\gamma \in [0.1, 0.9]$. Then fixing $\gamma = 0.5$, we considered more scenarios by varying $Q \in \{8, 16, 32, 64\}$, each organization with the number of EP $\in \{1, 2, 4, 8\}$, and $C \in \{8, 40, 125, 1000, 8000, 32000\}$ clients; in each scenario we fixed all parameters except one. These choices are made based

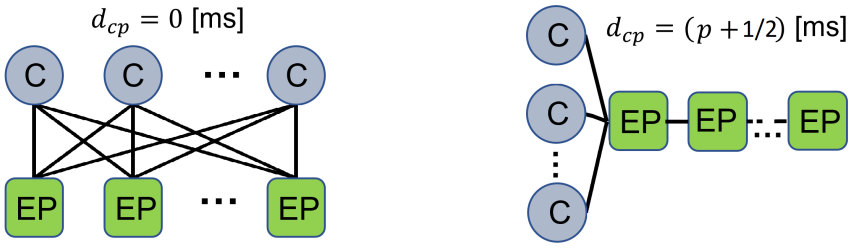
Table 3.3 Settings for different distributions of the computation time with different coefficient of variation (Cv).

Cv (Bimodal)	0.0	0.5	1	2	5
$P(\mu = \mu_1)$	0.5	0.6	0.75	0.9	0.98
$1/\mu_1$ [ms]	10.0	5.9	4.2	3.4	2.85
$1/\mu_2$ [ms]	10.0	16.1	27.3	70.0	360.0

on the values for some practical use-cases that are mentioned in Table 3.2, It is also worth noting that, according to the referenced works, the number of clients can be any number. To understand the performance under non-stationary requests, we also considered a Poisson-modulated process with squared-wave cyclo-stationary load, with a period equal to 1200 ms, duty cycle 50%, and normalized load $\gamma = 0.5$.

To consider the effect of different kinds of computation, we assume the computation time of each EP to be either exponentially distributed or bi-modal distributed with an average equal to 10 ms, whose value has been achieved from our practical measurements in HF EPs. In the bi-modal case, we assumed that, with a given probability, the computation time is constant with the value $1/\mu_1$, otherwise its value is $1/\mu_2$. Table 3.3 shows the coefficient of variations (Cv) for the adopted setting. To model the heterogeneity in the computing power and resources of the EPs, we considered a *non-homogenous scenario* in which we assigned different average computation times to different EPs (i.e., $(2,4,6,8,12,14,16,18)$ ms) where the computation time of each EP is exponentially distributed. We considered three scenarios for the network model, two of them are synthetic and the last one is real. Let d_{cp} be the network delay between client c and EP p. In the first scenario, denoted as S1, the network delays are negligible compared to the processing times at the EP, i.e., $d_{cp} = 0$ (Fig. 3.6a). In the second scenario, denoted as S2, we set linearly increasing delays between any client and the EPs, similarly to a linear topology where all clients are closer to the first EP, i.e., $d_{cp} = (p + 1/2)$ ms for $p \in [1, Q]$. This implies similar delays from each EP to any client while on average the total network delays are comparable to the processing times at the EPs (Fig. 3.6b).

In the third scenario, denoted as S3, we selected the *Highwinds* network from [93], shown in Fig. 3.7, as a real world-wide scenario where the link delays are calculated based on the physical distance between the geographical position of the nodes (using the Haversine formula) and the propagation speed is $2/3$ the speed of light. The



(a) S1 (bipartite) (b) S2 (linear)

Fig. 3.6 The two synthetic network topologies adopted for test scenarios in our simulations.

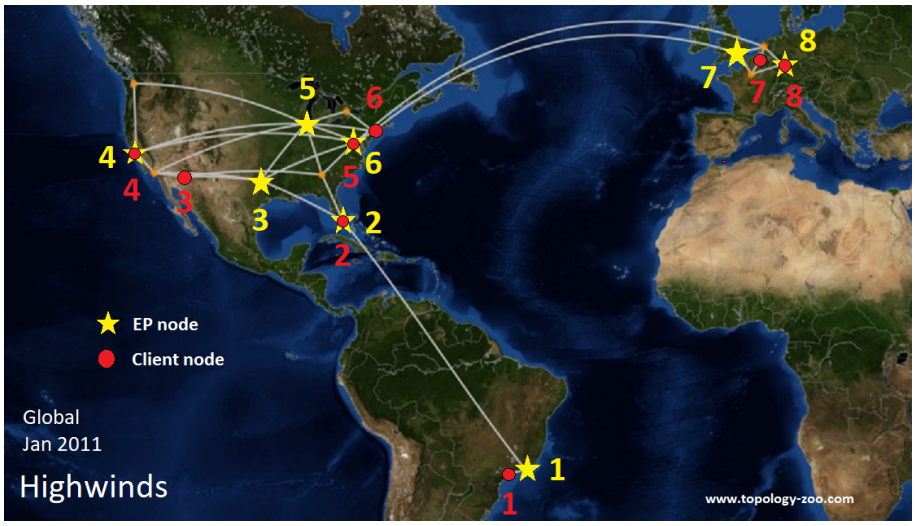


Fig. 3.7 Real network topology (S3) showing the EPs and clients placement with interconnecting network topology

clients here can be divided into two groups: (i) *far clients* placed in nodes $1,7,8$, and (ii) *centered clients* placed in nodes $2,3,4,5,6$.

We measure the average endorsement latency as the main performance metric. The endorsement latency is calculated from the moment the endorsement request is sent out from the client until the first response is received by the client. For comparison, we considered three EP selection algorithms, namely RND, OOD, and DSLM, where the first two are proposed by us.

Random EPs (RND)

RND is the policy adopted in the analytical model of Sec. 3.4. Every endorsement request is sent to R randomly chosen EPs. If $R = Q/2$, the policy is denoted as

1: **procedure** DSLM(n) ▷ Process TX^n
2: **if** $n = 1$ **then** ▷ Just for the first TX
3: $l_p \leftarrow \bar{x}_p \leftarrow x_p^0 \leftarrow 0, \forall p \in \mathscr{P}$ ▷ Init EP load and delay values
4: $\mathscr{P}_h \leftarrow$ `Select-`$|\mathscr{P}|$`/2-random-peers` ▷ Random half EPs
5: **for** $p \in \mathscr{P}_h$ **do**
6: $\bar{x}_p \leftarrow [\alpha \bar{x}_p + (1-\alpha) x_p^{n-1}]$ ▷ Average delay
7: **return** $\arg\min_{p \in \mathscr{P}_h}\{(\bar{x}_p^{0.5}+1)q_p\}$ ▷ Choose min product delay queue length.

Fig. 3.8 Pseudocode for DSLM adapted to our model

RND-half. If R adapts to the load according to the rule $R = Q/(2\gamma)$, as in (3.17), it is denoted as *RND-load*.

Dynamic Stochastic Load Minimization (DSLM)

Dynamic Stochastic Load Minimization (DSLM) was proposed in [73] and the pseudocode of the version adapted to our system model is shown in Fig. 3.8. Just for the first TX, DSLM initializes the load l_p and the measured response delay x_p^0 of any EP p (ln. 2-3). Typically, it randomly selects half of the EPs (ln. 4) and evaluates heuristically the load on each selected EP by the product of the square root of the response delay and the corresponding queue length (ln. 5-6). The average is obtained with an exponential moving average with parameter α. Finally, DSML returns the EP with the lowest estimated load among the selected ones (ln. 7).

Oracle Optimal Delays (OOD)

As a reference for all the endorsement algorithms, we define an online Oracle-based Optimal Delays (OOD) EP selection policy that minimizes the endorsement latency given a fixed replication factor R, denoted as OOD-R. The pseudocode of OOD-R is provided in Fig. 3.9. We assume an oracle that knows in advance the response delay of any endorsement request if sent to a specific EP. Thus, the oracle knows for any EP p: (i) the absolute time t_p^{busy} at which the EP will finish (or has finished) to serve the last received endorsement request TX^{n-1}, (ii) the processing time τ_p^{proc} of the endorsement request TX^n, and (iii) the overall network delay τ_p^{net} between each client and the EP. Thus, if sent to EP p, the response to TX^n will be received from

1: **procedure** OOD-$R(n)$ ▷ Process TX^n
2: **for** $p \in \mathscr{P}$ **do** ▷ For each EP
3: $t_p^{\mathrm{resp}} = \max\{t^{\mathrm{now}} + \tau_p^{\mathrm{net}}, t_p^{\mathrm{busy}}\} + \tau_p^{\mathrm{proc}}$ ▷ Compute response delay
4: $p = \arg\min_{p \in \mathscr{P}}\{t_p^{\mathrm{resp}}\}$ ▷ Choose the min response delay EP
5: $\mathscr{P}_e = \texttt{Find-}(R-1)\texttt{-EPs-with-largest-}\{t_p^{\mathrm{resp}}\}$
6: **return** $\{p\} \cup \mathscr{P}_e$ ▷ Return endorsement set

Fig. 3.9 Pseudocode of OOD-R

EP p at a predicted time t_p^{resp} (ln. 3) equal to:

$$t_p^{\mathrm{resp}} = \max\{t^{\mathrm{now}} + \tau_p^{\mathrm{net}}, t_p^{\mathrm{busy}}\} + \tau_p^{\mathrm{proc}} + \tau_p^{\mathrm{net}} \tag{3.18}$$

since if at the time of arriving the request to an EP its queue is empty, then the request will be served at $t^{\mathrm{now}} + \tau_p^{\mathrm{net}}$, otherwise at t_p^{busy}. Then, the request will be processed for τ_p^{proc} and the response will be sent back, experiencing τ_p^{net} delay. Now OOD-R chooses the EP with the smallest predicted time to minimize the response delay (ln. 4). The remaining $(R-1)$ endorsement requests (if any) will be sent to the EPs in decreasing order of predicted time (ln. 5). This allows to loading of the "slowest" EPs with requests whose responses will be received late and thus reduces the load on the "fastest" EPs, for the sake of future endorsement requests.

It should be noted that, in the case of RND-load, the request arrival is assumed to be stationary, thus, the system load can be estimated with high accuracy. Also, OOD is implementable with enough control information, but obtaining this information would need instantaneous communication with the EPs, which is challenging to accomplish in a practical situation. So, both algorithms are not practical in a real scenario.

3.6.1 Simulations results

For a fair comparison between OOD and other approaches, in all test scenarios we selected OOD-half, i.e., with the same R as OPEN, and RND-half, and slightly smaller R than RND-load, for which $R \in [Q/2, Q]$. Only DSLM has a completely different redundancy factor ($R = 1$).

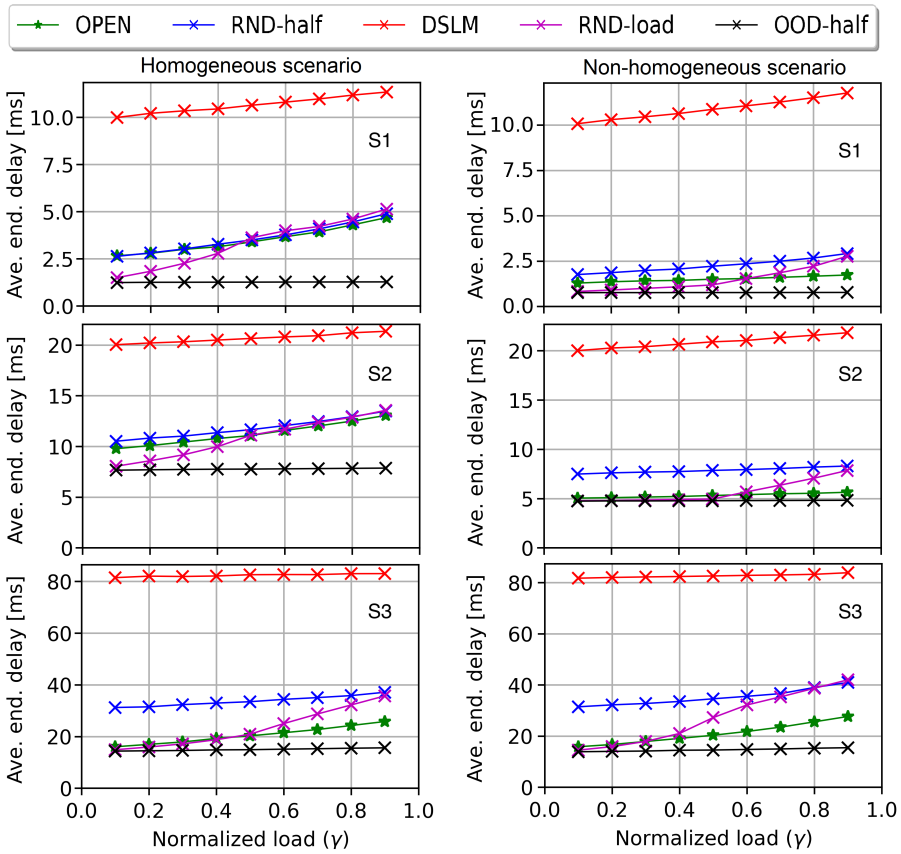


Fig. 3.10 Average endorsement delay for i) average computation time of 10 ms for each EP: S1 (left-up), S2 (left-middle), S3 (left-down), ii) different average computation times from [2 to 18] ms for each EP: S1 (right-up), S2 (right-middle), S3 (right-down).

Homogenous scenario

The left graphs in Fig. 3.10 show the simulation results for a homogenous scenario with all EPs with computation times that are exponentially distributed with the same average.

In scenario S1 (left-up), all delays are purely due to processing in the EPs. Since DSLM does not exploit redundancy and for $\gamma = 0.1$ the queuing at the EP is negligible, its response delay is around 10 ms, equal to the computation time. By increasing the load and hence the queueing, the average delay increases slightly. Instead, by exploiting the redundancy all the other approaches can get smaller delays, by a factor of 2 to 6. As expected, OOD-half achieves the best average endorsement latency among all solutions. At low loads, due to the maximum redundancy factor

(i.e., $R = 8$), RND-load performs closer to OOD-half by always having the fastest EP among its selection. By increasing the arrival rate, for both RND-half and RND-load, the endorsement latency increases as the selection of EPs is not as efficient as OOD-half, which knows in advance the best EP. At high loads, for RND-load, R is almost 4, hence RND-load shows similar results to RND-half. OPEN has a redundancy factor $R = 4$, as RND-half, but selects EPs with smaller estimated delays. At low loads, OPEN has a small advantage over RND-half, as the queueing is almost negligible. As the offered load increases, the higher queueing makes OPEN more efficient, also thanks to the higher frequency by which the response delays are estimated.

In scenario S2 (left-middle), as expected, OOD-half is the best algorithm, and DSLM is outperformed by all other solutions by a factor of 2 to 3. Due to the linearly increasing network delays, the effect of redundancy in EPs becomes less dominant, so the delay's improvement in scenario S2 is less than in S1. On the other hand, at low loads, OPEN acts slightly better than RND-half compared to S1, thanks to being aware of the network delays. At high loads, OPEN behaves close to RND-half since the queueing delays become dominant to network delays.

In scenario S3 (left-down), again OOD-half is the best approach, DSLM is outperformed by all other solutions by at least a factor of 2. Due to the different network delays, on average much larger than the computation times, the redundancy is less effective, thus a lower delay improvement is experienced in S3 compared to S2, and S1. OPEN performs quite similarly to RND-load in low loads even with a half number of selected EPs, and much better in high loads. OPEN completely outperforms RND-half in all loads since it exploits mainly the EP with lower network delays.

Non-homogenous scenario

The simulation results for a non-homogenous scenario are reported in the right graphs of Fig. 3.10. As a reminder, now the average computation times for the EPs are different, but the overall average is the same as in the homogenous scenario. In all three scenarios S1, S2, and S3, as DSLM is not able to exploit redundancy, it is not able to reduce its average latency. On the other hand, by exploiting redundancy, all other approaches can reduce their average latency, where OOD-half achieves the lowest latency thanks to its global knowledge of the system.

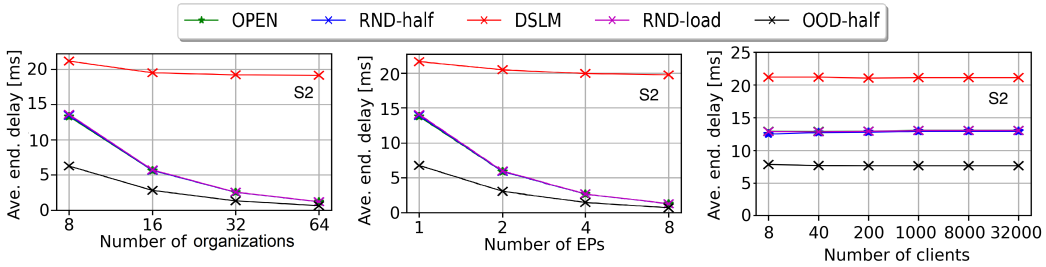


Fig. 3.11 Average endorsement delay for clients with average computation time of 10 ms for each EP in scenario S2, and $\gamma = 0.5$: scaling the number of organizations (left), scaling the number of EPs (middle), and scaling the number of clients (right).

In scenario S1, RND-load reduces the latency more than RND-half, since the higher redundancy factor increases the chance of selecting EPs with lower average computation times. With the same redundancy factor as RND-half, OPEN reduces the most the delays for all loads, as it employs the latency history to select EPs with lower average computation times. In scenario S2, a similar behavior as in S1 is observed for all the algorithms. OPEN, by exploiting the delay history comprising both the average computation times and the network delays, achieves the best performance by almost approaching OOD-half. In scenario S3, we observe almost similar results as in scenario S3 of the homogeneous case, as the variation in the average computation times is still negligible to the average network delays.

Scaling the number of organizations, EPs, and clients.

The simulation results for larger scenarios are shown in Fig. 3.11. We consider the S2 scenario, to get a heterogeneous system in terms of network delays, and we fixed $\gamma = 0.5$. By increasing the number of organizations, the overall number of EPs increases, thus the endorsement latency is reduced for all the algorithms exploiting redundancy, as shown in Fig. 3.11 (left). The same behavior is observed when the number of EPs in each organization increases (see Fig. 3.11 middle). The similarity with the previous graph is that we are considering the 1-OutOf-N policy here, which by recalling (2), for this endorsement policy there is no difference between two EPs of the same organization or different EPs of different organizations.

According to Fig. 3.11 (right), changing the number of clients does not affect the approaches. Note that increasing the number of clients will reduce the efficiency of

the information gained by OPEN and it will converge to the RND-half results for homogeneous cases with less dominant network delays.

Bi-modal computation times

The simulation results are shown in Fig. 3.12. In scenario S1, for constant computation time (Cv=0), the redundancy is not beneficial for delay reduction, while at high loads it can increase the EPs' queue length and thus the delay. For larger Cv, all the algorithms, except DSLM, decrease the average endorsement delay. This is because the average of the minimum between a sequence of i.i.d. random variables is smaller when the variance is larger. All the solutions, except for DSLM, behave similarly for low and high loads.

In scenario S2, also for Cv=0, the redundancy reduces the average delay. The reason is that DSLM considers the computation load at the EPs obliviously of the network delays, which are dominating the computation times. But, in the other approaches, redundancy increases the chance of selecting the EP with lower network delays. By increasing Cv, redundancy can reduce the latency even more, by benefiting from the variability in the computation times. At low load ($\gamma = 0.2$), OPEN performs quite well as it also selects EPs with lower network delays. RND-load is performing slightly better as it sends to all EPs. OOD-half is even better than RND-load with a small margin, thanks to the lower load guaranteed by setting $R = 4$. At high load ($\gamma = 0.8$), RND-load adopts $R = 5.7$ (on average) and the corresponding queueing penalizes the overall response delay. OPEN acts slightly better thanks to the smaller value of R.

In the S3 scenario, all approaches are not affected by Cv, as the variability in the computation times is compensated by the network delays which vary between 0 ms and 7 times the average computation time. At low load ($\gamma = 0.2$), OPEN selects closer EPs in terms of network delays and outperforms RND-half by a factor greater than 2, while being very close to OOD-half. RND-load achieves the same results as OPEN by selecting all the EPs (i.e., $R = 8$), which include the closest EP as well. At high load ($\gamma = 0.8$), as in scenario S2, RND-load is penalized by the queueing. OPEN reduces the endorsement delay up to 70% compared to DSLM. Notably, differently from OPEN, RND-load may not select the closest EPs. As expected, for both loads OOD-half performs the best, since it always selects the minimum combination of the network delay and the processing delay.

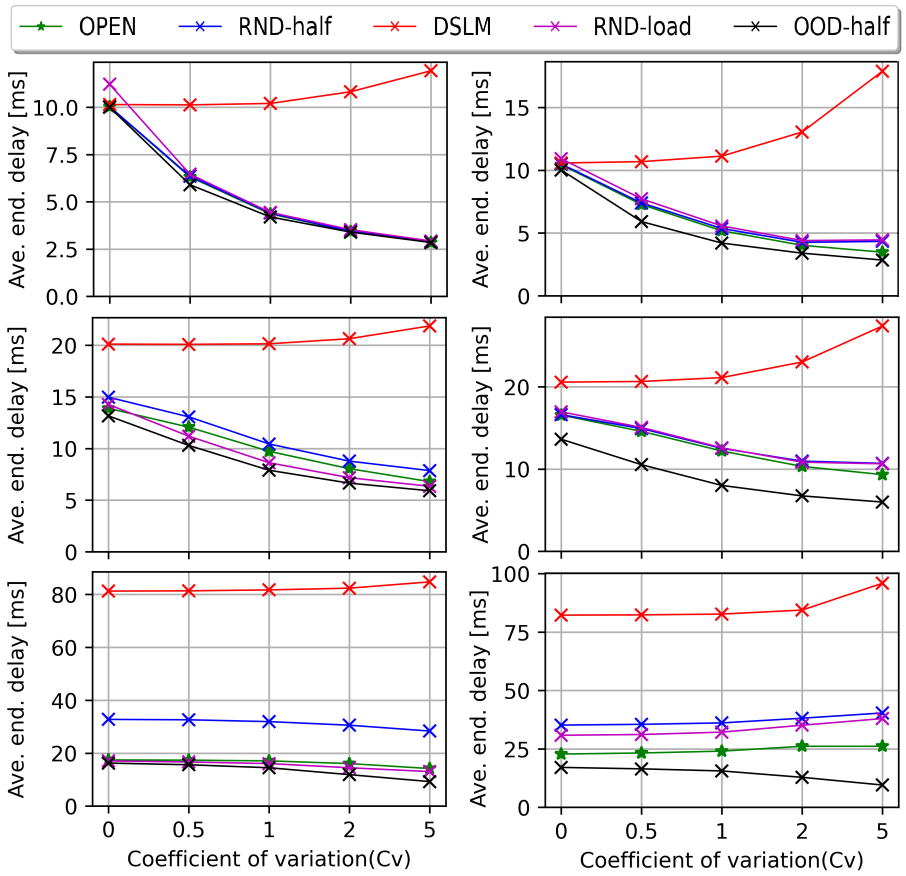


Fig. 3.12 Average endorsement latency under bimodal computation times: i) normalized load: $\gamma = 0.2$ (left), $\gamma = 0.8$ (right), ii) scenarios: S1 (up), S2 (middle), S3 (down)

For all scenarios, when Cv=0, all EPs have one computation time which is exponentially distributed with an average of 10 ms. So, the experienced latency in low loads and high loads is similar to what is reported in Fig. 3.12 (left) when $\gamma = 0.2$ and $\gamma = 0.8$ respectively. For all scenarios, the redundancy reduces the average delay.

In scenario S1, by increasing the CV, all the algorithms, except DSLM, decrease the average endorsement delay. This is because the average of the minimum between a sequence of i.i.d. random variables is smaller when the variance is larger. At low load ($\gamma = 0.2$), RND-load is performing a bit better than OPEN and RND-half, as it sends to all EPs. OOD-half is slightly better than RND-load thanks to the lower load guaranteed by setting $R = Q/2$. At high load ($\gamma = 0.8$), RND-load adopts $R > Q/2$ and the corresponding queueing penalizes the overall response delay. By increasing the CV, for DSLM in both low loads and high loads, the average endorsement delay

increases since even a small possibility of selecting an EP with very high computation time can increase the average endorsement delay.

In scenario S2, similar to scenario S1, in both low loads and high loads by increasing the CV, the average endorsement delay for DSLM increases. In the other approaches, redundancy increases the chance of selecting the EP with lower network delays. By increasing Cv, redundancy can benefit from more variability in the computation times, and reduce the latency even more. At low load ($\gamma = 0.2$), OPEN performs slightly better than RND-half as it also selects EPs with overall lower network delays and processing delays. RND-load is performing even better as it sends to all EPs. OOD-half is even better with a small margin, thanks to the lower load guaranteed by setting $R = Q/2$. At high load ($\gamma = 0.8$), RND-load adopts $R > Q/2$ and the corresponding queueing penalizes the overall response delay. OPEN acts slightly better thanks to the smaller value of R.

In the S3 scenario, all algorithms benefiting from the redundancy, are not affected by Cv, as the variability in the computation times is compensated (i.e., equalized) by the network delays which vary between 0 ms and 10 times the average computation time. At low load ($\gamma = 0.2$), OPEN being very close to OOD-half, outperforms RND-half by a factor around 2 due to selecting closer EPs in terms of network delays. RND-load achieves the same results as OPEN by containing the closest EP as it selects all the EPs (i.e., $R = Q$). At high load ($\gamma = 0.8$), same as scenario S2, OPEN reduces the endorsement latency up to 70% compared to DSLM. RND-load is penalized by both the queueing and lower number of selected EPs (i.e., $R < Q$), as differently from OPEN, RND-load may not select the closest EPs. Again, for both loads OOD-half performs the best since it always selects the minimum combination of the network delay, the queueing delay, and the processing delay.

Log-normal computation times

The simulation results are shown in Fig. 3.13. All the results are similar to the results of high loads ($\gamma = 0.8$) in Bi-modal computation times, except for scenario S1, in which for constant computation time (Cv=0), redundancy is not beneficial for delay reduction. For larger Cv, all the algorithms, except DSLM, decrease the average endorsement delay as the average of the minimum between a sequence of i.i.d. random variables is smaller when the variance is larger.

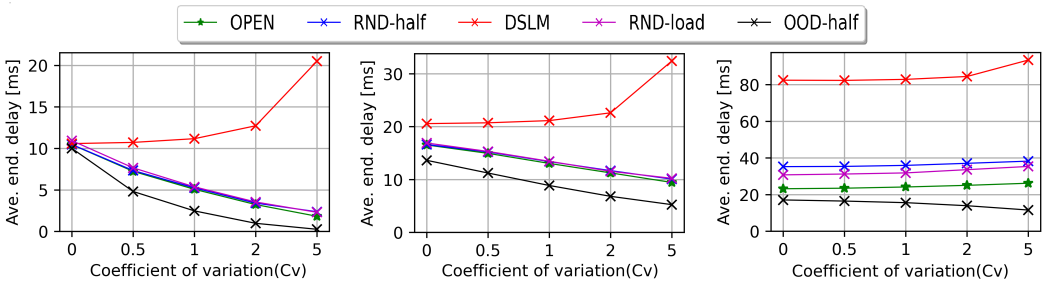


Fig. 3.13 Average endorsement latency under log-normal computation times for normalized load $\gamma = 0.8$ and different scenarios: S1 (left), S2 (center), S3 (right).

Table 3.4 Average endorsement delays for cyclo-stationary input rates with different scenarios.

Scenario	S1 [ms]	S2 [ms]	S3 [ms] centered clients	S3 [ms] far clients
OOD-half	1.2	7.1	13.6	16.9
OPEN	2.9	11.7	17.9	24.2
RND-load	3.1	11.2	19.2	23.7
RND-half	3.2	12.2	25.9	43.6
DSLM	10.8	22.8	65.9	110.3

Log-normal computation times

All the results are similar to the results of high loads ($\gamma = 0.8$) in Bi-modal computation times, except for scenario S1, in which for constant computation time (Cv=0), redundancy is not beneficial for delay reduction. For larger Cv, all the algorithms, except DSLM, decrease the average endorsement delay as the average of the minimum between a sequence of i.i.d. random variables is smaller when the variance is larger.

Cyclo-stationary request process

We compared OPEN with other approaches under Poisson-modulated cycle-stationary load. We evaluated the average endorsement delays by using an exponential moving average. The results are provided in Table 3.4. In S1 and S2, OPEN, RND-half, and RND-load showed almost constant average endorsement latency, while DSLM and ODD results are the highest and the lowest respectively. Interestingly, all the results for different approaches in S1 and S2 are very close to the results gained from

Fig. 3.10 (left) for $\gamma = 0.5$, even if the load was changing periodically. This means that all of them are robust to load change in homogeneous scenarios.

In S3, as a non-homogenous real scenario, OPEN shows a small difference of the average endorsement latency between centered and far clients (recall their definition in Sec. 3.6). This difference (6 ms) is negligible compared to the average network delays in S3 (50 ms). The same behavior is observed for RND-load. On the other hand, in RND-half the performance depends heavily on the client's position; even in the case of centered clients, RND-half experiences more endorsement latency than OPEN with far clients. As expected, OOD-half achieves the best endorsement latency with minimum difference regardless of the client's position. DSLM performs the worst with latencies about 4 times larger than OPEN.

These results show that OPEN adapts to load changes even in the presence of unbalanced network delays. Also, OPEN outperforms RND-half and DSLM, while it shows similar results to RND-load and to OOD-half.

3.7 Proof-of-concept implementation

We have validated OPEN in a practical scenario, by implementing a proof-of-concept solution in a real HF platform and testing it. Our preliminary results show that OPEN is easily implementable in HF, just with some coding on the client with no modification to HF core-engine code. We used Ubuntu 20.04.6 LTS on a single server with 32GB RAM and a 12X Intel Core-i7-12700H-(2.30-4.70)GHz CPU. Using HF v2.2, 8 EPs were implemented as Docker containers and connected directly through a Docker virtual network, following the setup of Fig. 3.14. A background load of endorsement requests is created across all the EPs such that each EP is loaded differently from the others, according to Table 3.5. Fig. 3.15 shows the corresponding processing delays at the different EPs, which are unbalanced as expected. In our settings, the network delays are practically negligible.

The measured endorsement delay is reported in Table 3.6 shows that OPEN can reduce the endorsement delay by 22% compared to RND-half. We can deduce that, in this non-homogeneous case, OPEN can "learn" the best EPs where to send the request.

Table 3.5 Background load for the EPs in the testbed

Endorser Peer id	Request rate [1/s]
1	1.2
2	2.3
3	3.5
4	4.7
5	5.9
6	7.1
7	8.3
8	9.5

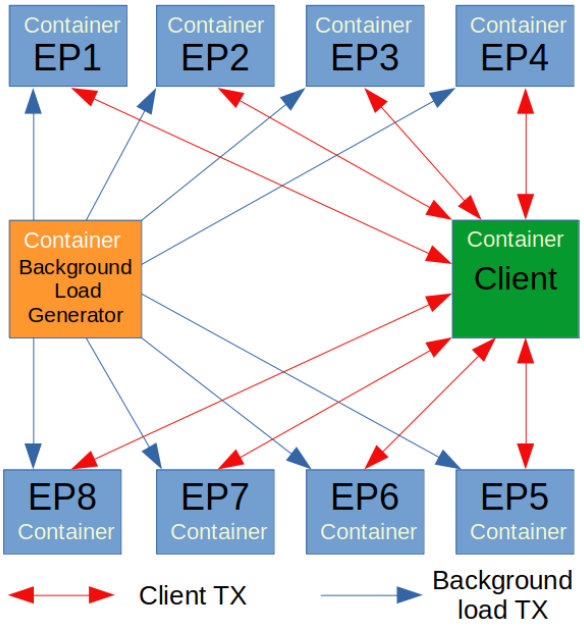


Fig. 3.14 Experimental testbed architecture

Table 3.6 Experimental results in the considered testbed

Algorithm	Ave. endorsement delay	95% confidence interval
RND-half	115 ms	$[110, 120]$ ms
OPEN	94 ms	$[87, 101]$ ms

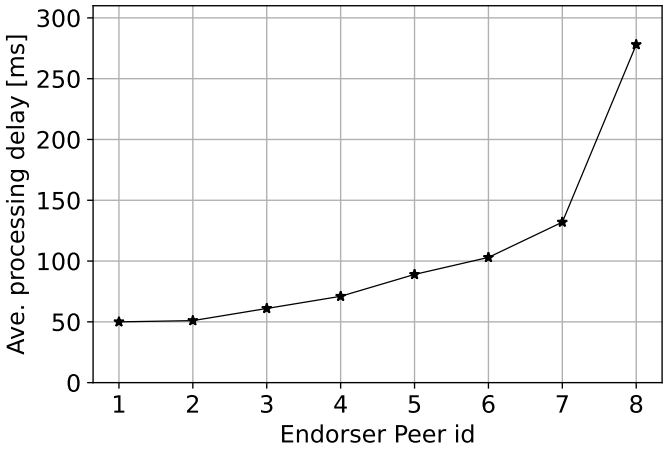


Fig. 3.15 Measured processing delay at each EP.

3.8 Discussion

We addressed the problem of minimizing the endorsement latency in HF. Leveraging some results obtained in a simplified queueing model, we proposed the OPEN algorithm to choose multiple EPs for each transaction by taking into account the measurements from the past requests, in a realistic scenario. Through simulations with OMNeT++, we showed that independently from the scenario, OPEN is robust and achieves performance remarkably close to the optimal oracle-based approach (OOD) and outperforms state-of-the-art solutions.

OPEN has been validated only by extensive simulations. Beyond the scope of this work, we implemented OPEN in HF to validate the proposed approach in a realistic setting. The experimental results of the first version of the proof-of-concept are very promising. We leave the optimization of the design of the client-based OPEN solution and its extensive experimental validation for future work.

Chapter 4

Optimizing Game Engine Module Placement in Cloud Gaming

A part of the work presented in this chapter has been submitted to IEEE Access to be published as:

- I. Lotfimahyari, L. De Giovanni, D. Gadia, P. Giaccone, D. Maggiorini, C. E. Palazzi, "MAP-MIND: An Offline Algorithm for Optimizing Game Engine Module Placement in Cloud Gaming".

4.1 Background and Context

In recent years, a significant trend has emerged in the gaming industry as video games increasingly transition into online services. In these online games, the gaming experience heavily relies on a central server that serves as a shared exchange point for multiple players. This server is responsible for managing shared resources efficiently, irrespective of the game mechanics, providing either single-player or multiplayer experiences. The evolution of online games has progressed through various stages. Initially, centralized physical servers managed all gaming sessions, leading to scalability issues. With the advent of cloud computing, gaming servers transitioned to distributed systems, running as virtualized software services in the cloud. Although this approach reduced costs and increased scalability, it often required proprietary user-side hardware or software. In the latest generation, complete cloud gaming, also

known as game streaming, has emerged. Here, computation is entirely offloaded to remote cloud servers, with a video stream sent to the player's device. This paradigm shift eliminates the need for high-end hardware, making high-quality gaming accessible. Services like Google Stadia, Amazon Luna, GeForce Cloud Gaming, and Microsoft Xbox Cloud exemplify this model. While cloud gaming offers significant benefits, mass migration to this model could strain even hyper-scaling cloud architectures, as they are not inherently designed to meet strict Quality of Service (QoS) requirements while organizing internal modules.

In today's cloud architectures, gaming services are placed where resources are available using legacy optimization policies and relying on over-provisioning to fulfill the QoS required by the user for an optimal gaming experience. We firmly anticipate that the future of gaming services will demand the distribution of game server software, operating within game engines, across cloud infrastructure. Accommodating this shift while prioritizing the gaming experience and resource optimization presents a substantial challenge, particularly for conventional monolithic game engines.

The primary research challenge at present revolves around the design and development of a new generation of game engines capable of harnessing the dynamic resource allocation provided by cloud architecture. These next-generation game engines are envisioned as being composed of discrete modules, each handling a specific aspect of the game's functioning, named Game Engine Modules (GEMs) [94]. GEMs can be launched on-demand, dispersed throughout the cloud continuum based on resource availability, and strategically located to reduce gaming latency for the user [23].

Strategically positioning these modules within the cloud continuum is an intricate endeavor that directly impacts the system's performance and user experience. One distinct feature of GEM placement, which differentiates it from virtual machine placement, is that GEMs can be utilized by multiple players simultaneously. As a result, the placement process necessitates a continuous, synchronous backstream for all participating players. This situation is comparable to synchronously streaming a video to a multitude of viewers, with the unique twist that these viewers can "interact" with the video they are watching, but are unable to reverse the stream.

The GEM placement challenge entails determining the most advantageous positions for these modules within the cloud continuum. The aim is to minimize

the latency perceived by the gamer, providing a smooth and seamless game-play experience. However, this is a complex task that presents various challenges. While the dynamic nature of cloud resources complicates the GEM placement problem, the focus is on the challenges associated with the varying requirements of different GEMs, such as compute power, memory, and network bandwidth. An equally critical consideration is the network latency between the GEMs and the gamers. Optimal GEM placement should prioritize latency reduction, ensuring a seamless and highly responsive gaming experience. However, achieving this requires a thorough understanding of the network structure and the geographical distribution of players. It is worth mentioning that, even the aforementioned cloud gaming platforms are closed-source and do not disclose the techniques they employ for the optimal placement of the monolithic games they offer.

In this study, we define the optimal GEM placement problem, considering the bandwidth and resource needs of each GEM. Initially focusing on a single-player per-game session, our approach is then extended to accommodate multiple players within the same game arena, both scenarios being of significant practical relevance. Our model aims to maximize the acceptance of GEM placement requests while minimizing the delay experienced by players. The first optimization goal leads to both optimized resource utilization and enhanced player satisfaction due to the acceptance of more play requests.

To solve this optimization problem, we develop a two-phase approximation algorithm, denoted as MAximized-Placement with MINimized-Delay (MAP-MIND), which runs on a set of GEM requests. In the first phase, MAP-MIND places the GEMS to maximize the number of accepted GEMs. In the second phase, MAP-MIND revises the placement to minimize delay. Subsequently, we introduce MAP-MIND*, a faster variant of MAP-MIND, which serves as a practical alternative for real-world applications, albeit with a slight trade-off in terms of delay compared to the original MAP-MIND. We compare MAP-MIND and MAP-MIND* with the optimal solver, which suffers from limited scalability. Given that no existing solutions share our specific optimization objective, we also compare them with standard placement algorithms that either minimize delay (derived from the scenario of Virtual Machine placement in cloud computing systems) or maximize placement through bin-packing standard approaches. Through extensive simulations, we show that MAP-MIND closely approximates the optimal solution achieved by the solver in terms of both the delay and the number of accepted GEMs.

Furthermore, it also outperforms the other algorithms in terms of delay, number of accepted GEMs, and running time. On the other hand, MAP-MIND* reduces the execution time at the cost of degraded delays. This demonstrates the effectiveness of our approach in tackling the GEM placement problem. It's important to highlight that our model can handle scenarios involving dynamic resource usage. However, our resource planning strategy is robust, primarily based on estimating the worst-case resource requirements for the games.

The remainder of this chapter is organized as follows. In Sec. 4.2, we discuss the related work in the state-of-the-art literature. Sec. 4.3 describes the architecture of distributed game engines. In Sec. 4.4, we provide a formal description of the problem and we formulate a mathematical optimization model, for the scenario of both a single player per GEM and many players per GEM (i.e., a game arena scenario). In Sec. 4.5, we propose MAP-MIND and MAP-MIND* as approximation algorithms to solve the problem presented in Sec. 4.4. In Sec. 4.6, we assess by simulation the performance of both MAP-MIND and MAP-MIND* by comparing them with some proposed variants and with the state-of-the-art approaches. Finally, we draw our conclusions in Sec. 3.8.

4.2 Related work

Various studies have focused on optimizing cloud gaming platforms by enhancing server infrastructure and communication channels. The research in [95–99] addresses GPU sharing, resource isolation, and improvements to existing clouds and games for efficiency. Distributed architectures, including P2P solutions [100, 101], edge server deployment strategies [102, 103], and distributed cloud gaming architectures-[104, 105], have also been investigated. Additionally, other works explored greedy heuristics for MMOG resource provisioning[106] and neural network-based predictions for VM deployment [107]. Regarding communication channels, some studies have explored data compression schemes such as cooperative video encoding [108, 109] and RoI-based rate control [110]. Rendering information and graphics compression techniques were employed to enhance encoding efficiency [98, 111–118]. Adaptive transmission techniques, including video adaptation modules [119, 120] and dynamic rate selection algorithms [121, 122], were used to adjust gaming video transmission based on available bandwidth and network conditions. Additionally,

novel frameworks like APHIS [123] optimized high frame rate video streaming, while object selection algorithms [124] reduced processing time for scene rendering, improving overall streaming efficiency. While they optimize the infrastructure to improve overall efficiency, they do not directly address the game placement problem within the provided infrastructure.

The Co-Location Placement (CLP) strategy, utilized in various studies, aligns with our approach where we treat all GEMs of a multi-GEM game as a single large abstract GEM [125–127]. Authors in [125] justify CLP by addressing seamless handovers for mobile users moving between providers using a fog-based authentication mechanism. The work in [126] uses CLP by treating each task as an independent inseparable request, each served by a single data center. In [127], CLP is justified assuming each task in the vehicular networks context can only be offloaded to one resource.

The GEM placement problem is similar to the VNF (Virtual Network Function) placement in cloud systems. The problem of optimal VNF placement is typically NP-hard, making optimal solutions computationally challenging. For instance, in [128], a heuristic based on Integer Linear Programming (ILP) for VNF placement was introduced, akin to our two-phase approach. While effective for larger instances, its high execution time, due to the iterative call to the ILP solver, may curtail its practical application. As a result, machine learning techniques [129–131] or heuristic/meta-heuristic algorithms [132–136] are frequently employed to efficiently address the VNF placement challenge.

In [129], the authors presented an online VNF placement strategy for Edge Computing networks using Reinforcement Learning. This strategy diminished user delay by positioning VNFs closer to users but exhibited a high rejection rate. Another study in [130] integrated Deep Reinforcement Learning with Graph Neural Networks to enhance VNF placement, thereby reducing service request rejections and adapting to broader network types. Meanwhile, [131] utilized deep reinforcement learning for VNF placement to reduce the average end-to-end delay by allocating more VNFs in MEC. However, this approach did not account for maximizing VNF acceptance. The authors of [132] introduced a heuristic solution, MaxSR, which dynamically orchestrates services in 5G networks using a backtracking approach. In [133], a genetic algorithm-based heuristic was developed to minimize resource costs. Another study [134] proposed the MINI heuristic algorithm to optimize resource

utilization, showing marginal improvements over another heuristic based on the Genetic Algorithm (GA) for VNF acceptance. Some works focused on offline batch request placements. For example, [135] proposed an offline simulated annealing-based heuristic for placing VNFs for delay-sensitive requests. The heuristic introduced in [136], termed Previous Window Deployment (PWD), is based on Learn and Deploy (LAD) and aims to maximize user service. Notably, none of these studies contemplated shared services among users, which corresponds to the multi-player case in our gaming scenario.

Several studies have explored VNF sharing, also better described as reusability, within the context of the VNF placement problem [137–145], mimicking the multi-player gaming scenario. For instance, [137] explored the VNF placement within a single physical node, allowing VNF sharing across multiple Network Services (NSs). They proposed a heuristic algorithm that prioritizes NSs on shared VNF instances, ensuring processing delays meet the required thresholds. [140] tackled the dynamic VNF placement challenge, deciding whether to migrate existing VNFs or deploy new ones to optimize VNF reuse. The objective was to enhance the network operator's profitability, and a column generation-based algorithm was suggested for this purpose. In [142], a dynamic heuristic algorithm was introduced to minimize overall network OPEX and physical resource fragmentation for the VNF placement issue, where multiple NSs can share VNF instances. Although these studies focus on VNF placement for NSs that incorporate VNF re-usability, none align with the multiplayer shared GEM challenge.

Several works have proposed heuristic-based algorithms emphasizing maximizing the number of accepted VNFs [136–138, 146–152, 145, 153]. For example, [146] introduced a potential game approach for VNF placement in satellite edge networks. Like our approach, they support VNF co-location, but they applied a distributed placement decision and did not consider shared VNFs. In [149], two heuristics, one greedy-based and the other Tabu search-based, were proposed to maximize profit through placement. They successfully increased the admission rate for offline placement but did not consider shared VNFs. Authors of [151] tackled the delay-aware VNF dynamic placement and routing problem by constructing a heuristic optimization data structure called VNF-splitted multi-stage edge-weight graph. Their algorithm demonstrated superior performance in average traffic acceptance rate compared to others. Like our approach, they considered a maximum tolerable delay but did not focus on minimizing the delay or considering shared VNFs. The study

in [153] presented an online heuristic framework named Holu, which aimed to solve the VNF placement problem by considering the centrality of the compute nodes and the power consumption of a VNF. They showed improvement in VNF acceptance but did not consider minimizing delay. Also, the shared VNF considered in their model is distinct in nature as users share resources but do not interact.

Some works have proposed heuristic algorithms for the VNF placement problem with a focus on delay minimization [150, 154–157]. For instance, in [150], an evolutionary algorithm was proposed to enhance various metrics for IoT devices' service placement. Their approach maximizes the use of the fog nodes for the placement, which in turn improves service delay and response times. However, they did not consider the shared VNF scenario. The work in [155] addressed the VNF placement problem in non-terrestrial networks by formulating it as a weighted graph-matching problem using a Linear Programming algorithm and the Hungarian-based algorithm. Their goal was to minimize delay and maximize resource utilization. Lastly, [157] introduced a genetic-based heuristic algorithm for service placement aimed at minimizing application delay. However, neither shared VNFs nor the maximization of VNF placement was targeted in these last two papers.

In summary, while the aforementioned related works have attempted to address various facets of our problem, to our knowledge, no existing research has holistically addressed all the requirements of our problem, especially concerning multiplayer GEMs. This distinction is significant, as the definitions of shared VNFs in current literature differ markedly from our approach.

4.3 CODEG: a Cloud-Oriented Distributed Game Engine Approach

The Cloud-Oriented Distributed Engine for Gaming (CODEG) model, as presented in [23], introduces a groundbreaking approach to cloud gaming by fully utilizing the capabilities of modern cloud infrastructure. This model redefines game engines as network-wide operating systems, which are partitioned into functional units known as GEMs. A GEM represents a fundamental element of a Game Engine (GE) (as demonstrated in Fig. 4.1) and is activated whenever its specific functionality is needed. Each GEM serves as a basic component of the game engine and is eventually linked

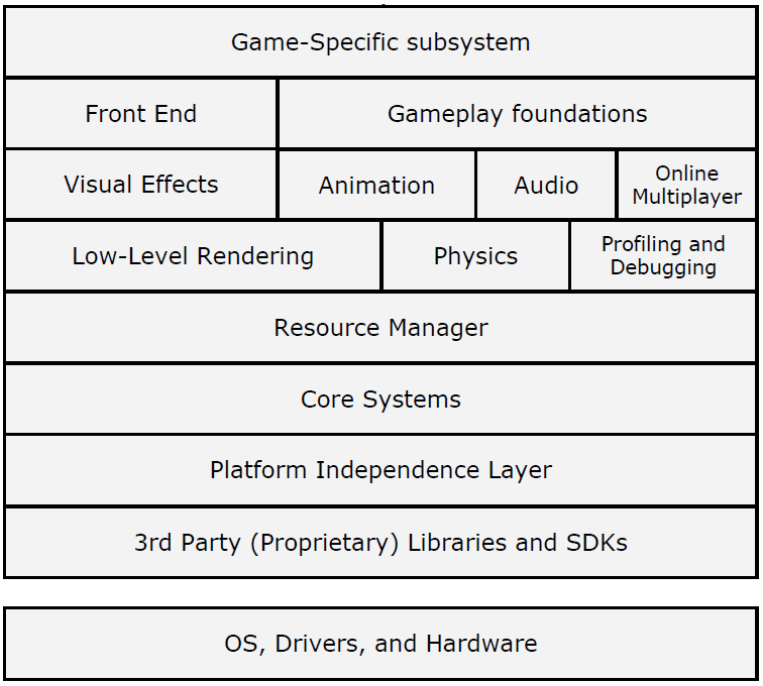


Fig. 4.1 Summary of a standard game engine (GE) architecture [94].

to other GEMs to implement the whole game service. Similar to conventional Game Engines, where various components work together to create complex functionalities, individual GEMs must be interconnected or 'chained together' to implement more sophisticated features and behaviors in the game. As a very simple example, in a multiplayer shooting game, various GEMs collaborate to deliver a seamless gaming experience. The Audio GEM manages voice chat functionality, while the Network GEM handles network connectivity. Player GEMs represent individual players and control their interactions within the game world, while the Weapon GEM manages weapon mechanics. This modular approach allows for greater flexibility and scalability, as GEMs can be deployed based on real-time game demand.

Fig. 4.2 shows an example of a game session in which GEMs are grouped into different functionalities necessary to implement the game service. The request for new game sessions triggers the request to start new GEMs. All the requests that arrive during a predefined observation time window are allocated based on the placement algorithm.

The CODEG model operates under the assumption that modern cloud infrastructures are composed of multiple Compute Nodes (CNs). These CNs could be physical

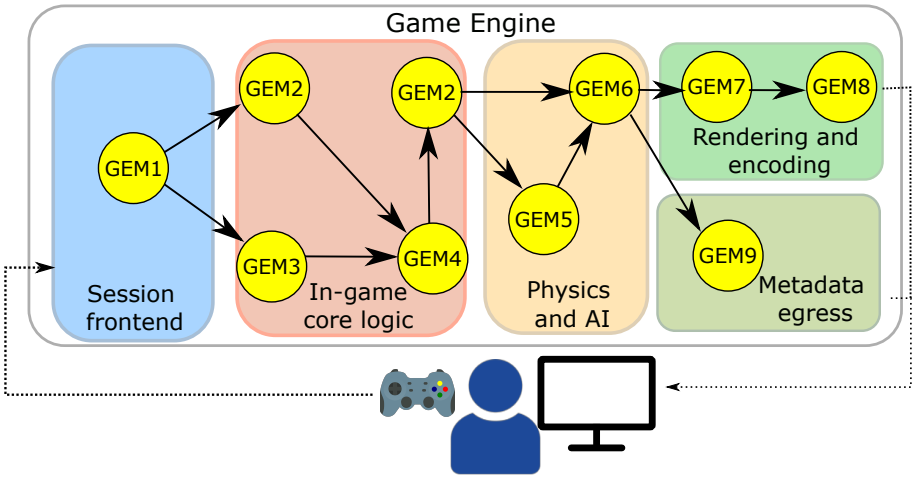


Fig. 4.2 Example of game session workflow in CODEG [23].

servers in a data center or virtual servers in a cloud environment, as shown in Fig. 4.3. The model utilizes this distributed architecture to host the GEMs, efficiently spreading the workload of the game engine across multiple CNs.

The model also assumes that all players are connected to the same network provider and that the data center hosting the game engine is optimally located to minimize delay. This is vital for ensuring a smooth gaming experience, as significant delays can result in interruptions and disruptions during gameplay. Furthermore, the CODEG model takes advantage of advanced network virtualization techniques. These techniques allow for the creation of logical networks with specific QoS levels. This ensures that game-related traffic has guarantees of minimum bandwidth and maximum delay. In the CODEG model, the servers hosting the GEMs are connected through the network. The nodes and edges of this network represent the CNs and the logical or physical links between them, respectively.

Each game session in the CODEG model has a unique maximum tolerable delay. This is defined based on the nature of the game and is set to maximize fairness among players of the same multiplayer game session and to satisfy the Quality of Experience (QoE) perceived by each player.

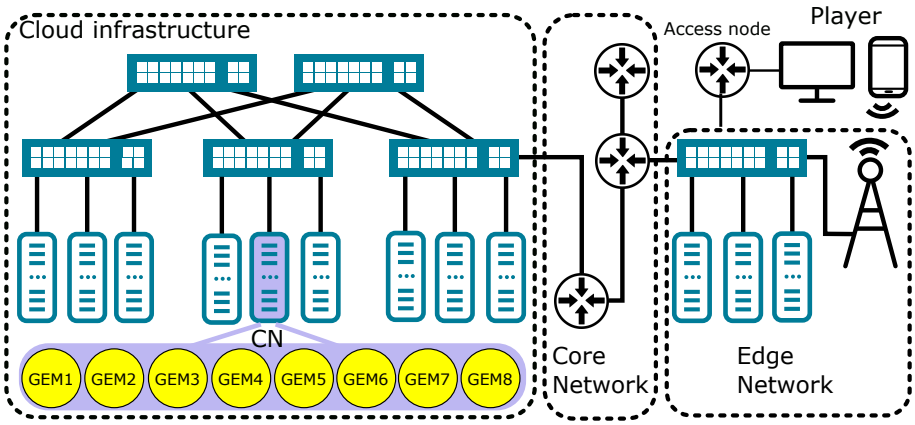


Fig. 4.3 Example of compute nodes (CN) and GEM co-location.

4.4 Optimal GEM placement

In this section, we describe our system model, and then we formulate the placement problem, whose objective is to maximize the number of accepted game sessions while minimizing the overall delay experienced by their players. This combined objective function is designed to maximize the profits of the game providers while maximizing the QoE perceived by the players as well. Each game can be implemented by composing multiple GEMs. However, for the sake of simplicity, this work considers games that contain a single GEM or multiple GEMs that are co-located, as shown in Fig. 4.3. This simplification is necessary due to the absence of practical examples of games designed through GEMs, which provide clear Directed Acyclic Graphs (DAGs) for our model. By focusing on scenarios involving either one single GEM or multiple co-located GEMs, we aim to establish foundational principles and insights despite the lack of real-world examples. Starting with simpler scenarios allows us to systematically explore design choices and analytical techniques before addressing the complexities of real-world applications. As the field evolves and practical examples emerge, refining the model to encompass a wider range of GEM configurations will be essential for its applicability to practical game design. We now introduce the adopted notation, reported also in Table 4.1. Let $\mathscr{T} = (N, E)$ represent the graph describing the network topology, where E is the set of network links and N is the set of network nodes, which can be either CNs or access nodes, where the players connect to the network. We define K as the set of resource types available in the nodes, including CPU, memory, and storage. We use r_n^k to denote the amount of resources of type $k \in K$ available at node $n \in N$.

Table 4.1 Notation.

Notation	Description
N	Set of network nodes (i.e., compute nodes or access nodes)
E	Set of network links
$\mathscr{T}$	Network topology
K	Types of resources available at the nodes
r_n^k	Amount of resource of type k available at node n
ρ_s^k	Amount of resource of type k needed by game session s
S	Set of game sessions to place
GEM_s	Group of co-located GEM running game session s
P	Set of all the players
P_s	Set of players in the game session s
$h(p)$	Access node of player p
$d(p,n)$	Delay experienced by player p if GEM_s is placed in node n
$d_{\max}(s,n)$	Maximum delay experienced by players in P_s if GEM_s is placed in node n
$d_{\text{tot}}(s,n)$	Summation of the delays experienced by the players in P_s if GEM_s is placed in node n
$d_s^{\max}$	Maximum delay allowed for each player of the game session s
$N(s)$	Set of nodes for which the maximum delay is satisfied for all the players in the game session s
λ_s	Reserved bandwidth for game session s
b_{uv}	Available bandwidth on link (u,v)
$\mathscr{P}(s,n)$	Set of links used by the traffic for game session s if GEM_s is placed in node n
ϕ_{sn}^{uv}	$= 1$ if the link (u,v) belongs to $\mathscr{P}(s,n)$, 0 otherwise
x_{sn}	$= 1$ if GEM for game session s is placed at node n, 0 otherwise

Let S be the set of game sessions to place. We assume multiple players for each game session; as a special case, a game session could be played by a single player. We also assume a single GEM or a group of co-located GEMs per game session s, and we denote it by GEM_s. Let P be the set of all the players, whereas $P_s \subseteq P$ be the subset of players associated with the game session $s \in S$. Let ρ_s^k, for any $k \in K$, be the amount of resource of type k that is required by GEM_s. Note that such value may depend on the actual number of players (e.g., according to a proportional law), serving as an upper bound for the worst-case scenario of the required resource amount. Let $h(p) \in N$ be the access node of player $p \in P$. Assume now that GEM_s is placed on node $n \in N$. Let $d(p,n)$ be the delay experienced

by player p, computed as the sum of the communication delay from $h(p)$ to n, the GEM$_s$ processing delay, and the communication delay from n to $h(p)$. Let $d_{\max}(s,n)$, for any $s \in S$, be the maximum delay experienced by any of the players in P_s: $d_{\max}(s,n) = \max_{p \in P_s} d(p,n)$. Similarly, let $d_{\text{tot}}(s,n)$ be the summation of all the delays experienced by each of the players in P_s: $d_{\text{tot}}(s,n) = \sum_{p \in P_s} d(p,n)$. Note that it is possible to pre-compute the set of all $d_{\max}(s,n)$ and $d_{\text{tot}}(s,n)$ with $O(N^2|P|)$ operations. Let $d_s^{\max}$ be the maximum allowed delay for game session s, which has been devised to satisfy the expected performance at the endpoint. This parameter expresses the main threshold related to the QoS constraint. Given $d_s^{\max}$, it is possible to pre-compute the set $N(s) \subseteq N$ of nodes where GEM$_s$ can be placed such that all the players experience a delay compatible with the maximum allowed one: $N(s) = \{n \in N \mid d_{\max}(s,n) \leq d_s^{\max}\}$.

We assume single-path routing according to the shortest path. Let $\mathscr{P}(s,n)$ be the overall routing paths, i.e., the set of links used to route traffic from all the access nodes of the players in the games session s to node n and back from n to all the access nodes. We define an indicator function ϕ_{sn}^{uv} equal to 1 iff the link (u,v) belongs to $\mathscr{P}(s,n)$. We assume that the bandwidth is reserved for each game session through a dedicated network slice. The bandwidth reserved for the game session s along the links belonging to the routing paths (i.e., for any $e \in \mathscr{P}(s,n)$) is denoted as λ_s, and b_{uv} represents the available bandwidth on link $(u,v) \in E$. Notably, λ_s is equal to the maximum bandwidth, among all the links in $\mathscr{P}(s,n)$, required by the aggregate traffic for game session s, and it typically grows with $|P_s|$. Note that all the parameters defined so far are pre-computed based on the network topology and the maximum allowed delay.

The decision variables are denoted by binary variables x_{sn}, defined as equal to 1 iff GEM$_s$ is placed at node n. Whenever there are not enough resources in a CN for a game session or the corresponding constraint on the maximum delay cannot be met, the game session request is dropped, and all the corresponding players abort the game.

We formulate the problem as an Integer Linear Programming (ILP) model. The problem can be viewed as a generalized assignment problem, as the sole decision revolves around identifying the CN where each game session GEM should be placed.

The optimal GEM placement for multi-player GEMs can be formulated as follows:

$$\max_{\{x_{sn}\}}(\alpha \sum_{s\in S}\sum_{n\in N(s)} x_{sn} - \sum_{s\in S}\sum_{n\in N(s)} x_{sn} d_{\text{tot}}(s,n)) \tag{4.1}$$

subject to

$$\sum_{n\in N(s)} x_{sn} \leq 1, \qquad \forall\, s \in S \tag{4.2}$$

$$\sum_{s\in S: n\in N(s)} \rho_s^k x_{sn} \leq r_n^k, \qquad \forall\, k \in K, n \in N \tag{4.3}$$

$$\sum_{s\in S, n\in N(s)} \lambda_s \phi_{sn}^{uv} x_{sn} \leq b_{uv}, \qquad \forall\, (u,v) \in E \tag{4.4}$$

$$x_{sn} \in \{0,1\}, \qquad \forall s \in S, n \in N(s) \tag{4.5}$$

The objective function (4.1) combines two components and aims to maximize the total number of accepted game sessions (first term), and secondly, to minimize the overall delay experienced by the players (second term). Indeed, α is a large enough constant such that the objective function will always prefer solutions with a larger number of accepted game sessions, independently from the overall delay. To achieve this, we choose $\alpha = |P|(2d^{\text{net}} + d^{\text{proc}})$, where d^{net} is the diameter of the network in terms of delay and d^{proc} is the maximum processing delay per game session. Constraint (4.2) assigns the GEM for each game session to at most one CN; indeed, a game session may be dropped due to a lack of resources in the CNs compatible with the related maximum delay requirement. Constraint (4.3) ensures that the CN resources are not exceeded. Constraint (4.4) ensures that the reserved bandwidth for each game session does not exceed the network link capacity. Finally, (4.5) defines the domain of the decision variables. We now claim the following:

Property 1. *The optimal GEM placement problem is NP-hard.*

Proof. We show the problem complexity by reduction from the multiple knapsack problem, which is known to be NP-hard, in a weak sense. By assuming no constraints on bandwidth (i.e., $b_{uv} = \infty, \forall (u,v) \in E$), (4.4) can be omitted. By assuming no constraints on the maximum delay, all CNs become delay-eligible for all GEMs (i.e., $N(s) = N$). Now, the placement of GEMs is decided solely by the resources on each CN. We can simplify these resources to just one type, such as CPU. The problem

of optimizing GEM placement then becomes identical to the multiple knapsack problem, where N represents the set of knapsacks, $\mathscr{S}$ the set of items, the CN's CPU capacity represents the capacity of each knapsack, and the CPU demand of the GEMs represents the sizes of the items to be selected, and the profit for item s in knapsack n is represented by $\alpha - d_{\text{tot}}(s,n)$. □

4.5 Approximated algorithms

Property 1 implies that an optimal solver for the GEM placement problem is likely not scaling well with problem instance size, and this motivates the design of an approximated approach able to solve large instances of the problem within a constrained timeframe. In response, we have introduced two approximate search algorithms inspired by local search methods, MAP-MIND and MAP-MIND*, each striking a distinct balance between efficiency and computational complexity.

We note that our devised approach is general and implementation-independent. It can be integrated with any resource orchestrator, e.g., Kubernetes for the case of virtualization obtained with containers.

4.5.1 The MAP-MIND algorithm

We are interested in scalable algorithms able to cope with large input instances. We propose an approximate search algorithm called *MAximize-Placement and MINimize-Delay* (*MAP-MIND*). *MAP-MIND* aims to simplify and solve the problem of optimized placement by addressing the multi-criteria objective function (4.1) through two distinct phases. The first phase, denoted as MAP, is devoted to maximizing the acceptance of the game sessions' requests, and the second one, denoted as MIND, aims at minimizing the average delay experienced by the players. Coherently with (4.1), MAP-MIND prioritizes maximizing acceptance over minimizing the average delay.

In the first phase, the algorithm strives to accommodate as many game session requests as possible, regardless of the incurred delay. To do so, the GEMs are sorted based on the maximum delay allowed by their players (i.e., $d_s^{\max}$ for GEM_s). This choice is motivated by the fact that it gives more chances to the GEMs with small

delay requirements to be placed in nodes closer to the access nodes compared to other GEMs with large delay requirements. The algorithm starts to place the sorted GEMs using a Best-Fit approach [158], where the GEMs' resource requirements are considered with respect to the available resources on the computing nodes. The algorithm divides the sum of each resource requirement by the total available resources of the same type within the network. The resource with the highest ratio is deemed the *decision-maker resource* by the Best-Fit algorithm. This algorithm prioritizes nodes with smaller available quantities of the decision-maker resource. As a result, this approach optimizes the utilization of available resources throughout the network.

For the second phase, the placement solution of the first phase is used as the initial solution to minimize the delays. This phase is iterative and adopts two possible steps to modify the current solution: (i) *move*, (ii) *swap*. A node n is said to be eligible for GEM_s if it belongs to $N(s)$, it has enough resources to run GEM_s, and there is enough bandwidth along the routing path, during the current iteration of the algorithm. The *move* step tries to move each placed GEM to another eligible node, if available, which gives the players of that GEM the minimum experienced delay. This procedure will be repeated until no GEM movement is available to reduce the players' experienced delay. In the subsequent *swap* step, the algorithm tries to swap the position of each GEM with another GEM to reduce the average delay experienced across all players. The reason behind prioritizing the *move* step over the *swap* step is due to the Best-Fit approach used in the first phase, which may lead to having some empty nodes (especially at low loads) that could be efficiently used by the *move* steps rather than the *swap* steps.

The pseudocode of MAP-MIND is provided in Fig. 4.4. It takes as input S with the game session requests, the graph describing the network topology, the available bandwidth on the links, and the available resources on the nodes.

In the first phase, after the initialization of the placement variables and a temporary placement set, the decision-maker resource will be determined (ln. 3-5). Now, GEMs are sorted in increasing order based on their maximum tolerable end-to-end delay (ln. 6). Then, for each GEM in sorted order, the best destination node and its score will be initialized (ln. 7-8). Then, all the nodes are checked to find the eligible node with the minimum available decision-maker resource, coherently with a Best-Fit approach (ln. 9-11). The GEM and the selected node are then added to the set of

Input: S: Set of game session requests
Input: $\mathscr{T} = (N, E)$: Network topology
Input: Available bandwidths and resources in the network
procedure MAP-MIND
  `PHASE 1: Maximize acceptance (MAP)`
  $Y \leftarrow \emptyset$ ▷ Initialize temporary placement
  $x_{sn} \leftarrow 0\ \forall s \in S, n \in N$ ▷ Initialize GEMs placement
  $k_1 = \underset{k \in K}{\operatorname{argmax}} \left(\frac{\sum_{s \in S} \rho_s^k}{\sum_{n \in N} r_n^k} \right)$ ▷ Identify the relative bottleneck resource
  sort $s \in S$ w.r.t. $d_s^{\max}$ in incr. order ▷ Sort by maximum tolerable delay
  for $s \in$ sorted S **do** ▷ For each game session request s
    $n' \leftarrow -1, \text{score} \leftarrow \infty$ ▷ Initialize best node and its score
    for $n \in N$ **do** ▷ Find the best eligible node
      if (n is eligible for s) $\wedge$ ($r_n^{k_1} <$ score) **then**
        score $\leftarrow r_n^{k_1}, n' \leftarrow n$ ▷ Update selected node and score
    if $n' \neq -1$ **then** ▷ If the best node is found
      $Y \leftarrow Y \cup \{(s, n')\}$ ▷ Update the temp placement
      $x_{sn'} \leftarrow 1$ ▷ Update placement variables
      Update the available bandwidth along $\mathscr{P}(s, n')$ and $r_{n'}^k\ \forall k \in K$
  `PHASE 2-1: Minimize average delays (Move step)`
  sort Y w.r.t. $d_{\text{tot}}(s, n)$ in desc. order ▷ Sort by total delay
  Improved = True
  while Improved = True **do** ▷ Iterate until no improvement is experienced
    Improved = False
    for $(s, n) \in$ sorted Y **do** ▷ For each GEM
      $n'' \leftarrow -1, \Delta \leftarrow 0$ ▷ Initialize best destination node and its score
      for $n' (\neq n) \in N$ **do**
        if (n' is eligible for s) $\wedge$ ($d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n') > \Delta$) **then**
          $n'' \leftarrow n', \Delta \leftarrow d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n')$ ▷ Update node, score
      if $\Delta \neq 0$ **then**
        $Y \leftarrow Y \setminus \{(s, n)\} \cup \{(s, n'')\}$ ▷ Move from n to n''
        $x_{sn''} \leftarrow 1, x_{sn} \leftarrow 0$ ▷ Update placement variables
        Update the available bandwidth along $\mathscr{P}(s, n)$ and $\mathscr{P}(s, n'')$
        Update $r_n^k, r_{n''}^k\ \forall k \in K$
        Improved = True
  `PHASE 2-2: Minimize average delays (Swap step)`
  Improved = True
  while Improved = True **do** ▷ Iterate until no improvement is experienced
    Improved = False
    for $(s, n) \in$ sorted Y **do** ▷ For each GEM
      $(s'', n'') \leftarrow (-1, -1), \Delta \leftarrow 0$ ▷ Initialize best swap and its score
      for $(s', n') (\neq (s, n)) \in$ sorted Y **do** ▷ For each GEM
        if (n' is eligible for s) $\wedge$ (n is eligible for s') **then**
          $\Delta' \leftarrow d_{\text{tot}}(s, n) + d_{\text{tot}}(s', n') - d_{\text{tot}}(s, n') - d_{\text{tot}}(s', n)$
          if $\Delta' > \Delta$ **then**
            $(s'', n'') \leftarrow (s', n')$ ▷ Update candidate
            $\Delta \leftarrow \Delta'$ ▷ Update score
      if $\Delta \neq 0$ **then**
        $Y' \leftarrow Y' \setminus \{(s, n), (s'', n'')\} \cup \{(s, n''), (s'', n)\}$ ▷ Swap
        $x_{sn} \leftarrow 0, x_{s''n''} \leftarrow 0$ ▷ Update old placement variables
        $x_{sn''} \leftarrow 1, x_{s''n} \leftarrow 1$ ▷ Update new placement variables
        Update the available bandwidth along $\mathscr{P}(s, n)$ and $\mathscr{P}(s'', n'')$
        Update the available bandwidth along $\mathscr{P}(s'', n)$ and $\mathscr{P}(s, n'')$
        Update $r_n^k, r_{n''}^k\ \forall k \in K$
        Improved = True
  return $\{x_{sn}\}_{s \in S, n \in N}$

Fig. 4.4 Pseudocode of `MAP-MIND`.

accepted GEMs, while the related placement variable and the available bandwidth and resources of the network are updated accordingly (ln. 12-15). If no eligible node is found, the GEM's related placement variables will remain unassigned.

Then, the second phase starts. During the first step (*move*), the accepted GEMs are sorted based on their average players' delay (ln. 17). For each accepted GEM, the algorithm tries to find another eligible node that minimizes the end-to-end delay compared to the current GEM placement. If such a node is found, the GEM placement is updated accordingly, and the resources of the source and destination nodes along with the corresponding bandwidths are updated (ln. 19-31). This process is repeated until no improvement *move* is found.

During the second step (*swap*), the algorithm finds pairs of accepted GEMs whose swap will be feasible in terms of maximum delay, bandwidth, and resources, and the overall delay will be decreased (ln. 33-43). If such a pair is found, it updates the placement allocation and the corresponding resource availability (i.e., bandwidth, memory, CPU, and storage) (ln. 45-50). This process is iterated until no improving swaps can be executed.

4.5.2 The MAP-MIND* algorithm

We introduce a simpler version of MAP-MIND, termed MAP-MIND*. You can find its pseudocode in Fig.4.5. It follows the structure of MAP-MIND but merges the move (ln.19-27) and swap (ln. 29-41) steps within each iteration. MAP-MIND* differs from MAP-MIND in how it handles these operations; rather than executing them one after the other, MAP-MIND* performs them simultaneously, intertwining them within each iteration.

This change in the execution of swap and move operations brings about a significant alteration in the algorithm's behavior. By mixing and executing swap and move operations concurrently, MAP-MIND* aims to enhance the optimization process's efficiency. This strategy streamlines the execution, potentially reducing the required number of iterations for convergence. Moreover, the concurrent execution of swap and move operations in MAP-MIND* leads to a reduction in computational complexity, as indicated in our writing. This complexity reduction is advantageous, potentially resulting in faster convergence and reduced resource needs, making MAP-MIND* a more efficient choice for specific optimization tasks.

Input: S: Set of game session requests
Input: $\mathscr{T} = (N, E)$: Network topology
Input: Available bandwidths and resources in the network
procedure MAP-MIND*
  `PHASE 1: Maximize acceptance (MAP)`
  $Y \leftarrow \emptyset$ ▷ Initialize temp placement
  $x_{sn} \leftarrow 0 \quad \forall s \in S, n \in N$ ▷ Initialize GEMs placement
  $k_1 = \underset{k \in K}{\operatorname{argmax}} \left(\frac{\sum_{s \in S} \rho_s^k}{\sum_{n \in N} r_n^k} \right)$ ▷ The relative bottleneck resource
  sort $s \in S$ w.r.t. $d_s^{\max}$ in incr. order ▷ Sort by maximum tolerable delay
  for $s \in$ sorted S **do** ▷ For each game session request s
    $n' \leftarrow -1, \text{score} \leftarrow \infty$ ▷ Initialize best node and its score
    for $n \in N$ **do**
      if (n is eligible for s) $\wedge$ ($r_n^{k_1} <$ score) **then**
        $n' \leftarrow n, \text{score} \leftarrow r_n^{k_1}$ ▷ Update selected node and score
    if $n' \neq -1$ **then**
      $Y \leftarrow Y \cup \{(s, n')\}$ ▷ Update the temp placement
      $x_{sn'} \leftarrow 1$ ▷ Update placement variables
      Update the available bandwidth along $\mathscr{P}(s, n')$ and $r_{n'}^k \ \forall k \in K$
  `PHASE 2: Minimize average delays (MOVE or SWAP)`
  sort Y w.r.t. $d_{\text{tot}}(s, n)$ in desc. order ▷ Sort by total delay
  for $(s, n) \in$ sorted Y **do**
    $n'' \leftarrow -1, \Delta \leftarrow 0$ ▷ Initialize best destination node and its score
    for $n' (\neq n) \in N$ **do** ▷ Check for possible MOVE
      if (n' is eligible for s) $\wedge$ ($d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n') > \Delta$) **then**
        $n'' \leftarrow n', \Delta \leftarrow d_{\text{tot}}(s, n) - d_{\text{tot}}(s, n')$ ▷ Update node, score
    if $\Delta \neq 0$ **then** ▷ MOVE
      $Y \leftarrow Y \setminus \{(s, n)\} \cup \{(s, n'')\}$ ▷ Move from n to n''
      $x_{sn''} \leftarrow 1, x_{sn} \leftarrow 0$ ▷ Update placement variables
      Update the available bandwidth along $\mathscr{P}(s, n)$ and $\mathscr{P}(s, n'')$
      Update $r_n^k, r_{n''}^k \ \forall k \in K$
    else ▷ Check for possible SWAP
      $(s'', n'') \leftarrow (-1, -1), \Delta \leftarrow 0$ ▷ Initialize best swap and its score
      for $(s', n') (\neq (s, n)) \in$ sorted Y **do** ▷ For each GEM
        if (n' is eligible for s) $\wedge$ (n is eligible for s') **then**
          $\Delta' \leftarrow d_{\text{tot}}(s, n) + d_{\text{tot}}(s', n') - d_{\text{tot}}(s, n') - d_{\text{tot}}(s', n)$
          if $\Delta' > \Delta$ **then**
            $(s'', n'') \leftarrow (s', n'), \Delta \leftarrow \Delta'$ ▷ Update GEM, score
      if $\Delta \neq 0$ **then** ▷ SWAP
        $Y' \leftarrow Y' \setminus \{(s, n), (s'', n'')\} \cup \{(s, n''), (s'', n)\}$
        $x_{sn} \leftarrow 0, x_{s''n''} \leftarrow 0$ ▷ Update old placement variables
        $x_{sn''} \leftarrow 1, x_{s''n} \leftarrow 1$ ▷ Update new placement variables
        Update the available bandwidth along $\mathscr{P}(s, n)$ and $\mathscr{P}(s'', n'')$
        Update the available bandwidth along $\mathscr{P}(s'', n)$ and $\mathscr{P}(s, n'')$
        Update $r_n^k, r_{n''}^k \ \forall k \in K$
  return $\{x_{sn}\}_{s \in S, n \in N}$

Fig. 4.5 Pseudocode of `MAP-MIND*`.

However, it's important to acknowledge that this optimization involves a trade-off; while MAP-MIND* may decrease running time compared to MAP-MIND, it may introduce a slightly longer delay. This trade-off should be carefully evaluated in practical scenarios, considering the benefits of reduced computational complexity against the potential delay increase. Further discussions on the distinctions between MAP-MIND* and MAP-MIND in executing swap and move operations, along with the implications for computational complexity and performance, will follow in subsequent sections.

4.6 Numerical evaluation

4.6.1 Simulation methodology

To evaluate the performance of our proposed algorithms, we developed an event-driven simulator using Python 3.8 on an Ubuntu 20.04.6 LTS system with 16GB RAM and an 8X Intel Core-i5-10210U-1.60GHz CPU. We also run an optimal solver (OPT) obtained by implementing the ILP formulation presented in Sec. 4.4 using AMPL as modeling language and Cplex 12.8.0 as a solver. The solver runs on an Ubuntu 18.04.6 LTS system with 32GB of RAM and a 12X Intel Core-i7-8700-3.20GHz CPU.

The network topology is a random geometric graph with a specified number of nodes (N) and average degree (g). We only considered connected instances of the graph. The edges of the graph represent the communication links between pairs of nodes, and the propagation delays are set proportional to the link lengths. For generality, we normalized the delay of each link to the average of all shortest paths in the same graph. We assumed a negligible processing delay for all GEMs at each node and null network access delay, since for simplicity (but without loss of generality) we defined $d_s^{\max}$ for game session s at the net of the processing delays and access delays. We also assumed unlimited bandwidth and routing based on the shortest path. We considered two main settings for the maximum tolerable delay of the GEMs: (i) No Delay Constraint (NDC) in which $d_s^{\max} = \infty$ to model all the games for which the perceived delay is not a relevant constraint (e.g., chess); (ii) Uniform Delay Constraint (UDC) in which $d_s^{\max}$ is uniformly distributed between 0 and maximum Round Trip Time (RTT) in the network graph, to model a large variety of game

scenarios with different sensitivity to delays. Players' access nodes for each game session are selected uniformly from the available nodes, where the same node can be selected more than once.

Each node is equipped with CPU, memory, and storage capacities set to 5 GHz, 32 GB, and 512 GB, respectively. For every GEM, the requirements for CPU, memory, and storage are uniformly distributed between 0 and 1 GHz, 0 and 1 GB, and 0 and 1 GB, respectively. We determine the Utilization Factor (UF) based on the system's bottleneck resource. According to the above scenario settings, we expect the CPU to be such a bottleneck. Our choice is influenced by the observation that, in real-world scenarios, one resource will invariably become the limiting factor in placement decisions. While our primary focus is on the bottleneck resource, we also factor in other potential non-bottleneck resources in our requirements. This approach ensures that we consider the influence of all resources on the algorithms. To achieve the desired UF, we generate GEMs sequentially. This generation continues until the cumulative bottleneck resource requirements of the produced GEMs in S either match or surpass the product of the target UF and the total CPU capacity across the entire network.

4.6.2 Test Scenarios

We explore the following test scenarios:

- Scenario 1: *Varying Number of Players per GEM* - We examine the influence of changing the number of players per GEM, i.e., for each game session, varying it in the set $\{1, 2, 10, 50\}$, with $N = 32$, $g = 4$, and UF $= 0.8$. The choice of UF = 0.8 aims to balance network resource utilization, avoiding both under-utilization and over-utilization, thereby optimizing resource allocation and performance. This setting ensures significant network utilization without pushing it to its limits, maximizing efficiency while maintaining reliability. Similarly, the selection of an average degree (g) of 4 is motivated by the goal of efficiently balancing local and global connectivity in future networks, ensuring robustness, respecting resource constraints, and enabling scalability. This configuration fosters reliability, stability, and effective information dissemination. We chose these values based on modern game genres. Single-player games like Red Dead Redemption II [159] and Horizon Forbidden West [160] often

require substantial resources due to large maps or complex AI. Two-player games can be competitive, like Street Fighter 6 [161] and FIFA 23 [162], or collaborative like Portal 2 [163]. In eSports, games typically involve 10 players with two teams of 5 competing, as seen in Dota 2 [164], League of Legends [165], Valorant [166], and Counter-Strike [167]. Battle royales like Fortnite [168], Player Unknown Battle Grounds (PUBG) [169], and Fall Guys [170] fall in the 50-player category, although Fortnite and PUBG can start with up to 100 players, and Fall Guys with 40 to 60, depending on the map. However, matchmaking timeout issues often reduce the starting player count, making an average of 50 players a reasonable assumption for our simulations.

- Scenario 2: *Varying Number of CNs* - We explore the system's behavior with different network sizes by varying $N \in \{16, 32, 48\}$, with $g = 4$, and UF $= 0.8$.

- Scenario 3: *Scaling CN resources* - We consider here a scenario where the CN resources are varied according to a scaling factor $\in [1, 4, 16]$ with respect to a base resource amount for the CNs equal to $\{1$ GHz$, 8$ GB$, 128$ GB$\}$ (i.e., CPU, memory, and storage, respectively); this allows to investigate the effect of scaling the CN resources in the infrastructure. We choose $N = 32$, $g = 4$, and UF $= 0.8$.

- Scenario 4: *Varying Average Graph Degree* - We investigate the effect of different connectivity levels within the network by varying $g \in \{3, 4, 5\}$, with $N = 32$, and UF $= 0.8$.

- Scenario 5: *Varying UF* - To study how different levels of resource utilization affect the system's performance, we vary the UF $\in \{0.1, 0.5, 0.8, 0.85, 0.9, 0.95$ $, 0.97, 0.99\}$, with $N = 32$, and $g = 4$. It is worth noting that, by considering the variation in UF, in general, we include the effect of changing the number of game session placement requests as well.

- Scenario 6: *Varying UF with Heterogeneous Resources* - We consider a scenario where the nodes' resources are randomly selected to be either in $\{1$ GHz$, 8$ GB$, 128$ GB$\}$ or in $\{5$ GHz$, 32$ GB$, 512$ GB$\}$ (i.e., CPU, memory, and storage respectively), simulating a more heterogeneous environment and reflecting variations in different resources across nodes. Similar to Scenario 5, we vary the UF $\in \{0.1, 0.5, 0.8, 0.85, 0.9, 0.95, 0.97, 0.99\}$, with $N = 32$, and $g = 4$.

- Scenario 7: *Multiple-resource Bottleneck* - We consider a scenario to assess how the algorithms behave in uncommon situations where more than one resource acts as a bottleneck. To create this scenario, we replicated Scenario 5 but introduced two resource bottlenecks by maintaining the same ratio between the two resource requirements of the GEMs (e.g., CPU and memory) and the corresponding available resources in the network. For each GEM, the CPU, memory, and storage requirements are uniformly distributed between 0 and 1 GHz, 0 and 6.4 GB, and 0 and 1 GB, respectively.

- Scenario 8: *Batch-placement in an online scenario* - We explore an online scenario considering a real system with game sessions arriving and leaving. We consider in total 10^5 requests for GEM to arrive following a Poisson process. Following the realistic cases, the game session associated with each GEM has a random duration in the interval $[60, 3600]$ s (e.g., the average game session length in PUBG [169]), after which the GEM releases the occupied resources. We assume that $N = 32$ and $g = 4$. The CPU is considered the sole bottleneck, creating a single-bottleneck resource scenario even with multiple resources, reflecting a realistic situation. We set the offered load to the network to be 0.5, resulting in average inter-arrival times for the GEMs of 11.25 s. We run the placement algorithm every Δt time, denoted as "batch-window", on all the GEMs that arrived during the last batch-window, considering the available resources. All the unplaced GEMs are dropped. We set $\Delta t \in \{0, 11.25, 25, 50, 100, 200, 400\}$ s, where the value 0 means that the placement algorithm runs GEM by GEM. The corresponding average batch sizes are shown in $\{0, 1, 2, 4.5, 9, 18, 36\}$ GEMs respectively. The different values for Δt reflect different tradeoffs between computation complexity and reactivity to changes in the workload. The number of players for each incoming GEM is uniformly selected from $\{1, 2, 4, 10, 50\}$ to showcase a real system in which different kinds of games exist. When each game session finishes, the corresponding resources are released. This scenario aims to show the adaptability of our approach to an online case.

We considered both 1-player GEMs and 2-player GEMs as 2 sub-scenarios for each of scenarios 1 to 6. These scenarios collectively provide a comprehensive system examination, considering various parameters and configurations. By exploring these different aspects, including the specific cases of 1-player and 2-player GEMs, we

can derive a robust understanding of the system's characteristics and performance under diverse conditions. For each test, we generated different random network topologies and for each topology, we generated multiple random sets S according to the mentioned parameters.

We measured two main metrics: the *acceptance probability* for the GEMs (as our main objective) and the *average normalized delay* (as the second objective), averaged across all the players and normalized based on the average RTT of each network topology[1]. Finally, we compared the complexity and execution time of all the algorithms. It is important to note that the described scenarios represent a selected subset of multiple test scenarios we considered to examine the algorithms' behavior in various situations. We excluded test cases that showed behavioral results similar to those already reported.

4.6.3 Alternative algorithms

For comparison, we considered the optimal algorithm, denoted as OPT, solving the problem (4.1)-(4.5). In addition, we considered the following alternative algorithms.

- *RaNDom (RND)*: It selects the GEMs in random order. For each selected GEM, it finds an eligible node in random order.
- *Quality-driven heuristic* (QDH*)*[2]: It selects the GEMs in random order. For each selected GEM, it finds an eligible node by searching the nodes with $d_{\text{tot}}(s,n)$ in increasing order. The algorithm is an extension for multiplayer to QDH proposed in [171], which was designed to place single-tenant containers.
- *First-Fit-Decreasing (FFD)*: It sorts the GEMs based on the CPU requirement in decreasing order. For each GEM, it finds an eligible node by searching

[1]This normalization not only offers us a dimensionless relative metric for each tested graph, simplifying the averaging among final measurements across different graphs and facilitating comparisons, but it also offers a straightforward explanation for the efficiency of the tested algorithms. It quantifies the relative reduction in average latency concerning the network topology utilized (e.g., algorithm A provides you 10ms in network X and 30ms in network Y as the average experienced latency for the players, versus algorithm A provides 50% of the average shortest path for each of networks A and B)

[2]Since in the multiplayer case we care about the average experienced latency by the players of each GEM, the extension to QDH is done by averaging (or even summing up) the total latency experienced by all of the players connecting to the same GEM

the nodes in random order. The algorithm belongs to the First-Fit-Decreasing (FFD) heuristics that have been shown to have given often good results for one-dimensional bin-packing problems [172].

- *MAP-RaNDom_First (MAP-RNDF)*: It is a two-phase algorithm, with the first phase identical to MAP in MAP-MIND. For the second phase, it selects the GEMs in random order, and for each selected GEM, it checks the nodes in random order to find an eligible node that reduces the overall delay if the GEM is moved there. Then it checks the other GEMs in random order to find the first one that is eligible to swap with and reduces the overall delay.

- *MAP-RaNDom_Greedy (MAP-RNDG)*: It is a two-phase algorithm similar to MAP-RNDF. But in the second phase, for each randomly selected GEM, it checks all the nodes for the move or swap steps (as in MAP-MIND) and chooses the one that minimizes the overall delay.

- *MAP-STeepest_Descent (MAP-STD)*: It is a two-phase algorithm extending the search options compared to MAP-RNDG. At each iteration, for each GEM, it considers all possible eligible nodes for the move and all possible GEMs to swap with and selects the option that minimizes the overall delay. It keeps iterating until no improvement on the overall delay is possible.

4.6.4 Simulation results

It is important to note that in all experiments, the dropping probability and delay graphs should be analyzed in conjunction, with a particular emphasis on the dropping probability as the key objective. Additionally, for all two-phase algorithms where MAP serves as the first phase (i.e., MAP-MIND, MAP-MIND*, MAP-RNDF, MAP-RNDG, and MAP-STD), the acceptance probability is the same. To maintain clarity in the probability graphs across all experimental scenarios and avoid redundancy, we represent all these algorithms solely with MAP.

Scenario 1: Varying the number of players per GEM

In Fig. 4.6 we report the acceptance ratio for the UDC setting, whereas Fig. 4.7 shows the average normalized delay for NDC and UDC settings, under scenario 1.

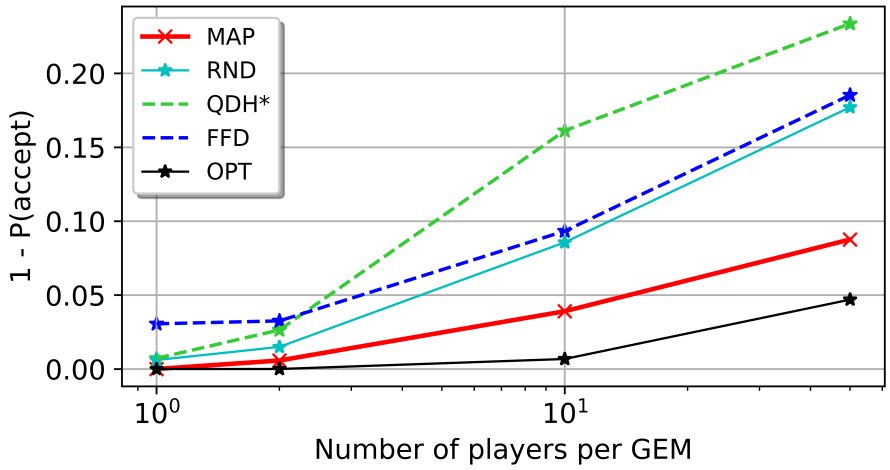


Fig. 4.6 Probability of dropping game session requests under scenario 1 for UDC.

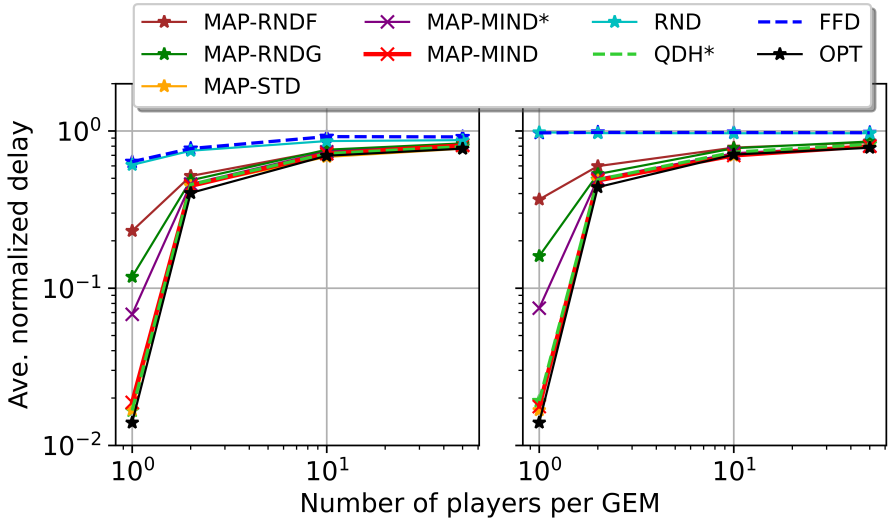


Fig. 4.7 Average normalized delay under scenario 1 for UDC (left) and NDC (right).

We have omitted the dropping probability for NDC since it consistently remains at zero.

In both UDC and NDC settings, each algorithm individually exhibits similar behavior. Generally, in the case of multi-player GEM with a high number of players, all algorithms tend to exhibit similar delays. This phenomenon occurs because, in high-player GEM scenarios, player distribution across different nodes forces GEM placement onto nodes with greater network centrality. This can result in higher dropping in less efficient algorithms, particularly in the case of UDC. However,

Table 4.2 Average execution time [s] under scenario 1 for UDC.

Algorithm	1-player	2-player	10-player	50-player
RND	0.010	0.012	0.035	0.143
FFD	0.016	0.019	0.048	0.179
QDH*	0.020	0.028	0.112	0.442
MAP	0.027	0.032	0.09	0.332
MAP-RNDF	0.813	0.830	1.05	2.05
MAP-RNDG	0.908	1.03	1.49	3.17
MAP-MIND*	0.873	0.969	1.54	3.71
MAP-MIND	1.37	1.96	6.30	15.6
MAP-STD	39.8	48.6	166	613
OPT	0.209	0.264	3.44	840 (i)

(i) Calculated only for the instances that were finished before 900 s wall-clock time (i.e., 88% of them).

certain algorithms opted to drop some GEMs, which subsequently led to lower delays in UDC, as we will elaborate on shortly. Lastly, as the number of players per GEM is reduced, higher efficiency algorithms demonstrate a greater reduction in the delay while maintaining a low dropping probability.

From the figures, as expected, all two-phase algorithms that use MAP as their first placement phase show the closest results for dropping probability to OPT. They have no dropping for 1-player GEMs, while they show a very small dropping probability for 2-player GEMs and a moderate increase in dropping probability up to 50-player GEMs. Among them, in terms of delay, MAP-MIND and MAP-STD show the closest delay to OPT while starting with MAP-MIND*, MAP-RNDG, and MAP-RNDF the experienced delay becomes worse. On the other hand, all the one-phase algorithms such as RND, QDH*, and FFD start dropping GEMs even for 1-player GEMs. For GEMs with a higher number of players, their dropping probability increases noticeably. RND shows the best probability of dropping among the three mentioned ones as it performs load balancing, which results in the highest delay as shown in the figures as well. QDH* achieves lower delay as it prioritizes reducing delay over dropping, so it drops more than RND while achieving near-optimal delay. As anticipated, FFD displays the worst delays since its primary goal is to enhance the acceptance probability. Interestingly, it presents the highest drop probability for 1-player and 2-player GEM scenarios. This is because it overlooks another crucial GEM placement constraint: the GEMs' maximum tolerable delay. However, for a

large number of players per GEM, specifically more than 10 players, the suitable nodes based on delay for different GEMs primarily narrow down to the network central nodes. Now, its dropping probability becomes less than QDH* algorithm and converges to the RND algorithm as it does a simple load balancing. Notably, all algorithms, except RND and FFD, converge to a similar value for the delay as the number of players per GEM increases.

The execution times for different algorithms under scenario 1 are reported in Table 4.2. When comparing the values for MAP with all the 2-phase algorithms, it is clear that the majority of the time is spent on improving the delay. Notably, MAP-STD, the heuristic closest to MAP-MIND, does not satisfy the typical time frames for setting up game sessions in real game environments. For most multiplayer games today, a wait time of around 10 s (i.e., up to 60 s before starting the game) to establish co-players or rivals in an arena is considered normal. Conversely, the time for MAP-MIND is calculated based on a worst-case scenario of around 256 games, each with 50 players, totaling more than $12,000$ players simultaneously. These players have diverse maximum tolerable delays, and the algorithm achieves near-optimal delay and less than 8% dropping. In practice, while MAP-MIND* already provides valuable execution time at the cost of a small increase in the offered delay, efficient algorithm implementation can significantly reduce execution time. By comparing the execution time of MAP and the execution time of the other two-phase algorithms based on MAP, it is clear that the MAP phase alone takes negligible time compared to the second phase of the other algorithms, which is devoted to the improvement of the placement solution in terms of delays through moves and swaps.

The time to compute the OPT results for both 1-player and 2-player GEM scenarios is typically less than 1 s. For the case of the 10-player GEM scenario, the average execution time is a few tens of seconds, with some instances solved around 40 s, which shows a high variability in the execution time. Interestingly, in the case of 50-player GEMs, the computation time varies significantly across instances, with some being solved in less than 10 s and some others taking over 900 s.

Scenario 2: Varying the number of CNs

In this experiment, we report results under scenario 2 and only for the UDC scheme. This is because, in the case of NDC, all algorithms' dropping probability is zero, and the delay results are similar to the UDC case. We report for the 1-player and

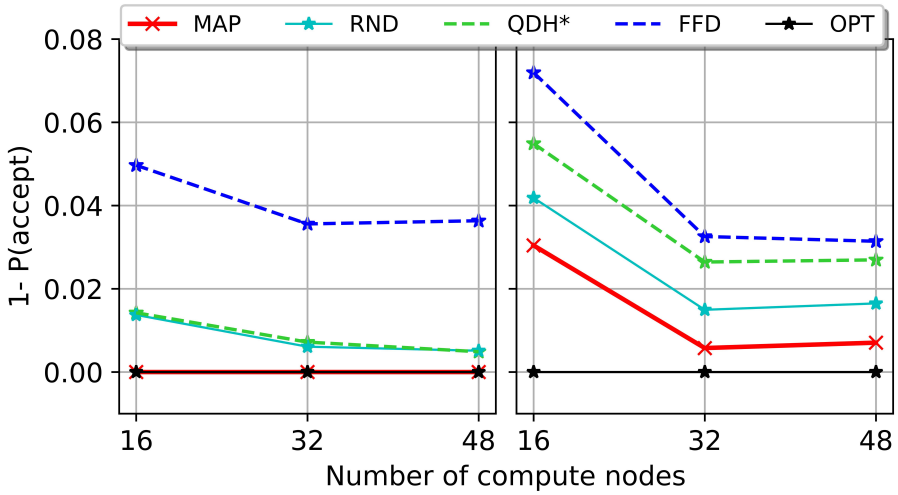


Fig. 4.8 Probability of dropping under scenario 2 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

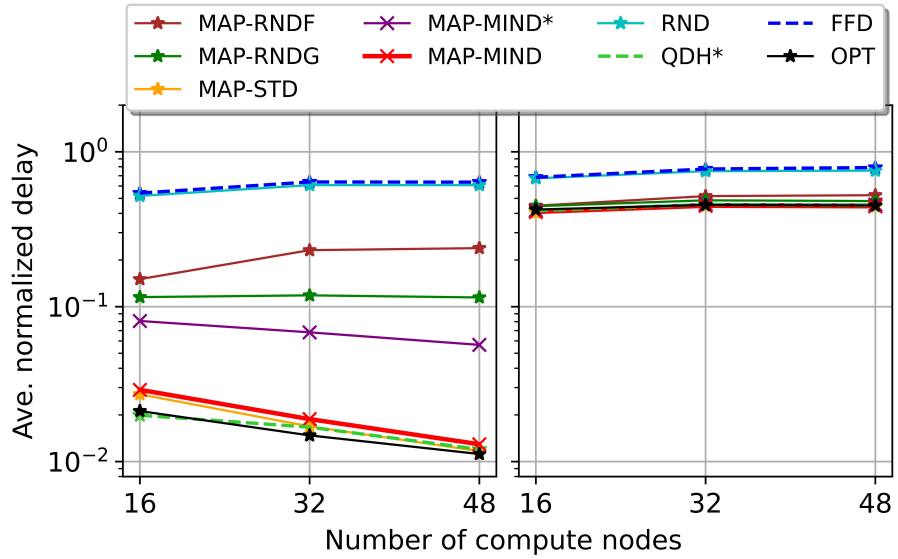


Fig. 4.9 Average normalized delay under scenario 2 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

2-player GEM cases. The dropping probability and the delay experienced by the GEMs for UDC are shown in Fig. 4.8 and Fig. 4.9 respectively.

The results regarding the experienced delay under scenario 2 are entirely consistent with the results of the 1-player and 2-player GEMs in the previous experiments under scenario 1. An interesting observation is that by increasing the number of CNs, the dropping probability of one-phase algorithms (i.e., FFD, QDH*, and RND),

Table 4.3 Average execution time [s] under scenario 2 for UDC.

Algorithm	$N = 16$		$N = 48$	
	1-player	2-player	1-player	2-player
RND	0.005	0.007	0.019	0.023
FFD	0.006	0.008	0.034	0.041
QDH*	0.009	0.013	0.040	0.059
MAP	0.010	0.013	0.055	0.073
MAP-RNDF	0.238	0.225	2.04	2.09
MAP-RNDG	0.246	0.274	2.46	2.78
MAP-MIND*	0.226	0.253	2.33	2.63
MAP-MIND	0.387	0.425	4.39	5.70
MAP-STD	4.41	4.94	158	206
OPT	0.084	0.098	0.489	0.653

decreases for both 1-player and 2-player GEMs. This is due to the increase in possible CNs for GEM placement. For 1-player GEMs, the delay for smarter algorithms reduces by increasing the CNs, but in 2-player GEMs, only the central CNs are eligible due to the possible positions of the 2 players of each GEM.

In summary, MAP-based algorithms experience the closest dropping probability to OPT, while one of our proposed algorithms, MAP-MIND, is also close to OPT in terms of experienced delay. We compared the execution times of the algorithms for this scenario as well. Note that, for this scenario, the average number of GEMs is 128, 256, and 384 respectively. We reported the results in Table 4.3.

Interestingly, MAP-STD, the heuristic closest to MAP-MIND, takes 10 to 30 times longer to achieve similar delay results. OPT needs, on average less than 1 s to obtain the results, regardless of the number of CNs. MAP-MIND* has a lower computation time than MAP-MIND. QDH* and MAP-STD have much larger computation times, but better delay results are achieved with higher GEM dropping. It is worth noting that, in a similar resource scaling test scenario, where we fixed the number of CNs and instead varied the resources of the CNs, similar results to the current scenario were obtained. These results have been omitted since redundant.

Scenario 3: Scaling CN resources

In this experiment, we report results under Scenario 3 and only for the UDC scheme. This is because, similar to scenario 2, in the case of NDC, all algorithms' dropping

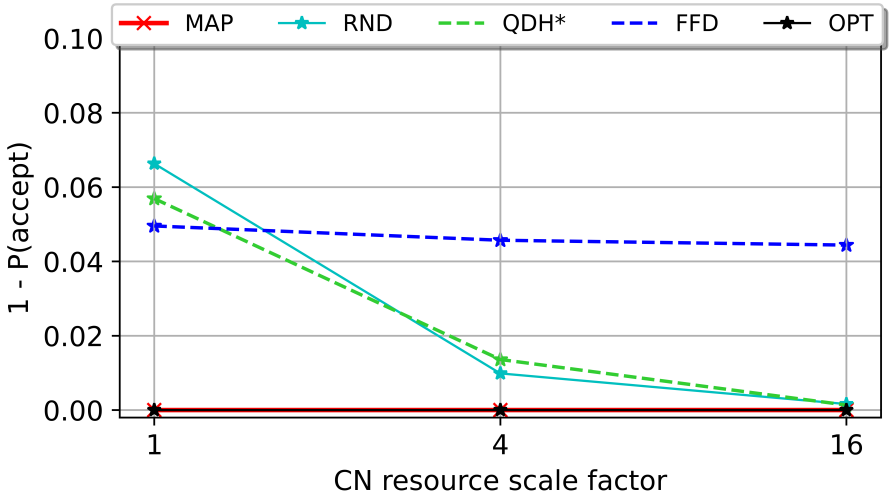


Fig. 4.10 Probability of dropping under Scenario 3 for UDC with1-player GEMs.

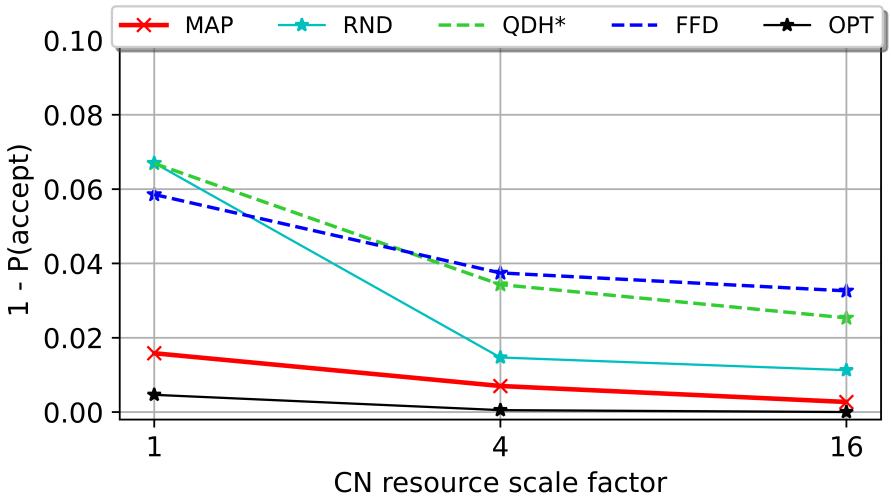


Fig. 4.11 Probability of dropping under Scenario 3 for UDC with 2-player GEMs.

probability is zero, and the delay results are similar to the UDC case. The results for 1-player GEMs are shown in Fig. 4.10 and Fig. 4.12 and the results for multiplayer GEMs are shown in Fig. 4.11 and Fig. 4.13 respectively.

As expected, for 1-player GEMs, our proposed algorithms keep the minimum dropping probability equal to zero (similar to optimal) among all competitors, regardless of the scale of the network. By increasing the resources, QDH* and RND rapidly approach our algorithms due to the increased difference between GEM sizes and resource sizes, reducing competition for the available resources. On

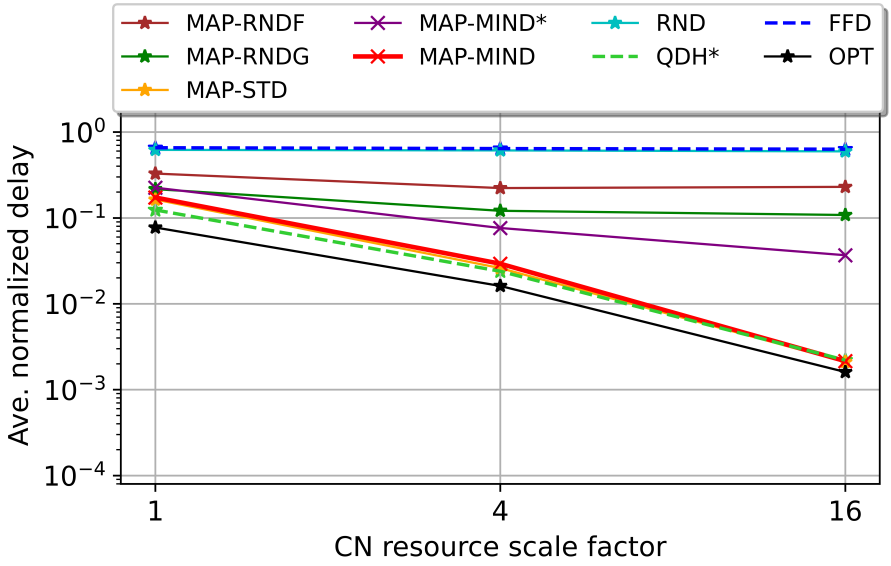


Fig. 4.12 Average normalized delay under Scenario 3 for UDC with 1-player GEMs.

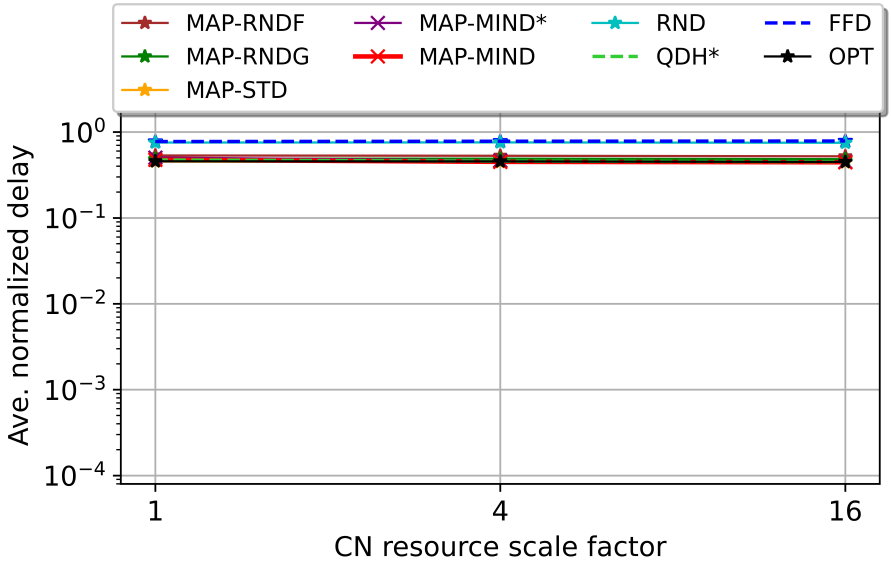


Fig. 4.13 Average normalized delay under Scenario 3 for UDC with 2-player GEMs.

the other hand, in terms of delay, MAP-MIND follows the same behavior, close to optimal and almost equal to QDH* while, as expected, MAP-MIND* trades off some delay with speed and yet outperforms other competitors except QDH*. Regarding multi-player GEMs, again, all MAP-based algorithms outperform RND, QDH*, and FFD in terms of dropping probability while achieving the closest results to the

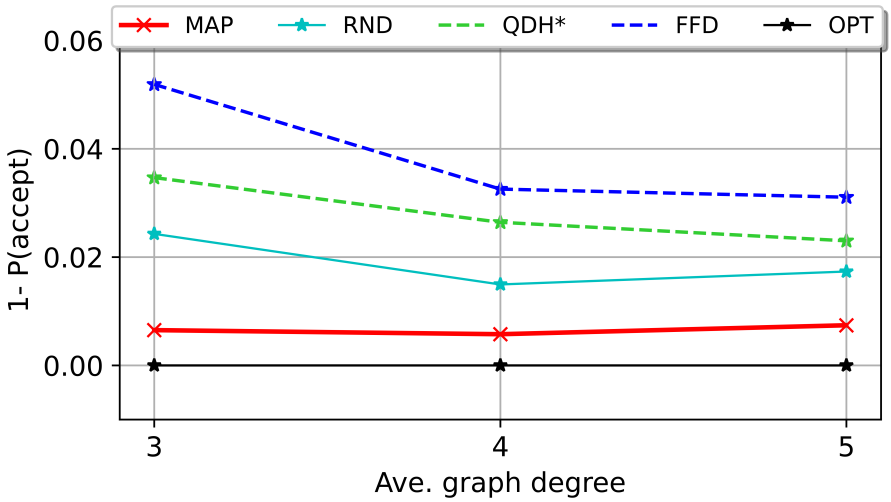


Fig. 4.14 Probability of dropping under Scenario 4 for UDC with 2-player GEMs.

optimal. In the case of the experienced delay, all algorithms demonstrate similar behavior, while RND and FDD achieve slightly worse delays.

Scenario 4: Varying average graph degree

In our experiments, we found that changing the average degree of the graph does not alter the average normalized delay results, and all previous observations regarding the behavior quality of the algorithms remain consistent. An interesting, yet expected, observation is that by increasing the average degree of the graph, the dropping probability of the less sophisticated algorithms slightly decreases for UDC with 2-player GEMs. This is because as the graph becomes more connected, the length of the shortest paths decreases, which is equivalent to reducing network delays. This leads to more delay-eligible CNs for less sophisticated algorithms at each GEM placement, thereby reducing their dropping probability. For NDC, no dropping occurs for all the algorithms. The dropping probability for UDC with 2-player GEMs is presented in Fig. 4.14.

Scenario 5: Varying the utilization factor

The dropping probability for UDC with both 1-player and 2-player GEMs is illustrated in Fig. 4.15. From the results of NDC, we report the dropping probability of both

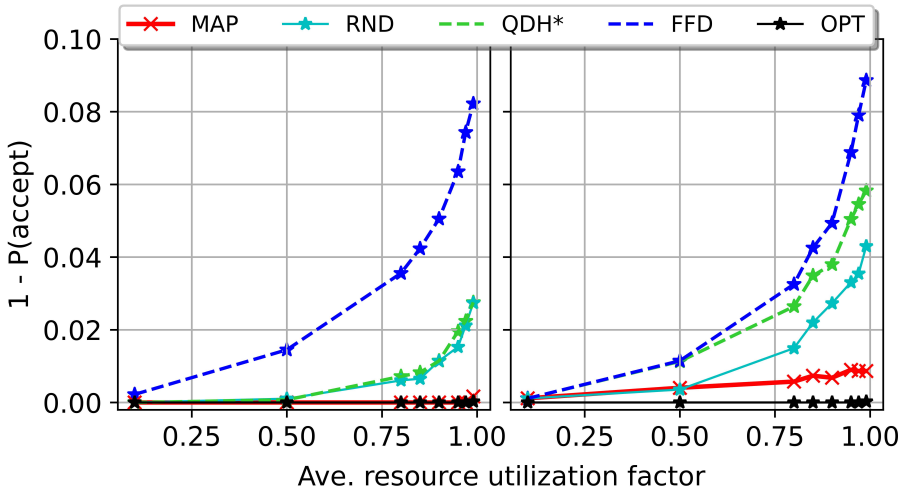


Fig. 4.15 Probability of dropping under Scenario 5 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

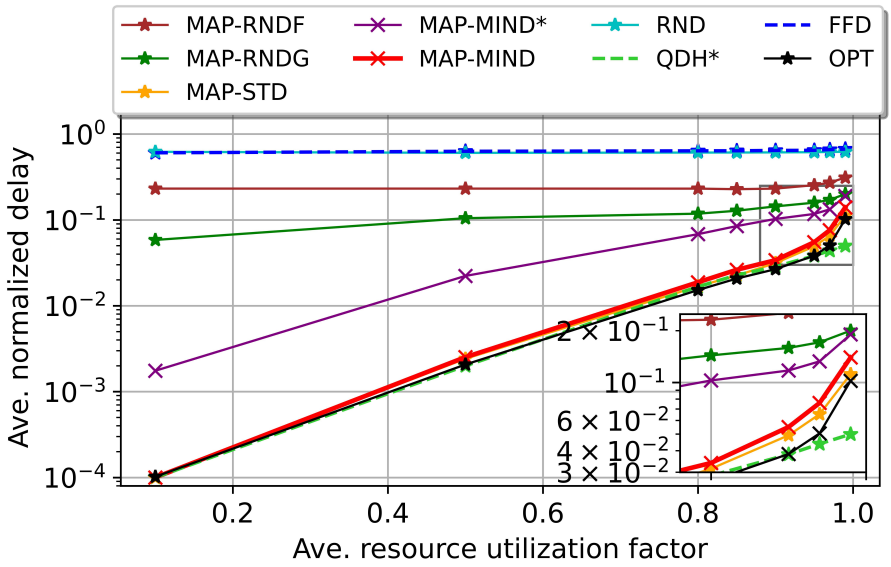


Fig. 4.16 Ave. normalized delay under Scenario 5 for UDC with 1-player GEMs.

1-player and 2-player GEMs as shown in Fig. 4.17. This is because it presents the only interesting behavior that differs from the aforementioned scenario. The delay results for UDC with 1-player GEMs are illustrated in Fig. 4.16. We do not report the delay results for UDC with 2-player GEMs, as it does not provide any interesting observations, being similar to the 2-player case in Fig. 4.9.

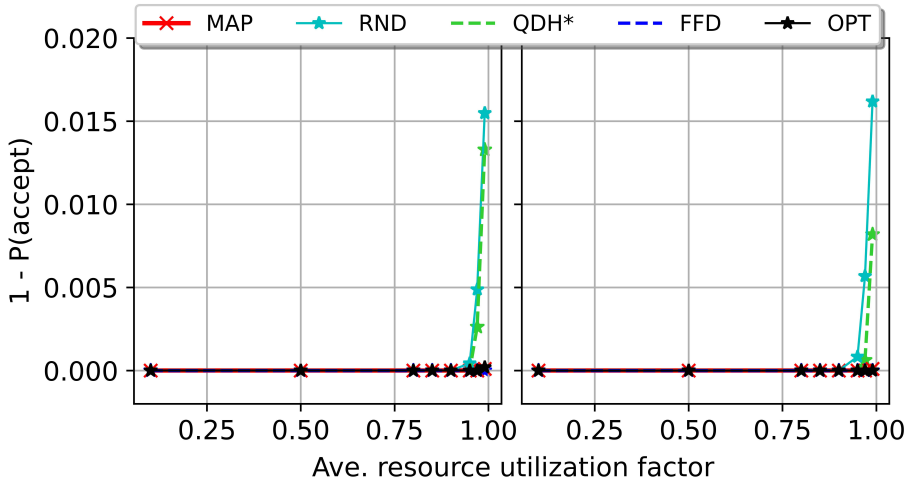


Fig. 4.17 Probability of dropping under Scenario 5 for NDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

For 1-player GEMs, considering Fig. 4.16 and Fig. 4.15 together, as expected, RND and FFD exhibit the worst delay since they are indifferent to the achieved delay. RND shows some minor dropping, especially at high loads. However, interestingly and contrary to what is expected, FFD displays a considerable dropping probability (i.e., the worst among all algorithms for UF ≥ 0.5). This is due to the algorithm ignoring the important role of maximum tolerable delay for GEMs while trying to maximize placement.

On the other hand, QDH* achieves a very good delay, almost approximating the OPT, which is gained by dropping some of the GEMs. All the MAP-based algorithms show almost zero dropping, even at the highest loads, while MAP-STD and MAP-MIND display the best delays, completely approximating the OPT. Notably, as discussed in Sec. 4.6.5, the computational complexity of MAP-MIND, translated into execution time, is much less than that of MAP-STD, making MAP-MIND the winner. Also, MAP-MIND* demonstrates a completely acceptable delay compared to other algorithms by considering its brilliant mix of execution time and dropping probability.

In the case of 2-player GEMs, other than RND and FFD, which achieve the worst delay, the remaining algorithms show almost identical delay results. The difference lies in the dropping probability, where we observe an increase for all algorithms, except for the OPT, compared to 1-player GEMs. This behavior for 2-player GEMs

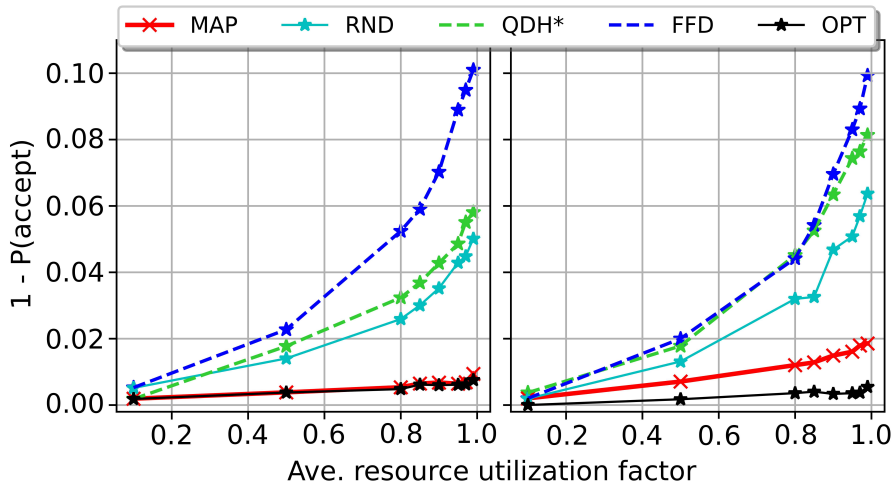


Fig. 4.18 Probability of dropping under Scenario 6 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

is expected, as in most cases, only the central CNs are eligible due to the potential positions of the two players of each GEM.

For NDC with 1-player GEMs, when the offered load exceeds 0.95 of the network resources, RND and QDH* start to drop the GEMs, even for NDC. This highlights the weakness of the algorithms that do not consider placement maximization in their logic. Now, compared to UDC cases, FFD can demonstrate its placement power by having no dropping, even at the highest load, as it does not consider the maximum tolerable delay as a constraint for GEM placement. MAP-based algorithms only show negligible dropping at the highest load.

Scenario 6: Varying UF with heterogeneous resources

The dropping probability for both 1-player and 2-player GEMs and the average normalized delay for 1-player GEMs are illustrated in Fig. 4.18 and Fig. 4.19, respectively.

When contrasted with a network featuring homogeneous resources, all algorithms with heterogeneous resources manifest qualitatively similar behavior regarding average normalized delay and dropping probability. As expected, in the presence of heterogeneous resources, we observe a small increase in both delays at low loads and dropping probabilities. The former observation can be attributed to the restricted

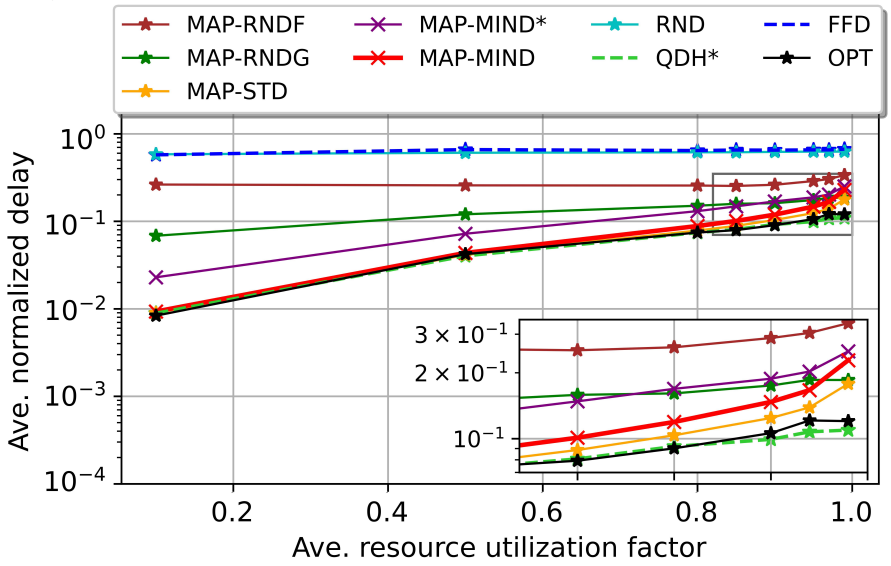


Fig. 4.19 Ave. normalized delay under Scenario 6 for UDC with 1-player GEMs.

resources on certain CNs, resulting in the placement of some GEMs on more distant CNs. The latter observation is a direct consequence of the former since distant nodes are more likely to exceed the maximum tolerable delay for the mentioned GEM, leading to its dropping.

Scenario 7: Multiple-resource bottleneck

The dropping probability for both 1-player and 2-player GEMs and the average normalized delay for 1-player GEMs are illustrated in Fig. 4.20 and Fig. 4.21, respectively. When comparing the results with Scenario 5 (i.e., single-resource bottleneck), no significant differences are observed except at very high loads (i.e., UF ≥ 0.95). At these high loads, all algorithms begin to drop the GEMs more rapidly. By concentrating solely on one bottleneck during placement, another bottleneck can emerge, leading to drops. In such cases, even the OPT struggles to accommodate all the GEMs within the network.

Scanrio 8: Batch-placement in an online scenario

The dropping probability and the average normalized delay for an online scenario are illustrated in Fig. 4.23 and Fig. 4.22.

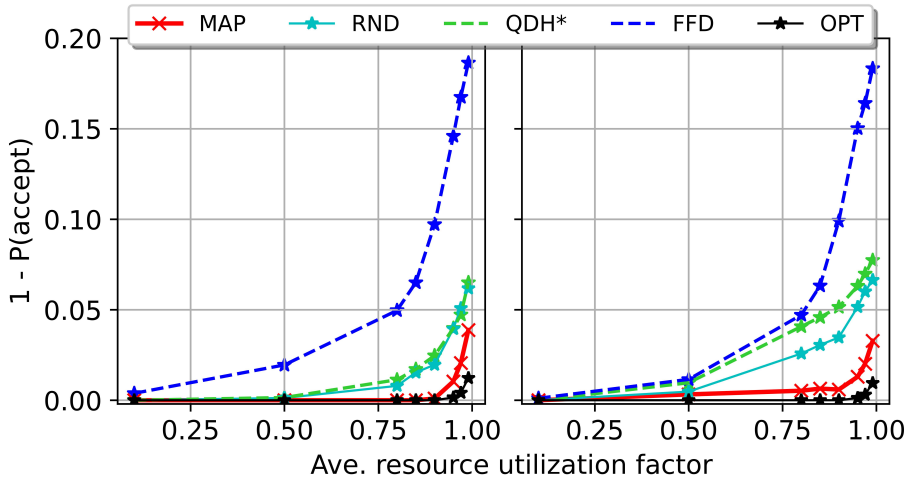


Fig. 4.20 Probability of dropping under Scenario 7 for UDC with: i) 1-player GEMs (left), ii) 2-player GEMs (right).

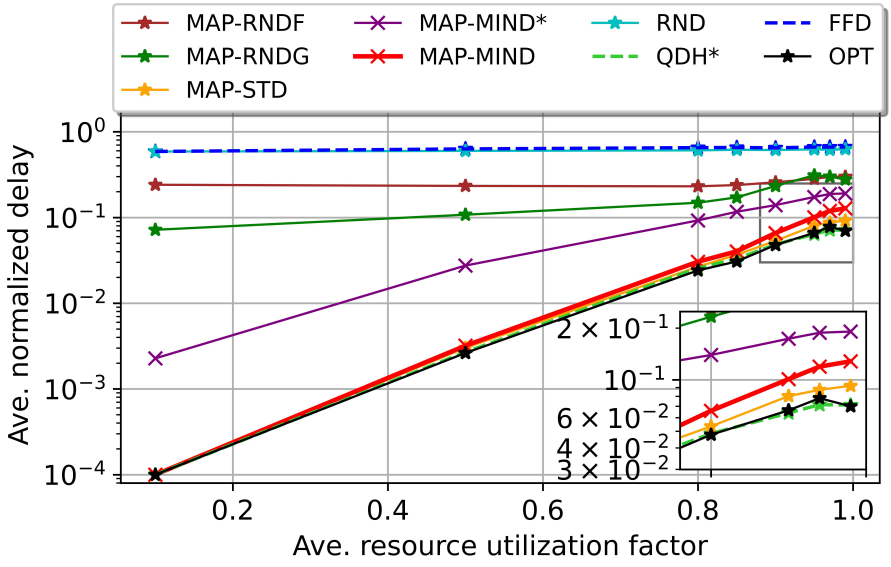


Fig. 4.21 Ave. normalized delay under Scenario 7 for UDC with 1-player GEMs.

In general, the decision to place the newly arrived GEMs at the end of each batch-window is based on a snapshot of GEMs, lacking foresight into upcoming GEMs in future windows. This myopic view can lead to sub-optimal global decisions over the entire simulation period. The effect should be explained by simultaneously considering two factors: (i) batch-window size Δt, and (ii) the efficiency of the algorithm. Here Δt has a paramount effect, as a larger size of collected GEMs

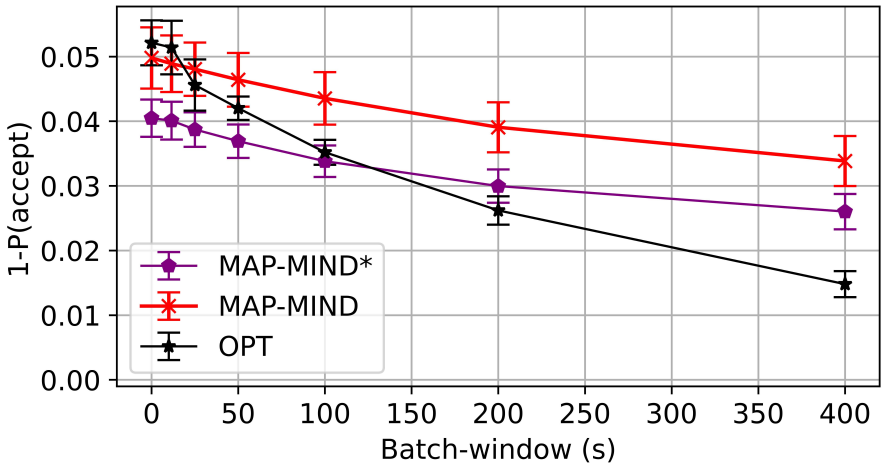


Fig. 4.22 Probability of dropping in UDC and under Scenario 8.

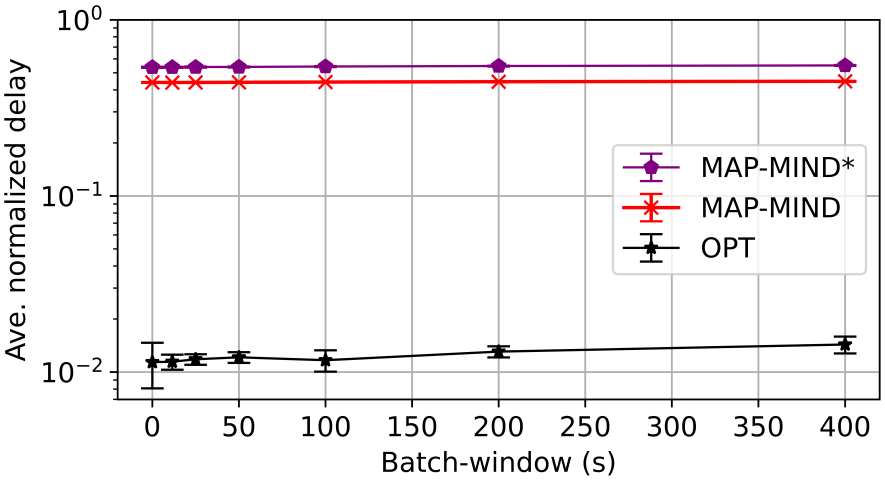


Fig. 4.23 Ave. normalized delay in UDC and under Scenario 8.

provides more information, enabling the proposed algorithms to make more informed decisions about GEM placement and lowering the probability of dropping.

From Fig.4.22, it is evident that increasing the value of Δt reduces the dropping probability for all algorithms. On the other hand, when algorithms make locally optimized decisions based on the current window's GEMs, they may allocate resources in a way that efficiently satisfies immediate GEMs but could potentially block critical resources needed for special or delay-critical GEMs in subsequent windows. Thus, more locally optimized algorithms are likely to achieve worse results globally for the smallest windows and better results for the largest windows.

This is coherent with the OPT results in Fig.4.22. Also, MAP-MIND demonstrates the same behavior, having worse results than MAP-MIND* for small windows. Considering the small structural difference between the two algorithms, investigating larger windows than shown in Fig. 4.22 is needed to show that the two algorithms eventually converge together and possibly change their positions as forecasted above.

In short, this behavior occurs because, with each decision, a locally optimized algorithm may exhaust the available resources on compute nodes, leaving insufficient capacity on possible eligible compute nodes for future requests. This myopic approach prioritizes immediate resource allocation, potentially neglecting the needs of upcoming requests. Consequently, this behavior can lead to sub-optimal global decisions over time, especially for algorithms making decisions solely based on current resources without considering future demands. So, simpler algorithms like MAP-MIND*, tend to do more load balancing leading to fewer request drops with the cost of increased experienced latency.

Regarding the delay, for all window sizes, OPT outperforms the algorithms as it aims to minimize placement delay, while MAP-MIND performs slightly better than MAP-MIND* due to slightly more dropped GEMs. It is worth reminding that, due to the nature of MAP as the first phase in our algorithms, to maximize placement, it trades delay for acceptance, leading to high acceptance at the cost of large but even acceptable delay (see Fig. 4.23).

On the other hand, introducing a collection window alters the incoming pattern, creating a burst of GEMs to place, which leads to raising the dropping probability. For instance, consider two distinct GEMs arriving at different times where only one specific server can serve them. If the computation of the first GEM concludes before the arrival of the second, one of them would need to be dropped if they coexist within the same window under batch placement. This scenario highlights the limitations of both window-based processing of collected GEMs and locally optimized decision-making. In the NDC case, where servers are interchangeable for GEMs, the effect of the window is limited to merely extending the average waiting time for GEMs to be served, which may result in dropping excess GEMs if Δt exceeds the maximum computation time. Conversely, in the UDC scenario, where servers are not interchangeable for GEMs (i.e., some servers are not suitable in terms of delay for some GEMs), a combination of the two aforementioned effects is observed.

Table 4.4 Computational complexity and average calculated iterations for different algorithms under the scenario 1.

Algorithm	Worst-case computational complexity $\mathcal{O}()$	Ave. iterations for k-player GEM with k equal to			
		1	2	10	50
RND	$\|S\|\|N\|$	908	928	$1.58k$	$2.37k$
QDH*	$\|S\|\|N\|^2 \log \|N\|$	350	780	$2.63k$	$3.57k$
FFD	$\|S\|(\log \|S\| + \|N\|)$	$4.32k$	$4.27k$	$4.37k$	$4.72k$
MAP-RNDF	$\|S\|^2$	$23.3k$	$14.5k$	$11.1k$	$10.5k$
MAP-RNDG	$\|S\|^2$	$74.2k$	$72.1k$	$68.5k$	$64.1k$
MAP-MIND*	$\|S\|^2$	$23.2k$	$29.8k$	$43.8k$	$48.2k$
MAP-MIND	$\|S\|^3$	$201k$	$247k$	$368k$	$388k$
MAP-STD	$\|S\|^3$	$15M$	$15M$	$16M$	$17M$

Interestingly, in the current landscape of online gaming, a wait time averaging more than 60 s to start a game is likely to be considered inconvenient by players. This suggests a window size not greater than 60 s, which, according to Fig. 4.22, showcases MAP-MIND* as a viable choice for online scenarios given its simplicity and efficiency.

4.6.5 Computational complexity of different algorithms

As a theoretical performance bound, we investigate and compare the computational complexity of our algorithms. The computational complexity of the process is primarily influenced by the number of GEMs ($|S|$) and CNs ($|N|$). As the worst-case scenario, we consider the NDC scenario to maximize the search space. In such a scenario, all nodes are eligible, resulting in the highest number of potential moves and swaps.

The first phase of all MAP-based algorithms, in the worst case, results in $|S||N|$ iterations. The moving phase of MAP-MIND iterates over all GEMs and CNs, resulting in $\mathcal{O}(|S||N|)$ iterations. In the worst case, this process is repeated $|S|$ times, leading to $\mathcal{O}(|S|^2|N|)$ total steps. The swapping phase iterates over all pairs of GEMs, resulting in $\mathcal{O}|S|^2$ iterations that, in the worst case, can be repeated $|S|$ times. This leads to $\mathcal{O}(|S||N| + |S|^2|N| + |S|^3)$ as the complexity of MAP-MIND. In contrast, in the delay-enhancing part, MAP-MIND* iterates over all GEMs while merging the move and swap in one phase, resulting in $\mathcal{O}|S|(|N| + |S|)$ steps in the worst

case. This leads to $\mathcal{O}(|S||N| + |S|^2)$ as the complexity of MAP-MIND*. It can be simply shown that the complexity of MAP-RNDF and MAP-RNDG is equal to that of MAP-MIND*. On the other hand, the MAP-STD moving phase iterates over all GEMs and CNs, resulting in $\mathcal{O}(|S||N|)$ iterations, and the swapping phase iterates over all pairs of GEMs, resulting in $\mathcal{O}(|S|^2)$ iterations. In the main iteration, the whole process can be repeated $|S|$ times in the worst case, as it can loop over all pairs of GEMs, which leads to a complexity of $\mathcal{O}(|S||N| + |S|^2|N| + |S|^3)$. Given that $|S| \gg |N|$, the worst-case computational complexity for all algorithms is reported in Table 4.4.

We reported the observed average number of iterations, moves, and swaps for all two-phased algorithms in Table 4.4. Upon comparing the two columns of Table 4.4, it is interesting to note that MAP-MIND*, despite having the same worst-case complexity as MAP-RNDF and MAP-RNDG, exhibits the minimum number of iterations and the best delay results among the three. However, while it slightly underperforms MAP-MIND in terms of delay, it is significantly more efficient and faster. Conversely, while MAP-MIND and MAP-STD have similar worst-case complexities and yield comparable delay performance, MAP-MIND proves to be substantially faster in practice compared to MAP-STD.

4.7 Discussion

The research presented in this work addresses the optimal placement of game engine modules (GEM) in cloud gaming scenarios. The proposed algorithms, MAP-MIND and MAP-MIND*, have been developed to effectively balance the maximization of the number of placed GEMs and the minimization of delay experienced by the players. We rigorously evaluated the performance of these algorithms through extensive simulations. MAP-MIND was shown to closely approximate the optimal solution, with a very small dropping probability in worst-case scenarios. Conversely, while the MAP-MIND* algorithm slightly under-perform in terms of delay compared to MAP-MIND, it offers significant advantages in terms of computation time, making it a practical alternative for real-world applications where large instances of the placement problem must be considered.

Chapter 5

Conclusion

During my research activity, we covered three different topics in the fields of programmable networks, blockchains, and cloud gaming with a unified focus on enhancing performance. At the core of these investigations is a unified focus on optimizing network-wide distributed applications through one or more of minimizing overall latency, reducing resource consumption, and alleviating network overhead.

In the first research activity, we introduced STARE, a design made for NFV and SDN-powered networks. STARE focuses on fast and efficient replicating states among multiple VNFs in 5G networks. To make this happen, it uses new data plane advances by shifting typical publish-subscribe broker functions to the programmable switches. At the same time, STARE offers a practically designed middleware that makes it easier for developers to interact with the VNFs. This is done by providing simple APIs that VNF source codes can easily access. we tested this approach on a realistic network testbed using P4-enabled switches and standard OpenFlow switches for vanilla SDN. we also compared our solution with other methods for sending published notifications using different technologies. Our tests show that using STARE with P4-enabled switches evenly spreads traffic across the network. This happens by using fixed-length headers that need fewer packets to deliver notifications to subscribers. Our approach has a slightly higher communication overhead compared to some alternatives, but it does not need interaction with a software broker or mapping the complete path between publishers and subscribers inside the packet. This completely separates publishers and subscribers. Also, our approach uses less memory than traditional methods relying on vanilla SDN.

Moreover, since STARE interacts less with a centralized controller and removes software brokers, it is expected to drastically reduce the state replication latency compared to traditional publish-subscribe systems.

In the second research activity, we aimed to reduce the overall latency on a widely used blockchain platform called Hyperledger Fabric (HF). Specifically, given an input *endorsement policy*, we studied the optimal choice to distribute the endorsement requests to the proper endorser peers (EPs). We proposed the "OPEN" algorithm, devised to minimize the latency due to both network delays and the processing times at the EPs. We provided a theoretical model, based on queuing theory, to compute the optimal number of EPs and to tune the parameters of "OPEN". By extensive simulations, we showed that "OPEN" can reduce the *endorsement latency* up to 70% compared to the state-of-the-art solution and approximated well the introduced optimal policies while offering a negligible implementation overhead compared to them. Additionally, we implemented the "OPEN" algorithm in Hyperledger Fabric to test its performance in a practical setting where the initial results from the test version of "OPEN" were promising. Yet, there is more work needed to perfect the design of the client-based "OPEN" solution and to test it extensively. Considering the rapid evolution of the HF, especially introducing the so-called "Gateway module", deciding on the proper place to implement OPEN is a future research opportunity.

In the latest research activity, we tackled optimizing game engine module (GEM) placement in cloud gaming setups. we introduced two new algorithms called MAP-MIND and MAP-MIND* to maximize the acceptance of the GEMs while reducing the average latency observed by the players. we ran extensive simulations to check these algorithms' performance. MAP-MIND almost perfectly approximates the optimal solution, even in the worst heterogeneous scenarios. On the other side, MAP-MIND* has slightly more latency but is highly efficient computationally, making it practical for real-world use, especially in scenarios with lots of placement instances. Our research aimed to improve these algorithms further by exploring optimizations and considering the ever-changing cloud gaming technology landscape.

In summary, drawing from extensive research across programmable networks, network-wide distributed blockchain technologies, and cloud gaming, this dissertation has successfully tackled the shared challenge of enhancing performance in distributed applications. By focusing on minimizing latency, reducing resource consumption, and alleviating network overhead, significant improvements have been achieved.

Through the introduction of novel fast and efficient mechanisms for state replication in 5G networks, optimization of blockchain processes for latency-sensitive applications, and the development of simultaneously fast and efficient algorithms for game engine module placement in distributed gaming environments, tangible progress has been made. These advancements not only address specific obstacles within each domain but also contribute to broader strategies for enhancing performance in network-wide distributed applications. As a result, this research lays a strong foundation for future endeavors aimed at optimizing distributed systems for improved efficiency and effectiveness.

References

[1] An Introduction to SDN. https://qmonnet.github.io/whirl-offload/2016/07/08/introduction-to-sdn/. Accessed: 2023-11.

[2] What is Network Function virtualization? https://www.juniper.net/it/it/research-topics/what-is-network-functions-virtualization-nfv.html. Accessed: 2023-11.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] ETSI. https://www.etsi.org/. Accessed: 2023-11.

[5] OpenDaylight. https://www.opendaylight.org/. Accessed: 2023-11.

[6] ONOS. https://opennetworking.org/onos/. Accessed: 2023-11.

[7] P4 Workflow. https://p4.org/. Accessed: 2023-11.

[8] P4 language evolution. https://opennetworking.org/news-and-events/blog/p4-language-evolution/. Accessed: 2023-11.

[9] Open Networking Foundation: P4. https://opennetworking.org/p4/. Accessed: 2023-11.

[10] Laurie Hughes, Yogesh K. Dwivedi, Santosh K. Misra, Nripendra P. Rana, Vishnupriya Raghavan, and Viswanadh Akella. Blockchain research, practice and policy: Applications, benefits, limitations, emerging research themes and research agenda. *International Journal of Information Management*, 49, 2019.

[11] Samuel Fosso Wamba and Maciel M. Queiroz. Blockchain in the operations and supply chain management: Benefits, challenges and future research opportunities. *International Journal of Information Management*, 52, 2020.

[12] Israa Abu-elezz, Asma Hassan, Anjanarani Nazeemudeen, Mowafa Househ, and Alaa Abd-alrazaq. The benefits and threats of blockchain technology in healthcare: A scoping review. *International Journal of Medical Informatics*, 2020.

[13] Shikah J. Alsunaidi and Fahd A. Alhaidari. A survey of consensus algorithms for blockchain technology. In *2019 International Conference on Computer and Information Sciences (ICCIS)*, 2019.

[14] Siamak Solat, P. Calvez, and Farid Naït-Abdesselam. Permissioned vs. permissionless blockchain: How and why there is only one right choice. *Journal of Software*, 2020.

[15] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 2008.

[16] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *ACM EuroSys*, 2018.

[17] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: An introduction, 2016.

[18] Hyperledger fabric: An introduction. =https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html. Accessed: 2023-11.

[19] Google Stadia. https://stadia.google.com. Accessed: 2023-11.

[20] Amazon Luna. https://www.amazon.com/luna. Accessed: 2023-11.

[21] GeForce Cloud Gaming. https://www.nvidia.com/geforce-now. Accessed: 2023-11.

[22] Xbox Cloud Gaming. https://www.xbox.com/en-US/cloud-gaming. Accessed: 2023-11.

[23] Luigi De Giovanni, Davide Gadia, Paolo Giaccone, Dario Maggiorini, Claudio E Palazzi, Laura Anna Ripamonti, and German Sviridov. Revamping cloud gaming with distributed engines. *IEEE Internet Computing*, 26(6), 2022.

[24] Iman Lotfimahyari, German Sviridov, Paolo Giaccone, and Andrea Bianco. Data-plane-assisted state replication with network function virtualization. *IEEE Systems Journal*, 16(2), 2022.

[25] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987.

[26] Ken Birman. The promise, and limitations, of gossip protocols. *ACM SIGOPS Operating Systems Review*, 41(5), 2007.

[27] Abdelsalam A Helal, Abdelsalam A Heddaya, and Bharat B Bhargava. *Replication techniques in distributed systems*, volume 4. Springer Science & Business Media, 2005.

[28] Eric Brewer. CAP twelve years later: How the rules have changed. *IEEE Computer Magazine*, 2012.

[29] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1), 2009.

[30] Manuel Peuster and Holger Karl. E-state: Distributed state management in elastic network function deployments. In *IEEE NetSoft*, 2016.

[31] Manuel Peuster, Hannes Küttner, and Holger Karl. A flow handover protocol to support state migration in softwarized networks. In *Wiley Online Library*, 2019.

[32] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. In *SIGCOMM Comput. Commun. Rev.*, volume 44, 2014.

[33] Marco Bonola, Roberto Bifulco, Luca Petrucci, Salvatore Pontarelli, Angelo Tulumello, and Giuseppe Bianchi. Implementing advanced network functions for datacenters with stateful programmable data planes. In *IEEE LANMAN*, 2017.

[34] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[35] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *ACM SIGCOMM*, 2015.

[36] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *USENIX OSDI*, 2006.

[37] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable Consistency in Scatter. In *ACM SOSP*, 2011.

[38] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), 2013.

[39] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing State: consistent updates for stateful and programmable data planes. In *ACM SOSR*, 2017.

[40] German Sviridov, Marco Bonola, Angelo Tulumello, Paolo Giaccone, Andrea Bianco, and Giuseppe Bianchi. LOcAl DEcisions on Replicated States (LOADER) in programmable dataplanes: Programming abstraction and experimental evaluation. *Computer Networks (Elsevier)*, 2020.

[41] Misha Hungyo and Mayank Pandey. SDN based implementation of publish/subscribe paradigm using OpenFlow multicast. In *IEEE ANTS*, 2016.

[42] Christian Wernecke, Helge Parzyjegla, Gero Mühl, Peter Danielis, and Dirk Timmermann. Realizing content-based publish/subscribe with P4. In *IEEE NFV-SDN*, 2018.

[43] Toyokazu Akiyama, Yuuichi Teranishi, Ryohei Banno, Katsuyoshi Iida, and Yukiko Kawai. Scalable pub/sub system using openflow control. *Journal of Information Processing*, 2016.

[44] Abubakar Siddique Muqaddas, German Sviridov, Paolo Giaccone, and Andrea Bianco. Optimal state replication in stateful data planes. *IEEE Journal on Selected Areas in Communications*, 2020.

[45] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, 1987.

[46] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Comput. Surv.*, 51(6), 2019.

[47] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2), 2003.

[48] Shehnaaz Yusuf. Survey of publish subscribe communication system. *Adv. Internet Appl. Syst*, 2004.

[49] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. Flowblaze: Stateful packet processing in hardware. In *NSDI*, 2019.

[50] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, 2016.

[51] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[52] Fabrizio Moggio, Mauro Boldi, Silvia Canale, Vincenzo Suraci, Claudio Casetti, Giacomo Bernini, Giada Landi, and Paolo Giaccone. 5G EVE a European platform for 5G application deployment. In *ACM WinTECH*, 2020.

[53] Kalkidan Gebru, Claudio Casetti, Carla Fabiana Chiasserini, and Paolo Giaccone. IoT-based mobility tracking for smart city applications. In *EuCNC*, 2020.

[54] State Sharing with P4 (STARE). https://github.com/imanlotfimahyari/State-Sharing-p4-python/blob/master/pubsub/pubsub_register/pub_sub.p4.

[55] P4 Runtime. https://github.com/p4lang/PI. Accessed: 2023-11.

[56] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in SDN-enabled switches. In *ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015.

[57] Frederik Hauser, Mark Schmidt, Marco Häberle, and Michael Menth. P4-macsec: Dynamic topology monitoring and data layer protection with macsec in p4-based sdn. *IEEE Access*, 8, 2020.

[58] Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/. Accessed: 2023-11.

[59] BMV2 Behavioral Model Version 2. https://github.com/p4lang/behavioral-model. Accessed: 2023-11.

[60] Build SDN agilely. https://ryu-sdn.org/. Accessed: 2023-11.

[61] OVSK Open Virtual Switch. https://www.openvswitch.org/. Accessed: 2023-11.

[62] Iman Lotfimahyari and Paolo Giaccone. Optimal endorsement for network-wide distributed blockchains. *IEEE Systems Journal*, 17(3), 2023.

[63] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014), 2014.

[64] Jean C Digitale, Jeffrey N Martin, and Medellena Maria Glymour. Tutorial on directed acyclic graphs. *Journal of Clinical Epidemiology*, 142, 2022.

[65] Wellington Fernandes Silvano and Roderval Marcelino. Iota tangle: A cryptocurrency to communicate internet-of-things data. *Future generation computer systems*, 112, 2020.

[66] Hyperledger Fabric documents: A new approach. https://hyperledger-fabric.readthedocs.io/en/release-2.5/whatis.html#:~:text=Fabric%20introduces%20a%20new%20architecture%20for%20transactions%20that%20we%20call%20execute%2Dorder%2Dvalidate.%20It%20addresses%20the%20resiliency%2C%20flexibility%2C%20scalability%2C%

20performance%20and%20confidentiality%20challenges%20faced%20by%20the%20order%2Dexecute%20model%20by%20separating%20the%20transaction%20flow%20into%20three%20steps%3A. Accessed: 2023-11.

[67] Hyperledger Fabric documents: Endorsement. https://hyperledger-fabric.readthedocs.io/en/release-2.2/glossary.html#endorsement. Accessed: 2023-11.

[68] Xiaoqiong Xu, Gang Sun, Long Luo, Huilong Cao, Hongfang Yu, and Athanasios V Vasilakos. Latency performance modeling and analysis for Hyperledger Fabric blockchain network. *Information Processing & Management*, 2021.

[69] Harish Sukhwani, Nan Wang, Kishor S Trivedi, and Andy Rindos. Performance modeling of Hyperledger Fabric (permissioned blockchain network). In *IEEE NCA*, 2018.

[70] Pu Yuan, Kan Zheng, Xiong Xiong, Kuan Zhang, and Lei Lei. Performance modeling and analysis of a Hyperledger-based system using GSPN. *Computer Communications, Elsevier*, 2020.

[71] Lei Hang and Do-Hyeun Kim. Optimal blockchain network construction methodology based on analysis of configurable components for enhancing Hyperledger Fabric performance. *Blockchain: Research and Applications*, 2021.

[72] Santiago Figueroa-Lorenzo, Javier Añorga, and Saioa Arrizabalaga. Methodological performance analysis applied to a novel IIoT access control system based on permissioned blockchain. *Information Processing & Management*, 2021.

[73] Chao Liu, Mingxuan Li, Yazhe Wang, Yu Wang, Dongdong Huo, and Ya Chen. Achieve better endorsement balance on blockchain systems. In *CSCWD*. IEEE, 2021.

[74] Jeeta Ann Chacko, Ruben Mayer, and Hans-Arno Jacobsen. Why do my blockchain transactions fail? a study of hyperledger fabric. In *ACM SIGMOD*, 2021.

[75] Feng Lu, Lu Gan, Zhongli Dong, Wei Li, Hai Jin, and Albert Y Zomaya. A cache enhanced endorser design for mitigating performance degradation in Hyperledger Fabric. In *IEEE International Conference on Blockchain*, 2018.

[76] Minsu Kwon and Heonchang Yu. Performance improvement of ordering and endorsement phase in Hyperledger Fabric. In *IOTSMS*. IEEE, 2019.

[77] Nihar B Shah, Kangwook Lee, and Kannan Ramchandran. When do redundant requests reduce latency? *IEEE Trans. on Communications*, 2015.

[78] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. More is less: Reducing latency via redundancy. In *ACM HotNets*, 2012.

[79] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *ACM CoNEXT*, 2013.

[80] Kristen Gardner, Samuel Zbarsky, Sherwin Doroudi, Mor Harchol-Balter, and Esa Hyytia. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Perf. Evaluation Review*, 2015.

[81] Kristen Gardner, Mor Harchol-Balter, Alan Scheller-Wolf, and Benny Van Houdt. A better model for job redundancy: Decoupling server slowdown and job size. *IEEE/ACM Transactions on Networking*, 25(6), 2017.

[82] Kristen Gardner and Samuel Zbarsky. Analyzing response time in the redundancy-d system. *ACM SIGMETRICS, Perf. Evaluation Review*, 2015.

[83] Joseph Hollinghurst, Ayalvadi Ganesh, and Timothy Baugé. Latency reduction in communication networks using redundant messages. In *ITC*, 2017.

[84] Gauri Joshi, Emina Soljanin, and Gregory Wornell. Efficient replication of queued tasks for latency reduction in cloud systems. In *Allerton Conference on Communication, Control, and Computing*, 2015.

[85] Endorsement policies. https://hyperledger-fabric.readthedocs.io/en/release-2.2/endorsement-policies.html.

[86] Peter Purdue. The M/M/1 queue in a Markovian Environment. *Operations research*, 22(3):562–569, 1974.

[87] Herbert A David and Haikady N Nagaraja. *Order statistics*. John Wiley & Sons, 2004.

[88] OMNeT++. https://omnetpp.org/.

[89] McSeth Antwi, Asma Adnane, Farhan Ahmad, Rasheed Hussain, Muhammad Habib ur Rehman, and Chaker Abdelaziz Kerrache. The case of hyperledger fabric as a blockchain solution for healthcare applications. *Blockchain: Research and Applications*, 2(1), 2021.

[90] Botao Zhong, Haitao Wu, Lieyun Ding, Hanbin Luo, Ying Luo, and Xing Pan. Hyperledger fabric-based consortium blockchain for construction quality information management. *Frontiers of engineering management*, 7(4), 2020.

[91] Charalampos Stamatellis, Pavlos Papadopoulos, Nikolaos Pitropakis, Sokratis Katsikas, and William J Buchanan. A privacy-preserving healthcare framework using hyperledger fabric. *Sensors*, 20(22), 2020.

[92] Mueen Uddin. Blockchain medledger: Hyperledger fabric enabled drug traceability system for counterfeit drugs in pharmaceutical industry. *International Journal of Pharmaceutics*, 597, 2021.

[93] The Internet Topology Zoo. http://www.topology-zoo.org. Accessed: 2023-11.

[94] Dario Maggiorini, Laura Anna Ripamonti, Eraldo Zanon, Armir Bujari, and Claudio Enrico Palazzi. SMASH: A distributed game engine achitecture. In *IEEE ISCC*, 2016.

[95] Sung-Soo Kim, Kyoung-Ill Kim, and Jongho Won. Multi-view rendering approach for cloud-based gaming services. In *Proceedings of the 3rd International Conference on Advances in Future Internet*, page 102107, 2011.

[96] Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(2):1–25, 2014.

[97] Chao Zhang, Jianguo Yao, Zhengwei Qi, Miao Yu, and Haibing Guan. vgasa: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):3036–3045, 2013.

[98] Mehdi Semsarzadeh, Mahdi Hemmati, Abbas Javadtalab, Abdulsalam Yassine, and Shervin Shirmohammadi. A video encoding speed-up architecture for cloud gaming. In *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, pages 1–6. IEEE, 2014.

[99] Mickael Hassam, Nadjia Kara, Fatna Belqasmi, and Roch Glitho. Virtualized infrastructure for video game applications in cloud environments. In *Proceedings of the 12th ACM international symposium on Mobility management and wireless access*, pages 109–114, 2014.

[100] Richard Süselbeck, Gregor Schiele, and Christian Becker. Peer-to-peer support for low-latency massively multiplayer online games in the cloud. In *2009 8th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–2. IEEE, 2009.

[101] S Prabu and Swarnalatha Purushotham. Cloud gaming with p2p network using xaml and windows azure. In *International Conference on Computing and Communication Systems*, pages 165–172. Springer, 2011.

[102] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6. IEEE, 2012.

[103] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. A hybrid edge-cloud architecture for reducing on-demand gaming latency. *Multimedia systems*, 20:503–519, 2014.

[104] Teemu Kämäräinen, Matti Siekkinen, Yu Xiao, and Antti Ylä-Jääski. Towards pervasive and mobile gaming with distributed cloud infrastructure. In *2014 13th Annual Workshop on Network and Systems Support for Games*, pages 1–6. IEEE, 2014.

[105] Hao Tian, Di Wu, Jian He, Yuedong Xu, and Min Chen. On achieving cost-effective adaptive cloud gaming in geo-distributed data centers. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12):2064–2077, 2015.

[106] Moreno Marzolla, Stefano Ferretti, and Gabriele D'angelo. Dynamic resource provisioning for cloud-based gaming infrastructures. *Computers in Entertainment (CIE)*, 10(1):1–20, 2012.

[107] Yusen Li, Xueyan Tang, and Wentong Cai. Play request dispatching for efficient virtual machine usage in cloud gaming. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12):2052–2063, 2015.

[108] Wei Cai, Victor CM Leung, and Long Hu. A cloudlet-assisted multiplayer cloud gaming system. *Mobile Networks and Applications*, 19, 2014.

[109] Wei Cai, Zhen Hong, Xiaofei Wang, Henry CB Chan, and Victor CM Leung. Quality-of-experience optimization for a cloud gaming system with ad hoc cloudlet assistance. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12), 2015.

[110] Kairan Sun and Dapeng Wu. Video rate control strategies for cloud gaming. *Journal of Visual Communication and Image Representation*, 30, 2015.

[111] Yao Liu, Sujit Dey, and Yao Lu. Enhancing video encoding for cloud gaming using rendering information. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12), 2015.

[112] Mehdi Semsarzadeh, Abdulsalam Yassine, and Shervin Shirmohammadi. Video encoding acceleration in cloud gaming. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12), 2015.

[113] Li Lin, Xiaofei Liao, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. Liverender: A cloud gaming system based on compressed graphics streaming. In *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[114] Dominik Meiländer, Frank Glinka, Sergei Gorlatch, Li Lin, Wei Zhang, and Xiaofei Liao. Bringing mobile online games to clouds. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2014.

[115] Seong-Ping Chuah and Ngai-Man Cheung. Layered coding for mobile cloud gaming. In *Proceedings of International Workshop on Massively Multiuser Virtual Environments*, 2014.

[116] Seong-Ping Chuah, Ngai-Man Cheung, and Chau Yuen. Layered coding for mobile cloud gaming using scalable blinn-phong lighting. *IEEE Transactions on Image Processing*, 25(7), 2016.

[117] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proceedings of the 19th ACM international conference on Multimedia*, 2011.

[118] Lingfeng Xu, Xun Guo, Yan Lu, Shipeng Li, Oscar C Au, and Lu Fang. A low latency cloud gaming system using edge preserved image homography. In *2014 IEEE International Conference on Multimedia and Expo (ICME)*, 2014.

[119] Sari Jarvinen, J-P Laulajainen, Tiia Sutinen, and Sami Sallinen. Qos-aware real-time video encoding how to improve the user experience of a gaming-on-demand service. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006.*, volume 2, 2006.

[120] J Laulajainen, Tiia Sutinen, and Sari Jarvinen. Experiments with qos-aware gaming-on-demand service. In *20th International Conference on Advanced Information Networking and Applications-Volume 1 (AINA'06)*, volume 1, 2006.

[121] Shaoxuan Wang and Sujit Dey. Addressing response time and video quality in remote server-based internet mobile gaming. In *2010 IEEE wireless communication and networking conference*, 2010.

[122] Shaoxuan Wang and Sujit Dey. Rendering adaptation to address communication and computation constraints in cloud mobile gaming. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, 2010.

[123] Jiyan Wu, Chau Yuen, Ngai-Man Cheung, Junliang Chen, and Chang Wen Chen. Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(12), 2015.

[124] Mahdi Hemmati, Abbas Javadtalab, Ali Asghar Nazari Shirehjini, Shervin Shirmohammadi, and Tarik Arici. Game as video: Bit rate reduction through adaptive object encoding. In *Proceeding of the 23rd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2013.

[125] Asad Ali, Tushin Mallick, Sadman Sakib, Md. Shohrab Hossain, and Ying-Dar Lin. Provisioning fog services to 3gpp subscribers: Authentication and application mobility. In *ICC 2022*, 2022.

[126] Takehiro Sato and Eiji Oki. Program file placement strategies for machine-to-machine service network platform in dynamic scenario. *IEICE Transactions on Communications*, 2020.

[127] Binayak Kar, Kuan-Min Shieh, Yuan-Cheng Lai, Ying-Dar Lin, and Huei-Wen Ferng. QoS violation probability minimization in federating vehicular-fogs with cloud and edge systems. *IEEE Transactions on Vehicular Technology*, 70(12), 2021.

[128] Bernardetta Addis, Giuliana Carello, and Meihui Gao. ILP-based heuristics for a virtual network function placement and routing problem. *Networks*, 78(3), 2021.

[129] Carlos Ruiz De Mendoza, Bahador Bakhshi, Engin Zeydan, and Josep Mangues-Bafalluy. Near optimal VNF placement in edge-enabled 6G networks. In *IEEE ICIN*, 2022.

[130] Penghao Sun, Julong Lan, Junfei Li, Zehua Guo, and Yuxiang Hu. Combining deep reinforcement learning with graph neural networks for optimal VNF placement. *IEEE Communications Letters*, 25(1), Jan 2021.

[131] Anestis Dalgkitsis, Prodromos-Vasileios Mekikis, Angelos Antonopoulos, Georgios Kormentzas, and Christos Verikoukis. Dynamic resource aware VNF placement with deep reinforcement learning for 5G networks. In *IEEE GLOBECOM*, 2020.

[132] Morteza Golkarifard, Carla Fabiana Chiasserini, Francesco Malandrino, and Ali Movaghar. Dynamic VNF placement, resource allocation and traffic routing in 5G. *Computer Networks*, 188, 2021.

[133] Nahida Kiran, Xuanlin Liu, Sihua Wang, and Changchuan Yin. VNF placement and resource allocation in SDN/NFV-enabled MEC networks. In *IEEE WCNCW*, 2020.

[134] Chen Zhiqi, Zhang Sheng, Wang Can, Qian Zhuzhong, Xiao Mingjun, Wu Jie, and Jawhar Imad. A novel algorithm for NFV chain placement in edge computing environments. In *IEEE GLOBECOM*, 2018.

[135] C. Siva Ram Murthy Prabhu Kaliyammal Thiruvasagam, Abhishek Chakraborty. Latency-aware and survivable mapping of VNFs in 5G network edge cloud. In *IEEE (DRCN)*, 2021.

[136] Dor Harris and Danny Raz. Dynamic VNF placement in 5G edge nodes. In *IEEE NetSoft*, 2022.

[137] Francesco Malandrino, Carla Fabiana Chiasserini, Gil Einziger, and Gabriel Scalosub. Reducing service deployment cost through VNF sharing. *IEEE/ACM Transactions on Networking*, 27(6), 2019.

[138] Tung V. Doan, Giang T. Nguyen, Martin Reisslein, and Frank H. P. Fitzek. SAP: Subchain-aware NFV service placement in mobile edge cloud. *IEEE Transactions on Network and Service Management*, 20(1), March 2023.

[139] Amir Mohamad and Hossam S Hassanein. On demonstrating the gain of SFC placement with VNF sharing at the edge. In *IEEE GLOBECOM*, 2019.

[140] Junjie Liu, Wei Lu, Fen Zhou, Ping Lu, and Zuqing Zhu. On dynamic service function chain deployment and readjustment. *IEEE Transactions on Network and Service Management*, 14(3), 2017.

[141] Panpan Jin, Xincai Fei, Qixia Zhang, Fangming Liu, and Bo Li. Latency-aware VNF chain deployment with efficient resource reuse at network edge. In *IEEE INFOCOM*, 2020.

[142] Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Otto Carlos Muniz Bandeira Duarte. Orchestrating virtualized network functions. *IEEE Transactions on Network and Service Management*, 13(4), 2016.

[143] Amir Mohamad and Hossam S. Hassanein. PSVShare: A priority-based SFC placement with VNF sharing. In *IEEE NFV-SDN*, 2020.

[144] Amin Ebrahimzadeh, Nattakorn Promwongsa, Seyedeh Negar Afrasiabi, Carla Mouradian, Wubin Li, Ákos Recse, Róbert Szabó, and Roch H Glitho. H-horizon sequential look-ahead greedy algorithm for VNF-FG embedding. In *IEEE NFV-SDN*, 2021.

[145] Yi Yue, Bo Cheng, Meng Wang, Biyi Li, Xuan Liu, and Junliang Chen. Throughput optimization and delay guarantee VNF placement for mapping SFC requests in NFV-enabled networks. *IEEE Transactions on Network and Service Management*, 18(4), 2021.

[146] Xiangqiang Gao, Rongke Liu, and Aryan Kaushik. Virtual network function placement in satellite edge computing with a potential game approach. *IEEE Transactions on Network and Service Management*, 19(2), 2022.

[147] Ali Hmaity, Marco Savi, Leila Askari, Francesco Musumeci, Massimo Tornatore, and Achille Pattavina. Latency-and capacity-aware placement of chained virtual network functions in FMC metro networks. *Optical Switching and Networking*, 35, 2020.

[148] Cédric Morin, Geraldine Texier, Christelle Caillouet, Gilles Desmangles, and Cao-Thanh Phan. VNF placement algorithms to address the mono and multi-tenant issues in edge and core networks. In *IEEE CloudNet*, 2019.

[149] Nattakorn Promwongsa, Amin Ebrahimzadeh, Roch H. Glitho, and Noel Crespi. Joint VNF placement and scheduling for latency-sensitive services. *IEEE Transactions on Network Science and Engineering*, 9, 2022.

[150] Chang Liu, Jin Wang, Liang Zhou, and Amin Rezaeipanah. Solving the multi-objective problem of IoT service placement in fog computing using cuckoo search algorithm. *Neural Processing Letters*, 54(3), 2022.

[151] Liang Liu, Songtao Guo, Guiyan Liu, and Yuanyuan Yang. Joint dynamical VNF placement and SFC routing in NFV-enabled SDNs. *IEEE Transactions on Network and Service Management*, 18(4), Dec 2021.

[152] Amir Mohamad and Hossam S. Hassanein. Prediction-based SFC placement with VNF sharing at the edge. In *IEEE LCN*, 2022.

[153] Amir Varasteh, Basavaraj Madiwalar, Amaury Van Bemten, Wolfgang Kellerer, and Carmen Mas-Machuca. Holu: Power-aware and delay-constrained VNF placement and chaining. *IEEE Transactions on Network and Service Management*, 18(2), 2021.

[154] Cziva Richard, Anagnostopoulos Christos, and P. Pezaros Dimitrios. Dynamic, latency-optimal VNF placement at the network edge. In *IEEE INFOCOM*, 2018.

[155] Yi Yue, Xiongyang Tang, Wencong Yang, Xuebei Zhang, Zhiyan Zhang, Chuyang Gao, and Lexi Xu. Delay-aware and resource-efficient VNF placement in 6G non-terrestrial networks. In *IEEE WCNC*, 2023.

[156] Marchetto Guido, Sisto Riccardo, Yusupov Jalolliddin, and Ksentinit Adlen. Formally verified latency-aware VNF placement in industrial internet of things. In *IEEE WFCS*, 2018.

[157] Nazanin Sarrafzade, Reza Entezari-Maleki, and Leonel Sousa. A genetic-based approach for service placement in fog computing. *The Journal of Supercomputing*, 78(8), 2022.

[158] György Dósa and Jiří Sgall. Optimal analysis of best fit bin packing. In *Automata, Languages, and Programming*. Springer, 2014.

[159] Red Dead Redemption II. https://www.rockstargames.com/reddeadredemption2/. Accessed: 2023-11.

[160] Horizon Forbidden West. https://www.playstation.com/it-it/games/horizon-forbidden-west/. Accessed: 2023-11.

[161] Street Fighter 6. https://www.streetfighter.com/6. Accessed: 2023-11.

[162] FIFA 23. https://www.ea.com/games/fifa/fifa-23. Accessed: 2023-11.

[163] Portal 2. https://store.steampowered.com/app/620/Portal_2/. Accessed: 2023-11.

[164] Dota 2. https://www.dota2.com/. Accessed: 2023-11.

[165] Leage of Legends. https://leagueoflegends.com/. Accessed: 2023-11.

[166] Valorant. https://playvalorant.com/. Accessed: 2023-11.

[167] Counter-Strike. https://www.counter-strike.net/. Accessed: 2023-11.

[168] Fortnite. https://www.fortnite.com/. Accessed: 2023-11.

[169] Player Unknown Battle Ground (PUBG). https://pubg.com/en-na. Accessed: 2023-11.

[170] Fall Guys. https://www.fallguys.com/en-US. Accessed: 2023-11.

[171] Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Placing virtual machines to optimize cloud gaming experience. *IEEE Transactions on Cloud Computing*, 3(1), 2015.

[172] Jatinder ND Gupta and Johnny C Ho. A new heuristic algorithm for the one-dimensional bin-packing problem. *Production planning & control*, 10(6), 1999.