Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

an der Universität Hamburg eingereichte

Doctoral Dissertation

Software-Defined Overlay Networking for the Deployment of Distributed Applications at the Edge

Heiko Tobias Bornholdt

Faculty of Mathematics, Informatics, and Natural Sciences
Department of Informatics
Computer Networks (NET) Group

Pinneberg, September 30, 2024

reviewers:

Prof. Dr. Winfried Lamersdorf
Prof. Dr. Christian Becker

Disputation date: December 12, 2024

urn:

`urn:nbn:de:gbv:18-ediss-124485`

I'm limited by the technology of my time.

— Howard Stark

Abstract

Today, most applications are distributed to manage the growing volume of data and processes. Simultaneously, the need for low latency, bandwidth efficiency, and privacy has driven the deployment of distributed applications to the network edge. However, the edge poses unique challenges, including widely distributed, heterogeneous, uncontrolled, and untrusted environments.

Accordingly, this *monographic* thesis presents contributions to simplify the development and deployment of distributed applications at the network edge.

The first contribution addresses the challenges of restricted connectivity in edge networks and the risk of edge device communication in untrusted networks to eavesdropping or forgery. Therefore, this contribution presents an approach for the efficient and secure linking of edge devices. Hence, this thesis introduces a new protocol that efficiently overcomes these connectivity restrictions and simultaneously establishes secure communication channels.

The second contribution simplifies the integration of dynamic overlay networks into edge applications. Previous work often restricts the application design or makes strong assumptions about the underlying network infrastructure. Therefore, this thesis introduces a new application programming interface for overlay network integration into edge applications. The approach establishes an IP overlay that can be adapted by any IP-based application without modification. Overlay services like routing, forwarding, quality of service, or security are dynamically defined and transparently applied to selected application communications. Such overlay services help applications in maximizing edge resource usage.

The third contribution simplifies the deployment of edge applications. Lack of central control, restricted reachability, and heterogeneous edge devices complicate deployment in edge environments. The approach brings capabilities known from software-defined networking, such as centralized network view and control, a common interface for edge device configuration, network configuration based on high-level functional requirements, and a closed-loop mechanism identifying and reverting unintended network changes to the edge.

All contributions have been combined into a fully functional open-source middleware, which was also evaluated in real-world experiments to demonstrate their seamless integration and functional characteristics.

Kurzfassung

Viele Anwendungen sind verteilt, um das wachsende Daten- und Prozessvolumen zu bewältigen. Niedrige Latenzzeiten, eine effiziente Bandbreitennutzung und ein verbesserter Datenschutz führen dazu, dass verteilte Anwendungen näher an die Datenquelle, an den so genannten „Rand“ des Netzwerks, verlagert werden. Dieses Anwendungsmodell bringt Herausforderungen aufgrund unkontrollierter, heterogener und unsicherer Netzwerke mit sich.

Daher stellt diese Arbeit Beiträge vor, die die Entwicklung und Bereitstellung verteilter Anwendungen am Netzwerkrand vereinfachen.

Im ersten Beitrag wird ein Ansatz zur effizienten und sicheren Vernetzung von Edge-Geräten vorgestellt. Er behandelt die Herausforderungen der eingeschränkten Konnektivität in Edge-Netzwerken und das Risiko von Abhör- oder Manipulationsversuchen in unsicheren Netzwerken. Es wird ein neues Netzwerkprotokoll vorgestellt, das diese Einschränkungen effizient überwindet und gleichzeitig eine sichere Kommunikation ermöglicht.

Im zweiten Beitrag wird ein Ansatz zur einfachen Integration von Overlay-Netzwerken in Edge-Anwendungen vorgestellt. Dies ist relevant, da existierende Arbeiten oft das Anwendungsdesign einschränken oder strenge Annahmen über die zugrundeliegende Netzwerkinfrastruktur treffen. Daher wird eine neue API zur flexiblen Integration von Overlay-Netzwerken in Edge-Anwendungen vorgestellt. Der Ansatz konstruiert ein IP-Overlay und erlaubt so die Benutzung durch jede IP-basierte Anwendung ohne Anpassung. Transparent für die Anwendung werden dabei dynamisch Overlay-Dienste wie z.B. Routing, QoS und Sicherheit auf ausgewählte Kommunikation angewendet.

Der dritte Beitrag stellt einen Ansatz für die vereinfachte Bereitstellung von Anwendungen im Edge-Bereich vor. Fehlende zentrale Kontrolle sowie heterogene und schwer zu erreichende Edge-Geräte stellen eine Herausforderung für die Bereitstellung dar. Der vorgestellte Ansatz erleichtert dies durch zentrale Kontrolle, eine einheitliches Konfigurationsinterface, Ziel-basierte Netzkonfiguration und einer Closed-Loop-Kontrolle zur automatischen Erkennung und Reparatur von unbeabsichtigten Netzveränderungen, angewendet werden.

Diese Beiträge wurden in eine voll funktionswähige Open-Source-Middleware integriert und durch mehrere Echtwelt-Experimente auf ihr Zusammenspiel und Funktionalität hin evaluiert.

Danksagung

Ohne die tatkräftige Unterstützung vieler Menschen, die mich in unterschiedlicher Form und in unterschiedlichem Maße begleitet haben, wäre der erfolgreiche Abschluss meiner Promotion nicht möglich gewesen.

Zunächst möchte ich mich bei meinem Doktorvater, Winfried Lamersdorf, besonders bedanken, der mir die Möglichkeit zur Promotion gegeben hat und mich allen Widrigkeiten zum Trotz während des gesamten Prozesses unterstützt hat. Durch die Freiheit, mein Vorhaben selbst zu gestalten, konnte ich vielschichtige Erfahrungen im wissenschaftlichen Arbeiten und im universitären Alltag sammeln. Ebenso gilt mein tiefer Dank Mathias, der mich bei sich aufgenommen und mich unermüdlich durch den Dschungel der Wissenschaft manövriert hat. Mit seinen fundierten Rückmeldungen und unserem konstruktiven - oft lebendigen - Austausch, hat er mich an meinem Promotionsvorhaben wachsen lassen. Seine Unterstützung und Begleitung bis zum Schluss haben entscheidend dazu beigetragen, dass ich meine Promotion erfolgreich abschließen konnte.

Ebenso möchte ich den (ehemaligen) Mitgliedern der Arbeitsbereiche DBIS, DOS, NET und VSYS danken. Ich möchte Anne, Felix, Janick, Philipp und Wolf hervorheben, die mich jeweils auf ihre ganz eigene Weise unterstützt haben.

Mein Dank gilt auch Christiane, die mich über viele Jahre auf dieser Reise begleitet und unterstützt hat. Ein besonderer Dank gilt auch Kevin, mit dem ich unzählige leidenschaftliche Diskussionen geführt habe – zu jeder Tages- und Nachtzeit – über die unterschiedlichsten Ansätze zur Lösung von Problemen. Ich blicke mit Freude auf die Zeit zurück, in der wir gemeinsam an „unserem System“ arbeiten konnten. Ein herzlicher Dank gilt Conny, die mich in den letzten und intensivsten Monaten meines Promotionsvorhabens unermüdlich unterstützt hat und in dieser Zeit auch zu einer wichtigen emotionalen Stütze für mich wurde. In den letzten Jahren sind sie für mich weit mehr als nur Arbeitskollegen geworden; mit ihnen haben sich tiefgehende, persönliche Verbindungen entwickelt, die über die berufliche Zusammenarbeit hinausgehen.

Zum Schluss danke ich meiner Familie, die es mir ermöglicht hat diesen Weg einzuschlagen. Ein großer Dank geht an meine Mutter, die mich überhaupt erst dazu gebracht hat, ein Informatik-Studium zu beginnen.

Contents

1 INTRODUCTION 1
1.1 Motivation and problem statement . . . 1
1.2 Research questions . . . 2
1.3 Contributions . . . 4
1.4 Outline . . . 7
1.5 Publications . . . 8
2 BACKGROUND 11
2.1 Software-defined networking . . . 11
2.2 Intent-based networking . . . 13
2.3 Overlay networks . . . 14
2.4 Network address translation . . . 18
2.4.1 NAT classification . . . 20
2.4.2 NAT traversal . . . 24
2.5 TLS fundamentals . . . 29
2.5.1 Full handshake . . . 30
2.5.2 0-RTT handshake . . . 30
2.6 The Tasklet system . . . 31
2.7 Chapter summary . . . 33
3 REQUIREMENTS ANALYSIS 35
3.1 Software-defined overlay networking scenario . . . 35
3.2 Stakeholders . . . 37
3.3 Functional requirements . . . 38
3.4 Nonfunctional requirements . . . 39
3.5 Chapter summary . . . 41
4 RELATED WORK 43
4.1 Classification . . . 43
4.2 Design methodologies for overlay networks . . . 44
4.3 Coding-support for overlay networks . . . 50
4.4 Testing frameworks for overlay networks . . . 54
4.5 Strategies for overlay network deployment . . . 56
4.6 Monitoring and managing overlay networks . . . 60
4.7 Chapter summary . . . 65
5 SYSTEM ARCHITECTURE FOR SDON AT THE EDGE 69
5.1 Software-defined overlay networking system model . . . 70

5.2 Overview on the distributed middleware system 73
5.3 Communication layer 78
5.3.1 Identity manager 79
5.3.2 Peer discoverer 80
5.3.3 Location-aware router 84
5.4 Service layer 86
5.4.1 Library of service mechanisms 87
5.4.2 Service goals interpreter 88
5.4.3 Service implementer 90
5.5 Software-defined overlay networking layer 92
5.5.1 Intent translator 94
5.5.2 Intent activator 95
5.5.3 Intent assurer 98
5.6 Chapter summary 100
6 IMPLEMENTATION OF A SDON MIDDLEWARE 101
6.1 The SDON middleware 101
6.2 Communication layer 102
6.2.1 Identity manager 102
6.2.2 Peer discoverer 103
6.2.3 Location-aware router 108
6.3 Service layer 110
6.3.1 Library of service mechanisms 110
6.3.2 Service goals interpreter 113
6.3.3 Service implementer 116
6.4 Software-defined overlay networking layer 116
6.4.1 Intent translator 117
6.4.2 Intent activator 121
6.4.3 Intent assurer 124
6.5 Chapter summary 124
7 EVALUATION 127
7.1 Evaluation of the secure connectivity protocol 127
7.1.1 Setup 128
7.1.2 Metrics 129
7.1.3 Results 130
7.1.4 Discussion of results and conclusion 131
7.2 Edge computing evaluation using the middleware 132
7.2.1 Setup 133
7.2.2 Metrics 139

7.2.3 Results 140
7.2.4 Conclusion 143
7.3 Evaluation of centrally-optimized overlay networks 144
7.3.1 Latency-based routing optimization 145
7.3.2 CPU-based node arrangement optimization 150
7.3.3 Conclusion 155
7.4 Chapter summary 155
8 CONCLUSION 157
A APPENDIX 163
A.1 Additional publications 163
A.2 Supervised theses 164
List of figures 167
List of tables 170
List of snippets 171

BIBLIOGRAPHY 173

1 Introduction

1.1 Motivation and Problem Statement

Most modern applications are designed to be distributed in order to handle the ever-increasing volume of data and processes of a progressively digital world. Distributed applications provide a level of scalability and availability that is unattainable for centralized applications. The development of distributed applications requires, however, careful consideration to aspects such as communication, coordination, consistency, replication, fault tolerance, and security [ST23]. The common denominator among all these aspects is the goal of maximizing the utilization of available computing resources.

While distributed applications in cloud environments benefit from access to a virtually unlimited pool of well-defined and highly configurable resources within a common infrastructure, edge computing applications often operate in untrusted environments. This kind of environment presents unique challenges for resource management due to its heterogeneous, uncontrolled, and unknown infrastructure [Ren+19; Mao+17]. Despite these challenges, the applications' need for low latency, high bandwidth efficiency, and privacy makes it necessary to run applications in edge environments instead.

One way of dealing with these challenges is the use of the overlay network technique. Overlay networks can help to manage edge devices by adding a logical layer between the network and the application layer. An overlay network provides services to an application layer without requiring changes to the underlay, thereby granting an idealized view and access over the edge. Despite its effectiveness in optimizing resource management, the development and deployment process of overlay networks at the edge presents several major problems:

Problem 1: Restricted connectivity and lack of trust in edge networks Edge devices are typically distributed across different networks. While connectivity within individual networks is usually unrestricted, it is often limited between different networks. Restrictions stem from incompatible network configurations and middleboxes (e.g., NATs and firewalls) that route

communication between these networks. Additionally, edge devices are often operated in untrusted networks, introducing the risk that transmitted data may be intercepted or manipulated. Therefore, overcoming connectivity barriers and implementing security services are essential for constructing overlay networks that span edge devices efficiently and securely.

PROBLEM 2: INFLEXIBLE OVERLAY NETWORK INTEGRATION INTO APPLICATIONS Integrating an overlay network into an application is a prerequisite for accessing the overlay's services. However, this integration can be complex, potentially introducing constraints on the application's design and functionality. Often, developers must adhere to a specific paradigm or build their application using a particular framework, which may not align with the application's original design goals [STS08; Rod+04; BP12; BL18; Lig09]. In addition, the services that an overlay can provide must be as versatile in their functionality as the application that uses the overlay. Many overlay network systems are inflexible in terms of overlay service selection. As a result, efficiently utilizing edge resources is difficult without the ability to select available or implement custom overlay services to help deal with edge constraints and heterogeneity. Therefore, the seamless integration of flexible overlay networks into the application is essential to assist developers in meeting their requirements.

PROBLEM 3: COMPLEX APPLICATION DEPLOYMENT TO THE EDGE Deploying distributed applications at the network edge introduces significant complexity due to the heterogeneity of devices and restrictive network conditions, making it challenging to access and manage these devices efficiently. The lack of central control makes it difficult to achieve consistent and coordinated actions across all edge devices hosting an application. Developers are required to identify low-level configuration steps for individual edge devices necessary to achieve the desired application development. After identifying these steps, these configurations must be applied on the edge devices. This process is complex and prone to errors. Therefore, application developers would benefit from a simplified edge deployment and a goal-oriented overlay networking, based on high-level functional requirements.

1.2 RESEARCH QUESTIONS

The problems outlined in the previous section highlight the need for advanced resource management and more flexibility in programming, deploying, and

operating overlay networks in untrusted edge environments. To address these problems, we[1] propose a new terminology called *software-defined overlay networking*. This new terminology reflects the thesis' incorporation of the advanced network management capabilities of software-defined networking (SDN) with the flexibility of overlay networks, allowing operation without altering the underlying network infrastructure. SDN is a paradigm where networks are automatically configured based on intents (high-level functional requirements), which are automatically translated to policies configuring network devices [Kre+15; Ope24; Cle+22; LF23]. SDN systems provide automatic network device configuration and ensure that the network is kept in the desired state in response to changes in available underlying resources. By collecting analytics and distilled insights, every unintended network change is tracked, potentially triggering automatic reconfiguration of the network devices. Further, an SDN system provides application programming interfaces (APIs) that enable centralized control of the network through a unified interface and configuration language. In this thesis, these SDN principles are applied to run overlay networks in edge networks, allowing similar network management capabilities at the edge without the need to reconfigure the underlying network infrastructure (Figure 1.1). This helps running distributed applications in untrusted edge environments, as constructed overlay networks provide an idealized view of the underlying network that matches the applications' needs. While providing the application with an idealized environment to run, the software-defined **overlay** networking system transparently provides means to cope with restricted communication and untrusted edge environments (Problem 1) as well as seamless overlay network integration and customization (Problem 2). Further, the API offers centralized control over the overlay network and simplified application deployment to the edge (Problem 3).

1 For better readability, the author refers to himself as *we* in the remainder of this thesis. Contributions are explicitly credited when resulting from collaborative efforts.

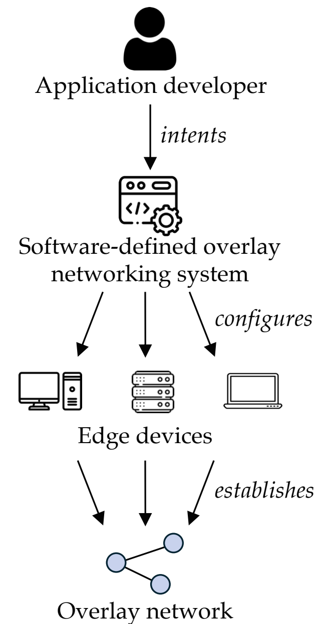


Figure 1.1: Software-defined overlay networking.

In this context, the following research questions and their corresponding sub-research questions are considered:

- **RQ 1** *How to overcome the connectivity restrictions and lack of trust in edge networks efficiently?*

> **RQ 1.1** How to achieve connectivity between edge devices efficiently?

> **RQ 1.2** How to ensure that edge devices are linked securely?

- **RQ 2** *How to seamlessly integrate dynamic overlay networks into edge applications?*

  > **RQ 2.1** How to integrate overlay networks into applications without imposing restrictions on application design and functionality?

  > **RQ 2.2** How to improve the use of heterogeneous edge resources through dynamic overlay networks?

- **RQ 3** *How to simplify efficient overlay network programming and deployment at the network edge?*

  > **RQ 3.1** How to program overlay networks that assist application developers to use edge resources efficiently?

  > **RQ 3.2** How to simplify overlay network deployment and management at the network edge?

1.3 CONTRIBUTIONS

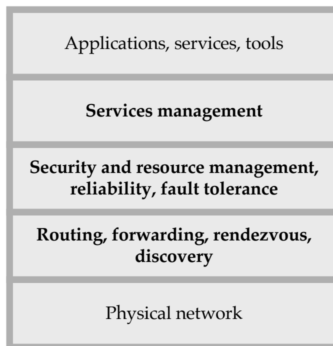


Figure 1.2: Layers common to all overlay networks [Tar10].

This thesis answers the research questions regarding flexible overlay network development, deployment, and operation in untrusted edge environments. This section briefly overviews the thesis's contributions and explains their relevance to the research questions.

This thesis presents three contributions, each advancing towards software-defined overlay networking at the edge. These contributions build upon one another. From bottom to top, the contributions loosely correspond to the three layers common to all overlay networks (see Figure 1.2) [Tar10]. Figure 1.3 summarizes the contributions and illustrates how they are related. The system described by these contributions has been fully implemented as a functional middleware. It is released as open-source software under a permissive license [BR23].

CONTRIBUTION 1: EFFICIENT AND SECURE EDGE DEVICE LINKING This contribution presents an approach for the efficient and secure linking of edge devices. This contribution addresses two problems present in edge computing: First, edge devices are often distributed across different networks, with connectivity between these networks being restricted. This restriction is caused by incompatible network configurations or middleboxes (e.g., NATs and firewalls) that route communication between the networks. Second, edge devices are often operated in untrusted networks, making any communication between these devices vulnerable to eavesdropping or forgery.

A new network protocol is proposed to address these two issues and answer research question RQ 1. It enables edge devices to link efficiently and securely with each other. This is achieved by combining middlebox traversal techniques, hole punching, and relaying with a Diffie-Hellman key agreement [BRK21; BRF23a]. This results in the protocol saving 1 to 2 RTTs compared to other approaches.

CONTRIBUTION 2: SEAMLESS DYNAMIC OVERLAY NETWORK INTEGRATION The second contribution presents an approach for simplifying the integration of dynamic overlay networks into edge applications without imposing restrictions on the application's design or functionality. This is relevant because existing overlay network approaches often need the application to be adapted to a specific overlay network implementation or impose restrictions on the application design [STS08; Rod+04; BP12; BL18; Lig09]. In addition, offloading application functionalities to the overlay would ease application development, but can be difficult to achieve. This is especially relevant for edge computing, where the overlay can help maximize edge resource usage.

To answer RQ 2, we introduce a new API for seamless dynamic overlay network integration into applications that help to utilize edge resources efficiently [Bor+23; Röb+23]. The proposed API establishes an IP overlay that transparently applies overlay services. While the edge application components communicate using regular IP packets, the overlay processes the packets according to configured services. Such services include routing, forwarding, quality of service (QoS), communication encryption, and other mechanisms that help improve the use of unreliable and heterogeneous edge resources. Therefore, this API allows a seamless integration of dynamic overlay networks into all IP-based applications.

CONTRIBUTION 3: SIMPLIFIED OVERLAY NETWORK DEPLOYMENT AT THE EDGE Finally, this contribution presents an approach for simplified

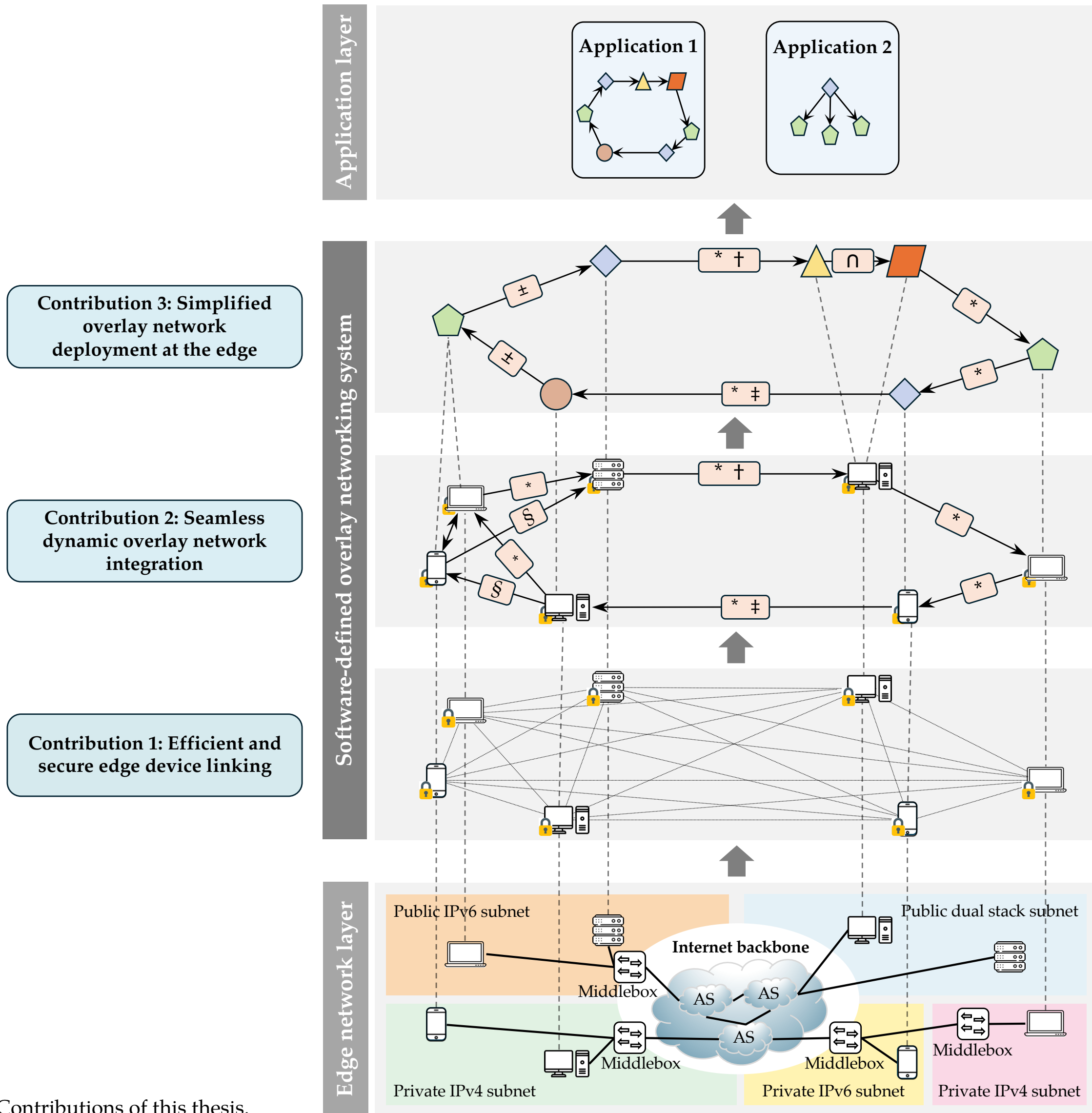


Figure 1.3: Contributions of this thesis.

development and deployment of distributed applications running in restricted edge networks. Deploying edge applications is inherently complex and error-prone due to the absence of centralized control and the heterogeneity and restrictions of edge devices.

To address RQ 3, this contribution presents an overlay networking system that adapts principles of software-defined networking to overlay networking at restricted edge networks [Bor+25]. Software-defined networking capabilities such as centralized control, a common interface for edge device configuration, network configuration based on high-level functional requirements, and a closed-loop mechanism to help identify and revert unintended network changes are brought to the edge through this contribution.

1.4 OUTLINE

Table 1.1 provides an overview of this thesis, focusing on the research questions, contributions, and publications released within the scope of this thesis. This thesis is structured as follows:

CHAPTER 2 provides the required background for this thesis on the terms of software-defined networking, intent-based networking, overlay networks, NAT and TLS fundamentals.

CHAPTER 3 derives requirements for a software-defined overlay networking system.

CHAPTER 4 discusses and compares related work with requirements identified in the previous chapter.

CHAPTER 5 presents an architecture of a software-defined overlay networking system.

CHAPTER 6 presents the implementation of a software-defined overlay networking middleware.

CHAPTER 7 evaluates the implemented prototype by conducting simulations and real-world experiments, showing that the system requirements are fulfilled.

CHAPTER 8 concludes this thesis by summarizing the contributions and providing an outlook to future work in the area of software-defined overlay networking.

Table 1.1: The outline of the thesis.

Section	Research question	Contribution	Publications
Chapter 1: Introduction			
Chapter 2: Background			
Chapter 3: Requirements analysis			
Chapter 4: Related work			
Chapter 5: Architecture			
Section 5.3	RQ 1	Contribution 1	[BRK21; BRF23a]
Section 5.4	RQ 2	Contribution 2	[Bor+23; Röb+23]
Section 5.5	RQ 3	Contribution 3	[Bor+25]
Chapter 6: Implementation			
Chapter 7: Evaluation			
Chapter 8: Conclusion			

1.5 PUBLICATIONS

Our publications that are within the context of this thesis were peer-reviewed and published. Additionally, open-source software implementations were released to encourage further research in these areas. The appendix also includes a list of additional publications to which the author contributed (see Appendix a.1) and student theses supervised by the author (see Appendix a.2).

1.5.1 *Main publications*

[Bor+25] Heiko Bornholdt, Kevin Röbert, Stefan Schulte, Janick Edinger, and Mathias Fischer. "A Software-Defined Overlay Networking Middleware for a Simplified Deployment of Distributed Application at the Edge." In: *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. SAC '25. Catania, Italy: Association for Computing Machinery, 2025 (cited on pages 7, 8, 160).

[Bor+23] Heiko Bornholdt, Kevin Röbert, Martin Breitbach, Mathias Fischer, and Janick Edinger. "Measuring the Edge: A Performance Evaluation of Edge Offloading." In: *2023 IEEE International Conference*

on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). 2nd Workshop on Serverless computing for pervasive cloud-edge-device systems and services (*LESS'24). Atlanta, GA, USA: IEEE, 2023, pages 212–218. DOI: 10.1109/PerComWorkshops56833.2023.10150261 (cited on pages 5, 8, 32, 132, 159).

[BRF23a] Heiko Bornholdt, Kevin Röbert, and Mathias Fischer. "Low-Latency TLS 1.3-Aware Hole Punching." In: *ICC 2023 - IEEE International Conference on Communications*. IEEE International Conference on Communications (ICC) 2023. Rome, Italy: IEEE, 2023, pages 1481–1486. DOI: 10.1109/ICC45041.2023.10279326 (cited on pages 5, 8, 27, 29, 127, 158).

[Bor21] Heiko Bornholdt. "Towards Citizen-Centric Marketplaces for Urban Sensed Data." In: *Advances in Service-Oriented and Cloud Computing, International Workshops of ESOCC 2020*. 8th European Conference On Service-Oriented And Cloud Computing (ESOCC 2020). Heraklion, Crete, Greece: Springer Cham, 2021, pages 140–150. DOI: 10.1007/978-3-030-71906-7_12.

[BBP21] Heiko Bornholdt, Dirk Bade, and Wolf Posdorfer. "Incorum: A Citizen-Centric Sensor Data Marketplace for Urban Participation." In: *Advances in Computer, Communication and Computational Sciences, Proceedings of IC4S 2019*. International Conference on Computer, Communication and Computational Sciences (IC4S 2019). Bangkok, Thailand: Springer Singapore, 2021, pages 659–669. DOI: 10.1007/978-981-15-4409-5_59.

[BRK21] Heiko Bornholdt, Kevin Röbert, and Philipp Kisters. "Accessing Smart City Services in Untrustworthy Environments via Decentralized Privacy-Preserving Overlay Networks." In: *2021 IEEE International Conference on Service-Oriented System Engineering (IEEE SOSE 2021)*. 15th IEEE International Conference On Service-Oriented System Engineering (IEEE SOSE 2021). Oxford, United Kingdom: IEEE, 2021, pages 144–149. DOI: 10.1109/SOSE52839.2021.00021 (cited on pages 5, 8).

1.5.2 *Software implementations*

[BR23] Heiko Bornholdt and Kevin Röbert. *drasyl - a middleware for rapid development of distributed applications*. Software. Version 0.10.0. Jan. 2023. DOI: 10.25592/uhhfdm.14206. URL: https://github.com/drasyl/drasyl (cited on pages 4, 101).

[Bor23a] Heiko Bornholdt. *NatPy - python-based network address translator with configurable mapping, allocation, and filtering behavior for Netfilter NFQUEUE*. Software. Dec. 2023. DOI: 10.25592/uhhfdm.14208. URL: https://github.com/HeikoBornholdt/NatPy (cited on page 147).

[Bor23b] Heiko Bornholdt. *netty-tun - netty channel communicating via TUN devices*. Software. Version 1.2.2. Apr. 2023. DOI: 10.25592/uhhfdm.14210. URL: https://github.com/drasyl/netty-tun (cited on page 116).

[BRF23b] Heiko Bornholdt, Kevin Röbert, and Mathias Fischer. *Low-Latency TLS 1.3-Aware Hole Punching Proof of Concept*. Software. Jan. 2023. DOI: 10.25592/uhhfdm.16613. URL: https://github.com/HeikoBornholdt/tls-hole-punching (cited on page 129).

2 Background

This chapter presents the nomenclature and methodological background necessary for the remainder of this thesis. The chapter is organized into the following sections:

- *Section 2.1 covers software-defined networking, which enables dynamic and programmable network configuration, forming the inspiration for software-defined overlay networking which applies this principles to overlay network programming.*
- *Section 2.2 introduces intent-based networking, where some concepts of which are applied to software-defined overlay networking.*
- *Section 2.3 explains overlay networks, highlighting their ability to abstract from underlying limitations, a key aspect of the approach in this work.*
- *Section 2.4 discusses network address translation, which complicate overlay network construction on the Internet by rendering many hosts unroutable, thus necessitating traversal techniques and careful assessment of host constrains.*
- *Section 2.5 provides an overview of the Transport Layer Security protocol, essential for securing communication between overlay nodes in untrusted environments.*
- *Section 2.6 introduces the Tasklet computation-offloading system.*
- *Section 2.7 summarizes this chapter.*

2.1 SOFTWARE-DEFINED NETWORKING

Software-defined networking (SDN) is a network architecture that enables dynamic, centralized, and programmatically efficient network management by decoupling the control plane from the data plane. Traditionally, network operators were constrained by proprietary software from vendors like Cisco and Juniper, which required manual setups and provided limited flexibility. SDN was developed to overcome these limitations by providing standardized interfaces like OpenFlow, allowing network configuration independent of the

vendor [TFW21]. In traditional networks, both planes are tightly integrated within the same network devices, which makes network management complex and static. In contrast, SDN separates these two planes:

- The *control plane* is responsible for the logic that determines traffic routing, such as selecting paths and managing network policies.
- The data plane consists of hardware (e.g., switches and routers) that forwards packets based on instructions from the control plane.

By moving the control plane into centralized controllers, SDN enables uniform configuration and management across heterogeneous network devices, regardless of vendor. Instead of manually configuring individual devices, SDN controllers automate network policies and configurations via standardized application programming interfaces (APIs). This centralization simplifies network management, enhances flexibility, and allows for real-time reconfigurations based on traffic patterns or policies.

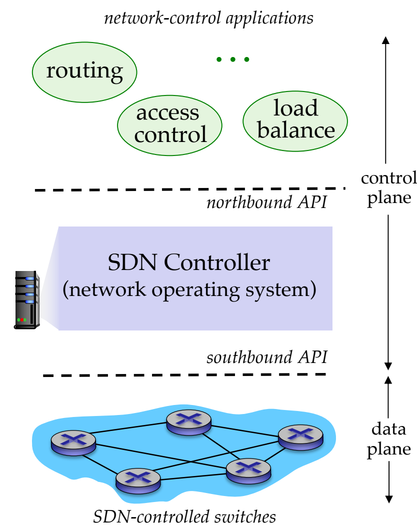


Figure 2.1: SDN-controlled switches in the data plane are configured by an SDN controller, which implements the network-control applications' configurations from the control plane [Fis22].

Figure 2.1 illustrates the architecture of a typical SDN system. At the bottom, SDN-enabled switches are configured by the control plane of an SDN controller. These switches are controlled through a southbound API using protocols such as OpenFlow. The SDN controller centralizes control logic and interacts with network control applications through a northbound API. At the top, network-control applications implement control functions using services and APIs provided by the SDN controller, enabling a flexible and unbundled network control ecosystem. [Fis22].

After discussing SDN and its capabilities for dynamic network configuration, intent-based networking (IBN) is explored. IBN further automates network management, therefore helping reduce manual intervention and minimizing configuration errors.

2.2 INTENT-BASED NETWORKING

Intent-based networking (IBN) builds upon the foundation of SDN (Section 2.1) to simplify network management and configuration through automation and abstraction [LF23]. While SDN centralizes control and allows dynamic network provisioning via a common API, it still requires manual policy definition and oversight [Doy22]. In contrast, IBN introduces a higher level of abstraction by allowing network operators to define "intents" as declarative, high-level goals describing the desired network behavior [Cle+22]. This omits the necessity to identify the individual configuration steps required to achieve the goal, which is a complex and error-prone task. Intents can be stated in natural language, such as a network operator specifying, "ensure all traffic between finance department is encrypted and prioritized over other internal traffic during business hours." or a developer requesting, "Ensure my video streaming application maintains a minimum bandwidth of 5 Mbit/s for a smooth user experience". An IBN system automatically translates intents into configurations and applies them to the network devices. Additionally, a closed-loop mechanism is integrated into the IBN system to continuously monitor the network, ensuring compliance and making real-time adjustments in response to unintended network changes [LF23].

A typical IBN system as shown in Figure 2.2 includes the following five components [LF23]:

- **Intent profiling:** This component processes and interprets the user's high-level intents, expressed in natural language, to define the desired outcome. Further, it checks if all intents can be translated into network configuration.
- **Intent translation:** This component translates the high-level intents into low-level network policies that can be applied to specific network devices. In this step, each network device's individual capabilities are considered.
- **Intent resolution:** This component checks for conflicts or incompatibilities between new and existing intents and attempts to resolve them automatically. It detects possible interference or unwanted side effects

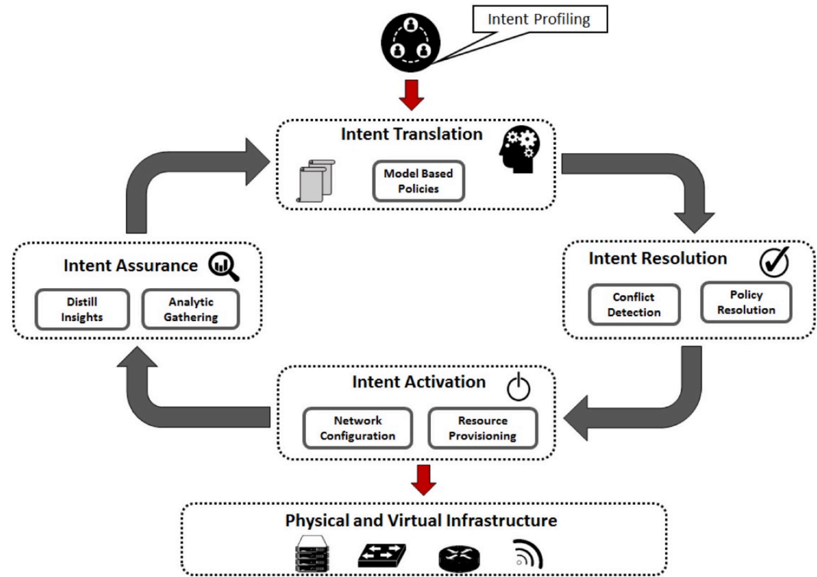


Figure 2.2: Interaction of main component of an intent-based networking system [LF23].

between intents. If automatic resolution is not possible, the user gets notified.

- **Intent activation:** This component deploys the network configuration based on the translated intent to the relevant devices.

  Intent assurance: This component ensures the network continues to behave as desired after deployment. It continuously monitors the network for unwanted changes and tries to correct any deviations from the intended configuration automatically.

In summary, IBN is built upon the foundational concepts of SDN. Further automation of network management is achieved through high-level intent declarations. Manual effort is reduced, and continuous alignment between network configurations and desired outcomes is ensured. The increasing complexity of networks is addressed through the principles of overlay networks. As a result, the overlay network provides an abstracted, idealized view while also maintaining control over the underlying infrastructure.

2.3 OVERLAY NETWORKS

Overlay networks are virtual networks built on top of existing ones. They provide their own addressing, routing, and service models while leveraging

the existing infrastructure. This virtualization layer enhances functionalities by optimizing traffic, bypassing limitations, improving scalability and security, and facilitating complex distributed applications while also providing better resilience and specialized data handling [Toy21]. As overlay networks do not require changes to the underlying network infrastructure, this technology is particularly useful for deploying new technologies or protocols like P2P systems, content delivery networks, or virtual private networks [Tar10].

An example of a simple overlay network is depicted in Figure 2.3. In this ring-shaped overlay, each node represents a user's machine participating in a distributed file-sharing system. Communication between two nodes over a virtual link e involves transmitting messages through multiple underlying network links, denoted as $e_1, \dots, e_4$. This underlying routing complexity is abstracted from the overlay's perspective. Overlay network topologies can optimize the efficiency of distributed algorithms, such as distributed search or data distribution [Sch10].

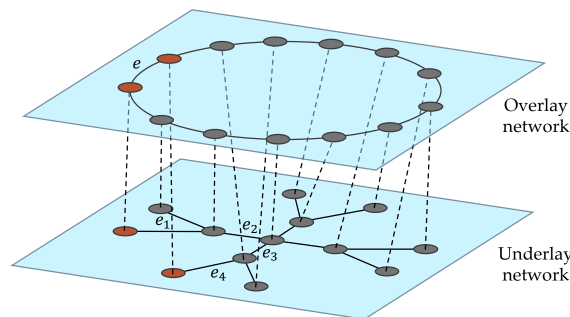


Figure 2.3: Example overlay network built on top of an Internet-style underlay [Sch10].

Depending on the requirements, different types of overlay network structures are utilized, categorized into unstructured, structured, and hybrid types, each with its own advantages and trade-offs [ST23]. Unstructured overlay networks offer high flexibility and expressiveness but face challenges with efficiency and consistency. Structured overlay networks' advantages include efficient and deterministic data access due to their well-defined topologies, although they come with increased complexity. Hybrid overlay networks combine features from both unstructured and structured approaches, aiming to balance flexibility and efficiency.

UNSTRUCTURED OVERLAY NETWORKS Unstructured overlay networks consist of nodes connected randomly, resulting in a non-deterministic structure. In unstructured overlays, maintaining the topology is simpler and more lightweight because nodes are not required to adhere to a rigid structure.

This flexibility makes the network more resilient to high node churn, as new nodes can join or leave without disrupting the overall connectivity or requiring complex reconfigurations. However, without a controlled structure, no assumptions must be made about effectively implementing distributed searches in such networks, as finding specific data is not guaranteed. Figure 2.4 illustrates an example of an unstructured overlay, with the P2P file-sharing application Gnutella serving as a practical example of its use. Each dot represents a virtual node and the rounded boxes correspond to data items with their respective keys. In Gnutella versions before 0.6 [Gnu03], search requests propagate through the network as nodes forward them to all their neighbors until a maximum of seven hops is reached [Cha+03; RF02]. This flooding can cause scalability issues due to the high traffic generated by search propagation, which at times reportedly accounted for a significant fraction of global Internet traffic [MIF02]. To increase the likelihood of finding the desired data item, it must be replicated to multiple nodes.

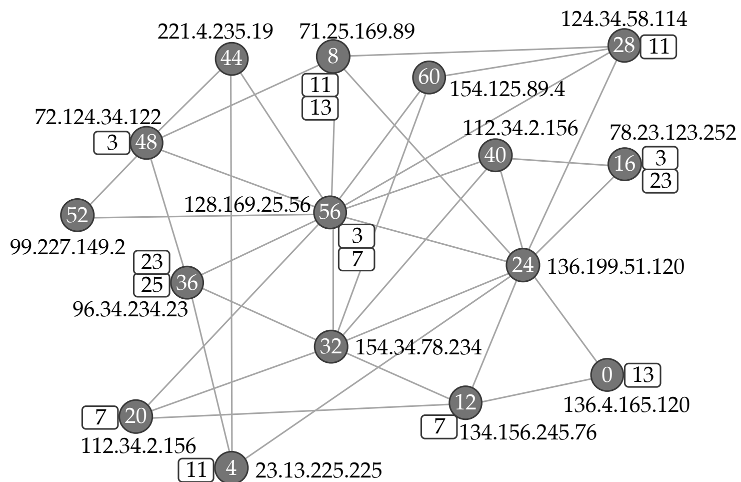


Figure 2.4: Example for an unstructured overlay network, as constructed by early version of Gnutella [Sch10].

STRUCTURED OVERLAY NETWORKS Structured overlay networks have defined topologies and algorithmic methods to organize and maintain node connections. They exhibit strict coordination, as nodes must follow specific rules for joining and leaving, resulting in higher management complexity. The deterministic nature of these networks ensures predictable routing paths, enabling efficient data localization and retrieval. In such overlays, management is distributed and employs protocols like DHTs to maintain order. The structured design ensures high consistency, allowing reliable data replication and

availability. These networks are usually hierarchical or grid-like, promoting completeness by systematically storing and retrieving data. Although structured overlay networks may limit expressiveness compared to unstructured systems, they excel in efficient and scalable query processing, making them ideal for applications that require reliable data access and distribution. The previous overlay network shown in Figure 2.3 is structured. However, the ring topology is impractical because it does not efficiently handle network dynamics and may lead to high latency and limited fault tolerance. A more realistic example of a structured overlay network is provided in Figure 2.6 which is generated by Chord [Sto+03]. While the nodes and data were randomly arranged in Figure 2.4, there is a strict sorting here. The lookup complexity for the Chord protocol is $O(\log N)$, where N is the number of nodes in the network. This logarithmic complexity is highly efficient, especially compared to unstructured or randomized overlay networks, where lookup operations often require traversing a significant portion of the network, leading to an expected complexity of $O(N)$ in the worst case (see Figure 2.5).

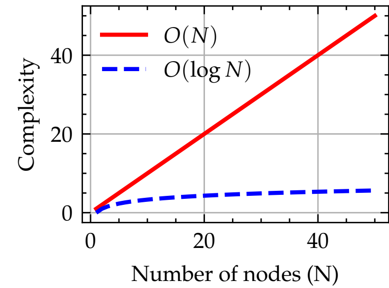


Figure 2.5: Comparison of $O(N)$ and $O(\log N)$ complexity.

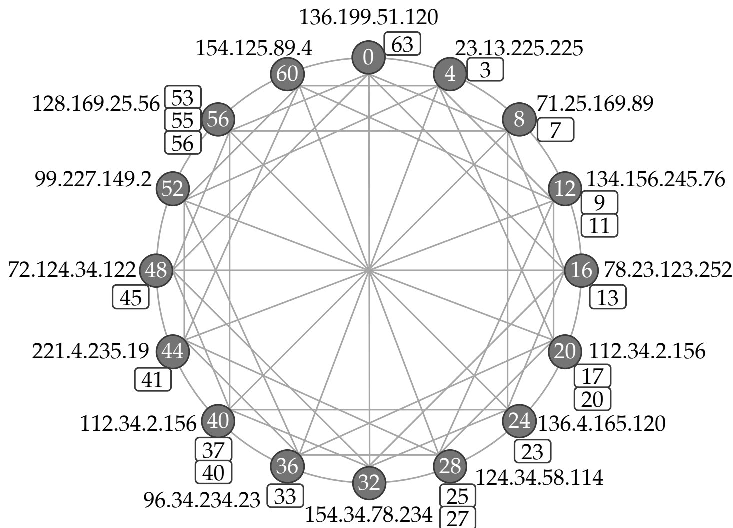


Figure 2.6: Example for a structured overlay network using a DHT, as generated by Chord [Sch10].

HYBRID OVERLAY NETWORKS Hybrid overlay networks feature moderate coupling, with some nodes functioning as super nodes or central coordinators while others operate more freely. The complexity and management of hybrid systems are higher than those of unstructured networks but lower than those of fully structured ones. These networks offer a mix of deterministic and non-deterministic features, with structured elements ensuring reliable data

access and unstructured aspects providing greater flexibility. Completeness is generally high due to organized data management, while expressiveness remains flexible, supporting varied query types. Hybrid overlay networks are versatile, optimizing resource discovery and scalability while maintaining robustness and adaptability. BitTorrent's file sharing protocol employs hybrid overlay networks, where centralized servers, known as trackers, provide a directory service by maintaining an index of available files and the peers sharing them. However, the actual file transfers occur in a decentralized, P2P manner, allowing peers to directly exchange data without relying on the centralized server for the transfer itself. An example of such an overlay network is shown in Figure 2.7, where the nodes 12, 24, 20 and 44 act as super peers.

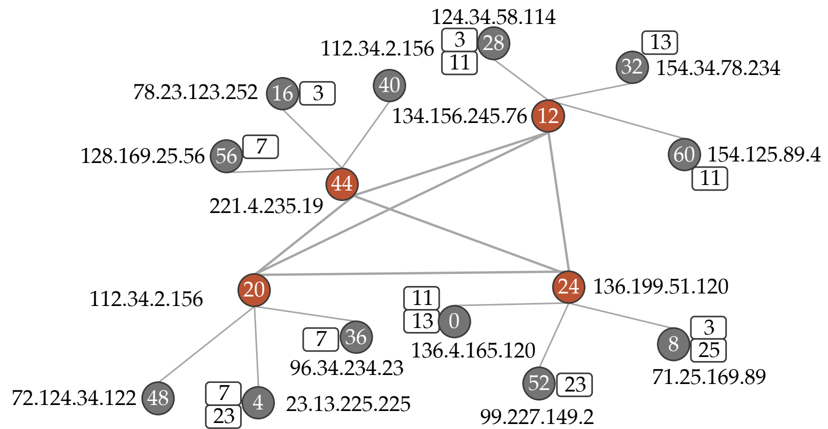


Figure 2.7: Example for a hybrid overlay network. Hierarchical topology using super peers each serving a subset of leaf peers. [GG09]

After discussing the various types of overlay networks, it is important to consider the practical challenges they face in real-world deployment. One significant challenge is the presence of NAT, which can hinder direct communication between peers. This limitation prevents the overlay topology from being fully realized, as nodes may be unable to establish direct connections, requiring additional mechanisms to maintain network functionality.

2.4 NETWORK ADDRESS TRANSLATION

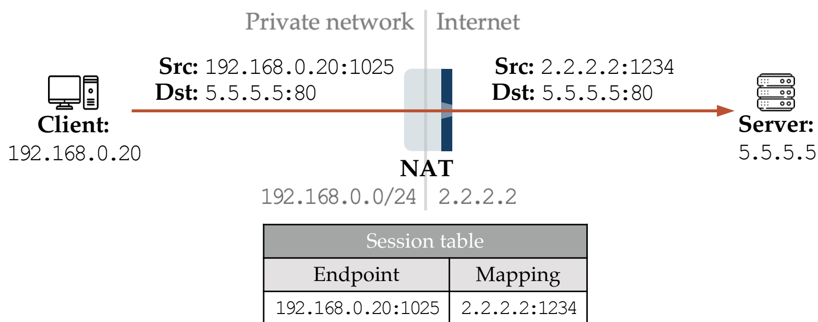
NAT is a method of mapping IP address spaces from one to another, facilitating transparent routing between private networks, as defined in [Mos+96], and external networks by modifying network address information in packet headers [HS99]. NAT is often deployed on middleboxes like Internet gateways

and firewalls. A middlebox is an intermediary network device that performs functions beyond the standard operations of an IP router [BC02].

Depending on whether the sender, receiver or both IP address information is translated, NAT is classified into source, destination, or bidirectional NAT. Source NAT translates the sender IP addresses, which is used when a private host needs to access the Internet. Accordingly, destination NAT translates the receiver addresses, which is used when an Internet host needs to access private services. Finally, bidirectional NAT combines source and destination NAT, allowing both Internet and private hosts to access private services.

Initially, NAT translated only network address information on the IP header. Later, network address/port translation (NAPT) was introduced [ES01], translating also transport identifiers (such as a TCP/UDP port numbers or ICMP query ids). [Qia24].

Today, the most common NAT deployed on the Internet is the source NAPT. This kind of NAT is typically applied on residential network routers, corporate network firewalls, or mobile carrier networks [Per+13]. Source translation helps mitigate the exhaustion problem of IPv4 addresses as multiple private hosts can be mapped to a single public IP address [ICA11]. Further, access from the Internet to a private network is restricted which is a security benefit. Today, when referring to "NAT" in the literature, it usually means source NAPT. Therefore, unless otherwise stated, this thesis always refers to this kind of translator when the term "NAT" is used.



Session table	
Endpoint	Mapping
192.168.0.20:1025	2.2.2.2:1234

Figure 2.8: Flow of source NAPT. The outgoing packet's source information with the client's address is replaced with mapped address information of the NAT device.

Figure 2.8 shows an example flow of an outgoing packet sent from client endpoint `192.168.0.20:1025` in a private network to a server endpoint `5.5.5.5:80` on the Internet, where the NAT device transparently replaces the client's endpoint information with its own mapped endpoint `2.2.2.2:1234` (an endpoint refers to an IP address and port number). The mapping is stored in the NAT

device's session table, which routes corresponding inbound packets back to the client.

2.4.1 *NAT classification*

The method by which the NAT device populates its session table, specifically when a new mapping is created or an existing one is reused, and which incoming packets are mapped to a client, depends on the NAT type. Traditionally, four NAT types are distinguished, whose different implementations are displayed in Figure 2.9 and are described now [Ros+03]:

2.4.1.1 *Traditional NAT types*

FULL-CONE NAT All requests originating from the same private endpoint are mapped to the same public endpoint. Any Internet host can communicate with the private host by sending packets to the mapped public endpoint (Figure 2.9a). This is the least restrictive type.

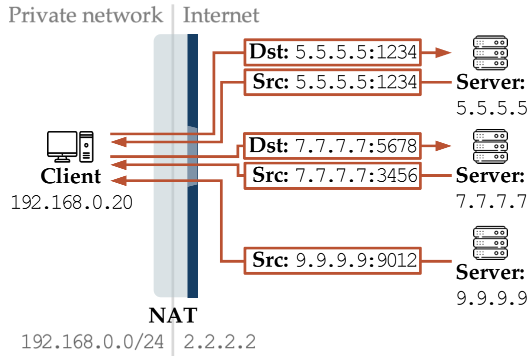
RESTRICTED-CONE NAT The mapping behavior is the same as a full-cone NAT, but the private host can only be reached by Internet hosts if the private host has previously initiated communication by sending a packet to the Internet host (Figure 2.9b).

PORT-RESTRICTED-CONE NAT This type is similar to a restricted-cone NAT, but the restriction includes port numbers. An Internet host can send packets to a private host only if the private host has previously contacted the Internet host on the same endpoint (Figure 2.9c).

SYMMETRIC NAT A new mapping is created whenever the private host contacts a new Internet host or the same host on a different port. Apart from this mapping behavior, the filtering of incoming packets is identical to that of a port-restricted cone NAT (Figure 2.9d). This is the most restrictive type.

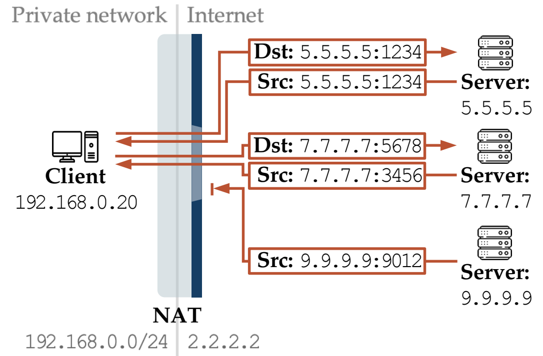
2.4.1.2 *NAT behavioral classification*

The traditional classification provides an overview of different NAT types but does not cover all the aspects of NAT behavior. For instance, these NAT types do not detail how private ports are mapped to public ports. Consequently, while the traditional classification is still commonly used in literature, a more sophis-



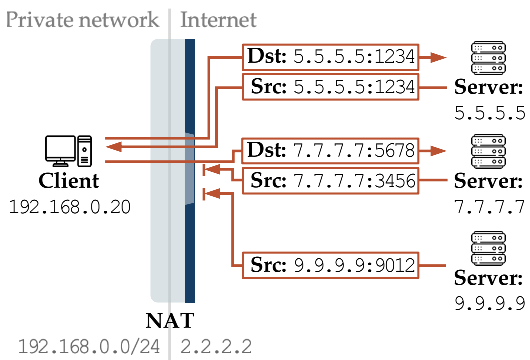
Session table		
Endpoint	Mapping	Allowed
192.168.0.20:1025	2.2.2.2:1234	*:*

(a) Full-cone NAT.



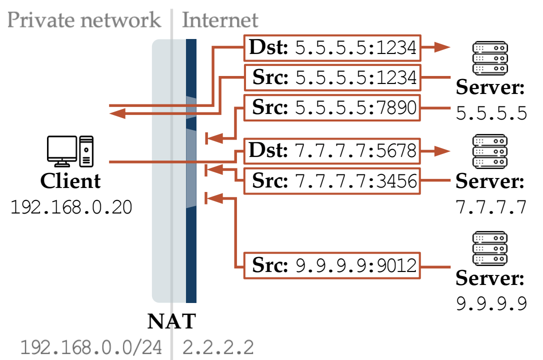
Session table		
Endpoint	Mapping	Allowed
192.168.0.20:1025	2.2.2.2:1234	5.5.5.5:*
		7.7.7.7:*

(b) Restricted-cone NAT.



Session table		
Endpoint	Mapping	Allowed
192.168.0.20:1025	2.2.2.2:1234	5.5.5.5:1234
		7.7.7.7:5678

(c) Port-restricted-cone NAT.



Session table		
Endpoint	Mapping	Allowed
192.168.0.20:1025	2.2.2.2:1234	5.5.5.5:1234
	2.2.2.2:7890	7.7.7.7:5678

(d) Symmetric NAT.

Figure 2.9: Behavior of different NAT types. Outbound client packets result in different session table entries populated.

ticated taxonomy has emerged [Pen+16]. This new behavioral classification describes the NAT as a combination of three different policies [REH09]:

MAPPING POLICY This policy is activated when a packet is sent from a private endpoint to an Internet endpoint. The mapping policy determines whether a new mapping should be added to the session table or an existing one should be reused. There are three different behaviors: *Endpoint-independent* mapping bases its decision solely on the private endpoint, reusing existing mappings as long as the private address and port remain unchanged. *Address-dependent* mapping maintains existing mappings if the public address stays the same. *Endpoint-dependent* mapping only reuses existing mappings when both the public address and port remain the same.

PORT ALLOCATION POLICY While the mapping policy determines *when* a new public port should be assigned, this policy specifies *which* port should be assigned. There are three types of behavior: *Port-preservation* allocation method keeps the same private port number in the mapping. *Port-contiguity* allocation assigns ports in a sequentially increasing order. *Random* allocation assigns a random port number.

FILTERING POLICY This policy determines whether a packet from the Internet, directed to a mapped endpoint of a NAT, is allowed to be forwarded to the corresponding private endpoint or discarded. Three different behaviors are possible: *Endpoint-independent* filtering allows packets from any public endpoint to reach the private endpoint, provided the private endpoint has previously communicated with any address. *Host-dependent* filtering only allows packets from a specific public address that the private endpoint has previously contacted. *Endpoint-dependent* filtering permitting packets only from the same public endpoint that the private endpoint has previously communicated with.

Rather than classifying NATs into just four traditional types, these policies can describe 27 different NATs. Table 2.1 illustrates how traditional types correspond to behavioral policies. This finer-grained distinction helps in better distinguishing NATs, which is important and will be further explained in the upcoming section Section 2.4.2, where several approaches for traversing connection restrictions imposed by NATs are discussed.

<table>
<tr><td colspan="2"></td><td colspan="3">Policy</td></tr>
<tr><td colspan="2"></td><td>Mapping</td><td>Port allocation</td><td>Filtering</td></tr>
<tr><td rowspan="4">NAT type</td><td>Full-cone</td><td rowspan="3">Endpoint-independent</td><td rowspan="4">undefined</td><td>Endpoint-independent</td></tr>
<tr><td>Restricted-cone</td><td>Address-dependent</td></tr>
<tr><td>Port-restricted cone</td><td rowspan="2">Endpoint-dependent</td></tr>
<tr><td>Symmetric</td><td>Address- or Endpoint-dependent</td></tr>
</table>

Table 2.1: Mapping of traditional NAT types to behavior policies.

2.4.1.3 *Further NAT behavior considerations*

NAT devices in real-world networks exhibit a wide range of behaviors, and even the classification by these three policies fails to capture all aspects. In this section, some further NAT behaviors are discussed that have been encountered during the literature review and experiment conduction in the context of this thesis.

First, it is undefined how a NAT device behaves when it exhausts its available resources. This situation can occur when all available IP addresses and ports are already assigned to existing mappings. In such cases, the NAT device may either cease routing packets for which it cannot establish a new mapping or begin cleaning up existing entries in the session table.

The mechanisms for cleaning up session table entries vary. While a NAT device can anticipate the termination of a connection-oriented communication, such as TCP, by listening for `FIN` segment, it must use timeout mechanisms for connectionless protocols like UDP. The applied timeout value can differ significantly, with experimental studies revealing most values range from 1 min to 10 min [Hät+10]. The same studies indicate that some devices apply different timeout values depending on the protocol used and whether the mapping was used for outbound or bidirectional communication.

Some NAT devices preserve port parity during mapping [JA07]. This practice is infrequent and aims to maintain compatibility with protocols like Real-time Transport Protocol (RTP), where the protocol specification recommends using even port numbers for RTP and odd-numbered ports for RTP Control Protocol (RTCP) [Sch+03].

Some NAT devices not only modify a packet's address and port information but also rewrite the payload to replace private endpoints with the mapped endpoint. This mechanism aims to enhance NAT compatibility but can interfere with protocols like Session Traversal Utilities for NAT (STUN) that need to

communicate private endpoints. As a workaround, STUN uses an XOR function to obfuscate endpoints, preventing interference from such NAT behavior.

In an attempt to standardize the behavior of NAT devices, several RFCs specifying the requirements for translating protocols such as UDP, TCP, and ICMP [JA07; For+08; Guh+09]. However, an experimental study by Hätönen et al. revealed that many devices do not conform to the RFCs specifications.

2.4.2 *NAT traversal*

In the previous section Section 2.4, the source NAT was identified as the most commonly present type, preventing Internet hosts from reaching private hosts. This significant level of unreachability poses a problem for many applications, particularly in the P2P and VoIP domains, which rely on participants being able to communicate with each other. Studies have revealed that up to 92 % of devices in today's popular distributed applications remain unreachable because of this [Haa+16; WP17]. To address this, multiple NAT traversal techniques have been established to facilitate communication across NAT devices.

As the mapping and filtering behavior of a NAT device can vary, NAT traversal can range from simple to complex, or even be unfeasible [REH09; Hät+10]. The following section presents different techniques to overcome restrictions imposed by NAT devices. The discussion includes the types of NAT that these techniques can overcome, protocols implementing these techniques, and their respective drawbacks.

2.4.2.1 *Port forwarding*

Port forwarding, sometimes referred as port mapping, is a technique where (permanent) mappings are created on a NAT device. This method enables private hosts to appear as though they are directly connected to the unrestricted public Internet. Port forwarding involves reconfiguring the NAT device, either manually by a user via a management interface or through a network protocol that allows applications running on private hosts to create these mappings automatically. Three network protocols for creating port mappings are currently used, with most devices typically supporting only one of these protocols. Port forwarding is often restricted to privileged users and disabled for security reasons in non-residential networks.

NAT PORT MAPPING PROTOCOL (NAT-PMP) Introduced in 2005 [CK13], this protocol operates over UDP and follows a simple request-response model

[HW03]. Private hosts act as clients, while the NAT device functions as the server. In NAT Port Mapping Protocol (NAT-PMP), requests are always sent by clients to UDP port 5351 of their default gateway, assuming it serves as the server.

With this protocol, clients not only request new mappings, but can also learn the external IPv4 address. A mapping request includes the desired protocol (UDP or TCP), the internal port to be mapped, the external port to use, and a requested mapping lifetime. The server treats the external port and mapping lifetime as suggestions and may adjust them. It responds with a message specifying the protocol, internal port, public port, and the mapping lifetime. Clients must refresh their mapping within the specified lifetime by sending a new request. To delete a mapping, a request with a mapping lifetime of 0 is sent.

port control protocol (pcp) Port Control Protocol (PCP) is the proposed successor to NAT-PMP, introduced in 2013, using compatible packet formats. Support is provided for IPv6, outbound mapping management, and firewall rule configuration. Compatibility with large-scale NATs using a pool of external addresses is ensured. Error lifetimes are managed, and an extension mechanism is included to facilitate future enhancements [Win+13].

PCP mapping requests are similar to those in NAT-PMP but also include the client IP address (to detect unexpected NATs between client and server), a nonce that must be copied to the response to enable client verification, and a suggestion for the public IP address to use for mapping. This suggestion is needed to ensure that a mapping will maintain its external endpoint.

universal plug and play (upnp) internet gateway device-port control protocol interworking function The Universal Plug and Play (UPnP) Internet Gateway Device (IGD)-Port Control Protocol, introduced in year 2013, is based on UPnP to add, remove, or enumerate mappings and to discover the external IP address [BPW13].

Initially, the IGD must be discovered. For this, private hosts send an IP multicast discovery request using a UPnP protocol to locate Internet gateways. Matching devices respond with a UPnP device description XML. Private hosts then verify if the gateway is connected and if it has an external address. Next, an `AddPortMapping` SOAP request is sent to the gateway, containing information about the internal endpoint, the desired external endpoint, the network protocol, and a textual description for identification. Although the protocol includes a field to specify the desired lifetime of the mapping, this field's value

should always be set to `0`, for the following reason: "[...] if the client requests a lease other than zero, some IGD home gateways may ignore the request, fail in other ways, or even crash completely." [CK13, page 30]

2.4.2.2 *Relaying*

This technique uses third-party hosts acting as intermediaries to facilitate communication between devices behind NATs. This method ensures connectivity with all kinds of NATs but can introduce additional latency, increase bandwidth costs, create a bottleneck, and present a single point of failure (SPOF). Additionally, relaying raises security and privacy concerns, as all communication passes through the relay server, potentially exposing sensitive data to unauthorized access. SOCKS and Traversal Using Relays around NAT (TURN) are two widely used protocols for relaying network traffic [Lee96; Red+20].

In both SOCKS and TURN protocols, private hosts register with a relay server, which assigns them a relay address and port. This address serves as an endpoint through which traffic sent to the relay is forwarded to the private host. Discovery of each other's relay endpoint is not part of these protocols and is typically achieved through some signaling communication implemented by the respective application.

2.4.2.3 *Hole punching*

Hole punching tries to exploit a NAT's mapping and filtering behavior to create desired entries in the session table enabling P2P connections. This technique is compatible with most NAT types, but will not work reliably if both peers are operated behind separate NATs both applying endpoint-depended mapping [GTF04; DA08]. In terms of compatibility, a distinction must be made between UDP and TCP hole punching were the latter one comes with more restrictions as discussed later [FSK05; GTF04; Big+05].

UDP HOLE PUNCHING UDP hole punching, when successful, establishes UDP connectivity across NAT devices. Consider two clients, *A* and *B*, both potentially behind NAT devices, as illustrated in Figure 2.10. Initially, clients only know their local IP addresses. To become reachable by others, both clients register with a publicly accessible rendezvous server *S* located on the Internet. The registration message includes information about their local endpoints, causing any present NAT to create a mapping and filter that permits *S* to respond. Additionally, *S* records the mapped endpoints from which the regis-

tration messages are received. If client A now wants to connect to client B, the following steps occur [BRF23a]:

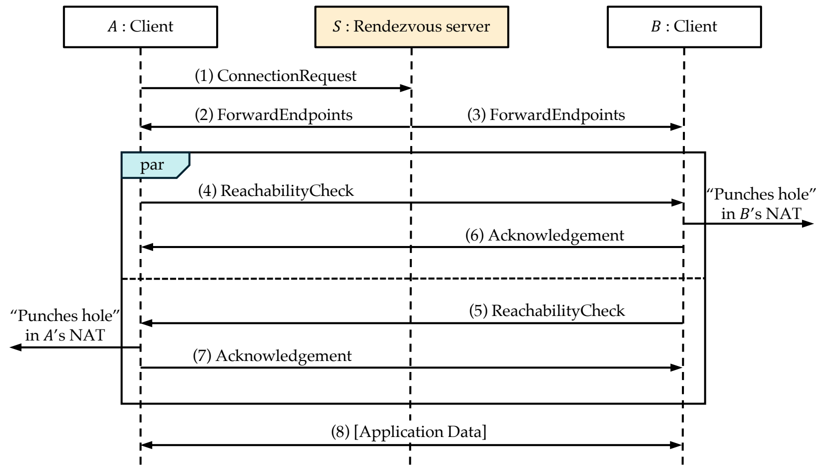


Figure 2.10: The UDP hole punching process. The NAT devices are not included for space reasons. However, any communication from either client must always pass through an existing NAT device first. [BRF23a]

(1) As A is unaware of how to route to B, it has to request connection information at S using a `ConnectionRequest`.

(2) Because of the preceding registrations, S knows a list of possible endpoints to reach B. Therefore, S will send this list to A by using a `ForwardEndpoints` message.

(3) At the same time, S will also send B all it knows about how to reach A with another `ForwardEndpoints` message.

(4) Once this information is received, A will try to reach B through all received endpoints. Implicitly, A's `ReachabilityCheck` will "punch a hole" into any present NAT device between A and the Internet, allowing it to be contacted back by B.

(5) B will do the same in parallel once it has received A's endpoints from S. B's `ReachabilityCheck` will also implicitly punch a hole into any present NAT device of B.

(6) A will wait for any `ReachabilityCheck` from B and will reply with `Acknowledgment` when received.

(7) In parallel, B will behave identically for any received `ReachabilityCheck` from A.

(8) Each client will "lock in" to the endpoint for which it receives an `Acknowledgment` first. As soon as both clients have locked in, bidirectional communication is possible through the locked endpoints. The hole punching steps are concluded with this, and any other UDP-based protocol (e.g., HTTP/3) can take over. However, in case of a connection loss, the hole punching process must be repeated.

TCP HOLE PUNCHING Establishing P2P TCP connections between hosts behind P2Ps is slightly more complex than for UDP. A standard TCP connection, initiated by sending a `SYN` packet, requires one peer to listen for this initiation. This approach only works when one peer is behind a NAT and initiates the connection. In hole punching scenarios, where both peers are behind NATs, the listening peer cannot receive the `SYN` packet because it is dropped by the listener's NAT. Different techniques are available for TCP hole punching, which all impose some smaller or larger drawbacks that result in TCP not being as reliable as UDP in many environments [GF05].

- TCP SIMULTANEOUS OPEN is a connection establishment procedure where both peers initiate a connection by sending `SYN` packets simultaneously, followed by `ACK` packets from both sides to complete the handshake [Edd22]. This method would reliably facilitate hole punching if the NAT devices involved used endpoint-independent mapping and if neither NAT responded with a *FIN* packet when inbound packets are discarded, which would prematurely close the connection. However, many platforms do not support TCP simultaneous open, limiting its applicability in practice.

- USER SPACE TCP IMPLEMENTATIONS allow applications to apply non-standard TCP behavior, which is mandatory by some TCP hole punching methods (like ignoring `FIN`s send by a NAT). While TCP implementation is normally provided by the platform, a raw socket can be used for a user space implementation. However, access to raw sockets generally requires elevated privileges, which may not be available in all environments due to security and administrative restrictions.

- TTL MANIPULATION sends outbound packets with a TTL value that is just high enough to pass the NAT. This action creates a mapping but prevents the peer's NAT from sending a `FIN` segment that could prematurely terminate the connection. After sending the low TTL `SYN` packet, the private host closes the socket and reopens another socket on the same local port, ready to listen for

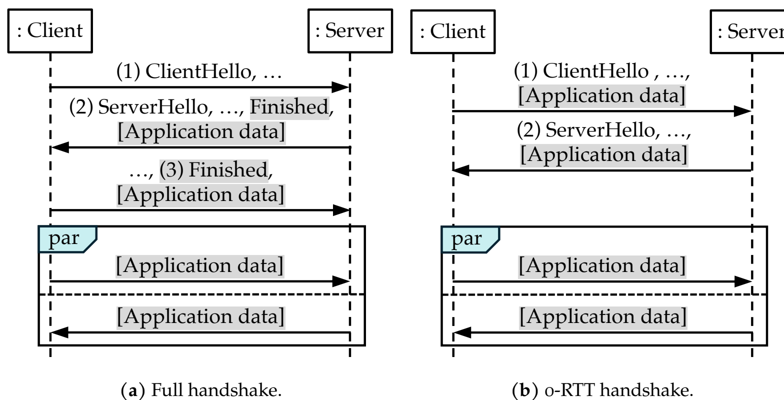
incoming connections. Now the other host can initiate a TCP connection. While this approach does not need a custom TCP implementation, setting a custom TTL value is also a privileged action. Finding the right TTL might also be challenging in complex NAT environments [GTF04].

Having explored the intricacies of NAT and its influence on network communication, the foundational aspects of TLS are now the focus. TLS is used to secure data transmission over networks, ensuring that communication between overlay network nodes remains confidential and authentic, especially in untrusted environments.

2.5 TLS FUNDAMENTALS

TLS is a cryptographic protocol designed to provide secure communication over a computer network [Res18]. It is widely used to ensure privacy, integrity, and authenticity between two communicating applications, such as a web browser and a server. TLS encrypts the data transmitted between the client and server, thus protecting it from eavesdropping and tampering.

The protocol begins with a handshake process, during which the client and server negotiate key parameters for the secure session, including the TLS version, encryption cipher suites, and session keys. The handshake is initiated by one peer (the client) and then confirmed by the other (server). During this handshake, the server presents its digital certificate, which the client verifies to ensure the server's identity. Once the handshake is completed, a secure, encrypted communication channel is established using symmetric encryption for efficiency. Public key cryptography is used during the handshake to exchange the encryption keys securely.



(a) Full handshake.

(b) 0-RTT handshake.

Figure 2.11: TLS handshake modes with and without reuse of previously received information, highlighting encrypted data. [BRF23a].

Starting with version 1.3, TLS provides two handshake modes relevant to distinguish [Res18]:

2.5.1 *Full handshake*

The full 1-RTT handshake (Figure 2.11a) is used for the initial connection establishment when no previous cryptographic information exists between client and server.

(1) The client initiates the handshake by sending a `ClientHello` message to the server. This message contains the client's cryptographic capabilities, including information like the supported cipher suites, TLS versions, and any extensions.

(2) The server responds with a `ServerHello` message, which includes the server's choice of cipher suite and the cryptographic parameters for the session. The server may also send its certificate or other optional messages required for authentication and key exchange. The server then completes its part of the handshake with a `Finished` message. At this point, the server can optionally start sending encrypted application data.

(3) After receiving the server's messages, the client responds with a `Finished` message, confirming that it has received the server's information and that the handshake is complete from its side as well. Once this is done, the client can also start sending encrypted application data.

2.5.2 *0-RTT handshake*

The 0-RTT handshake (Figure 2.11b) allows the client to send data to the server immediately without waiting for the full handshake to complete. This is possible when the client and server have communicated previously, and the client has stored session information from an earlier connection.

(1) The client initiates the handshake by sending a `ClientHello` message indicating that the client wants to initiate a 0-RTT handshake. Following this, the client sends encrypted application data based on previously established session information, assuming the data to be processed by the server.

(2) After receiving the `ClientHello` and application data, the server checks session information and proceeds to decrypt and process the data if it's

valid. The server's response includes a `ServerHello` to confirm session resumption. Further, the server can send encrypted application data immediately afterwards.

2.6 THE TASKLET SYSTEM

The Tasklet edge computing system [Sch+16] shown in Figure 2.12 consists of three main components: resource providers, resource consumers, and resource brokers.

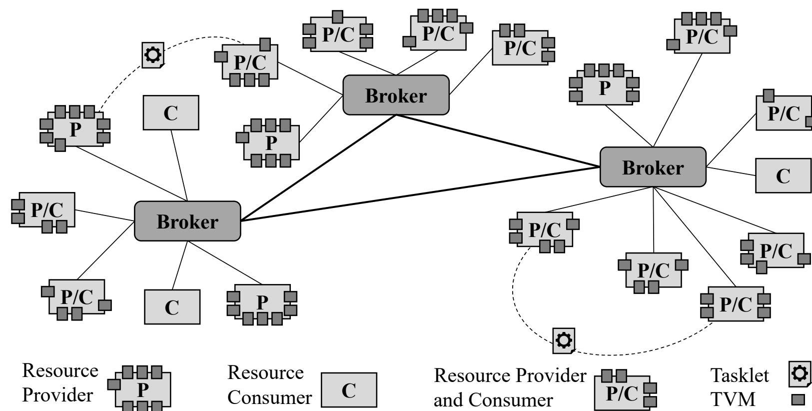


Figure 2.12: The Tasklet system. Computation resources are offered by resource providers (P) in the form of virtual machines (TVMs) to resource consumers (C). Scheduling is performed by the peer-to-peer broker overlay. Tasklets and results are exchanged directly between providers and consumers.

- **Resource providers** are participants in the Tasklet system who offer their local computing resources to support the execution of Tasklets from other participants. Tasklets are self-contained computation units, typically at the granularity of function calls, that can be executed on remote resources. These providers make their processing power, memory, and other computational resources available for use by resource consumers, facilitating the offloading of tasks that require additional capacity.
- **Resource consumers** are participants that offload local computations by utilizing Tasklets. This offloading occurs for various reasons, such as when local resources are insufficient or exhausted, or when faster computation is desired to achieve quicker results. By leveraging external computational resources, consumers can optimize performance and manage resource constraints more effectively.
- The **broker** is essential for the Tasklet system, facilitating the matchmaking process between resource consumers and providers.

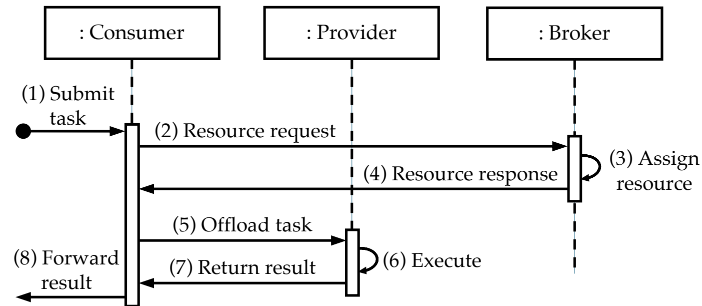


Figure 2.13: The Tasklet life cycle [Bor+23].

After the various components of the Tasklets system have been introduced, the life cycle of a Tasklet [Bor+23], as shown in Figure 2.13, is now presented:

(1) The application requiring additional computing power packages the local computation into a Tasklet and submits it to the locally running Tasklets system, which processes the "Submit task" request.

(2) The locally running Tasklets system sends a "Resource request" to the broker to obtain the necessary computing resources.

(3) The broker selects the most suitable provider for this request and performs the "Assign resource" operation.

(4) The broker sends a "Resource response" back to the consumer, informing it of the selected provider.

(5) The consumer "Offloads task" by forwarding the Tasklet directly to the chosen provider for execution.

(6) The provider "Execute" the Tasklet locally using its available resources.

(7) After execution, the provider performs a "Return result" operation of the Tasklet to the consumer.

(8) The local Tasklet system performs a "Forward result" operation to the application, completing the Tasklet's life cycle.

It is necessary for communication between providers, consumers, and the broker so that all parties can reach each other. However, this is not always the case in edge environments, which would prevent the Tasklet system from easily utilizing such resources. The existing application's limitation is overcome by using middleware, which provides the necessary routability between all participants.

2.7 CHAPTER SUMMARY

This chapter presented various topics that all contributed to making the underlying edge network infrastructure and its hosts more accessible and manageable. **SDN** is a networking management approach that enables centralized network control through a unified interface and configuration language. **IBN** is built on many SDN concepts and provides network configuration through a declarative abstraction that describes what is desired from the network rather than how it is achieved. Some of the concepts of SDN and IBN have inspired the system proposed by this thesis. **Overlay networks** help abstract from the limitations of underlying edge resources, providing an idealized view and control of available resources while transparently hiding corresponding complexity from the application. **NAT** is a widely deployed technology on the Internet, introducing indirections in host communication. This imposes restrictions on individual devices and methods for traversal, which must be considered when constructing overlay networks of edge devices, as up to 92 % of Internet hosts are affected by NAT [Haa+16; WP17]. **TLS** is a proven protocol providing secure communication between edge devices. The **Tasklet system** is a computation-offloading system that can transform computations into self-contained units that can be sent to other computers.

3 Requirements analysis

This chapter introduces requirements for a software-defined overlay networking system. These resulting requirements are used in assessing prior work and that guided the development of a software-defined overlay networking system, which is presented later in this manuscript. The chapter consists of the following sections:

- *Section 3.1 defines a scenario to identify all stakeholders of a software-defined overlay networking system.*
- *Section 3.2 describes each software-defined overlay networking system stakeholder's needs.*
- *Section 3.3 and Section 3.4 specify both the functional requirements and non-functional requirements of a software-defined overlay networking system.*
- *Section 3.5 summarizes the chapter.*

3.1 Software-defined overlay networking scenario

A scenario is created to identify the stakeholders involved in an software-defined overlay networking (SDON) system used by a distributed application. Scenarios serve as informal descriptions of a system's usage, offering a high-level overview of its functionality. They typically outline the system's operation, different user groups, and potential exceptional situations [LK22]. Further, the interactions between the stakeholders and the system are explored. Figure 3.1 illustrates the scenario, where the stakeholders and their interactions are as follows:

The *application developer* has specific requirements for the overlay network needed by its distributed application. The developer used the application programming interface (API), provided by the SDON system, to express the overlay network requirements. Thereby, the developer can use overlay network-related functionalities of the SDON system, and thus do not need to be implemented by the application. This allows the developer to focus on the development of other functionalities of the application.

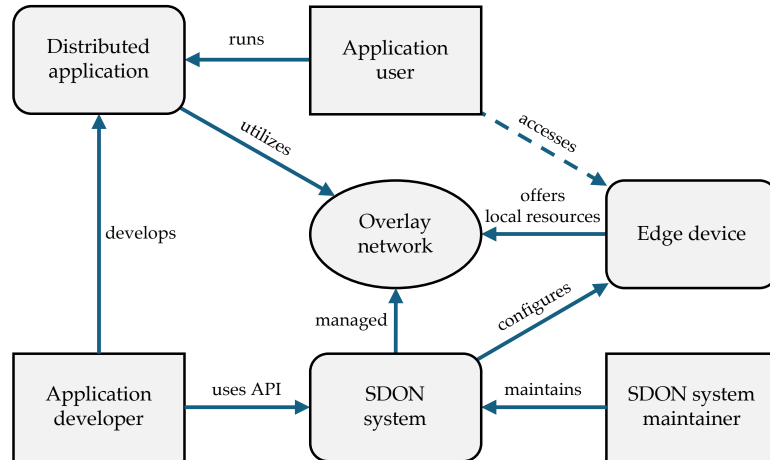


Figure 3.1: Stakeholders and their interactions in a software-defined overlay networking (SDON) system.

The *distributed application* is a standalone software that relies on a network to enable communication between its distributed components. While the application is used like any other software, it transparently utilizes the overlay network managed by the SDON system. Further, the application may specify new intents during runtime.

The *SDON system* manages the overlay network desired by the distributed application by configuring the edge devices that form the overlay.

The *overlay network* provides an idealized view and access to the underlying resources. It offers services to the application that are not provided by the underlay. The complexities of managing and utilizing the overlay network are transparently hidden from the application and the user.

The *application user* runs the application like any other software, while the application transparently uses the overlay network to provide the desired functionalities. This allows the user to seamlessly access resources from various edge devices without needing to understand how these resources are managed or offered.

An *edge device* is any computer system that is willing to offer its local resources to a SDON system. This could be dedicated devices waiting to be configured by the SDON system or, in the case of a P2P application, the computer system currently running the distributed application.

Lastly, the *SDON system maintainer* maintains the implementation of the SDON system. It ensures the ongoing health by fixing reported bugs and addressing feature requests from users and developers.

3.2 STAKEHOLDERS

In the subsequent section, each stakeholder is described by outlining their interests and responsibilities [LK22] as well as tangible benefits [Mac07].

APPLICATION DEVELOPER This stakeholder is responsible for integrating the SDON system into the distributed application.

The developer expects the SDON system to manage any overlay network-related tasks. Therefore, possible tasks are, e.g., communication protocol design, peer discovery, overlay network implementation and operation, resource management, and security. By delegating these tasks to the SDON system, the developer expects a streamlined development process and transparent overlay network management during runtime. The development process itself must also be supported by effective testing and debugging options.

Additionally, the developer expects the SDON system to use available device resources as efficiently as custom overlay network management in both small- and large-scale systems. Overall resource usage should be maximized, resulting in a minimized cost of running the application. The system is required to not only use the optimal resources but also include suboptimal ones, as long as they are still adequate, in order to distribute the workload more evenly across all available resources.

Distributed application may have requirements regarding additional overlay services, like custom routing, forwarding, quality of service (QoS), communication encryption, and other mechanisms that help to improve the use of unreliable and heterogeneous edge resources.

APPLICATION USER The application user runs the application and expects seamless functionality, whereas the overlay network runs transparently, ensuring that the distributed application behaves as a unified system. Additionally, the user expects the system to provide high reliability, with minimal performance degradation, even in the face of communication failures, resource churn, or malicious resources. The distributed application should maintain seamless functionality.

Furthermore, in a P2P application scenario, the user accesses and offers resources simultaneously. In this case, the user should also be able to stop resourcing sharing at any time, such as by exiting the application.

The application may involve transmitting sensitive data, which raises security and privacy concerns. Therefore, the SDON system should support the integration of appropriate safeguards.

SDON SYSTEM MAINTAINER The SDON system maintainer aims to support developers in creating performant distributed applications and ensure reliable operation. To achieve this, the maintainer should deliver an easy-to-use, well-defined, and documented API. As applications' requirements vary, the system should be extensible to match the wide range of possible use cases. The maintainer is also responsible for ensuring the system's longevity by fixing reported bugs and adding requested features without unnecessarily disrupting its overall architecture.

In summary, the needs and concerns of all key participants involved in a SDON system are discussed, providing a comprehensive understanding of the overall system. Some of the mentioned interests conflict with each other, such as the application user's need for the best possible user experience and SDON system's inclusion of suboptimal resources. Therefore, the SDON system must consider these conflicting interests and achieve a balanced trade-off between these needs. The insights gained in this section assist in defining the requirements that a SDON system must meet, which are addressed in the following sections.

3.3 FUNCTIONAL REQUIREMENTS

After identifying all relevant stakeholders, in this section, the functional requirements (FRs) are extracted to specify *what* a SDON system must be able to do.

FR 1: DYNAMIC OVERLAY NETWORK PROGRAMMING The primary function of the SDON system is to equip distributed applications with an overlay network that aligns with the application requirements. No restrictions should be imposed on the type of overlay networks, especially regarding topology, routing behavior, and optimization [LHM10]. Given that both the application's requirements and the environment in which the overlay network operates can change during runtime, the system must be capable of dynamically adapting the overlay network to meet evolving constraints.

FR 2: SELF-CONFIGURATION The SDON system is intended to free the developer from tasks related to the overlay network programming and management. Therefore, developers should only specify the overlay network's high-level goals. The whole system must be operational without external interventions by autonomously configuring itself [Fis12]. Further, the system should automatically devise a strategy based on known mechanisms to achieve those goals efficiently. Achieving consistent and coordinated actions across all components in overlay networking is essential. This ensures the system can quickly adapt to changing application requirements and edge resources. Such a level of coherence and efficiency is only possible when a unified, centralized control instance governs the configuration and behavior of all nodes.

FR 3: OVERLAY SERVICES Since application requirements vary, the overlay network required varies as well. Therefore, the overlay network should provide a wide range of services, helping the application offload functionalities to the overlay system and to make better use edge resources. This includes services such as custom routing, forwarding, QoS, or communication encryption. It must be possible to dynamically enable these services and restrict them to specific application components. Services should also be able to operate without interfering with each other. Finally, applications should be able to add their own overlay services to the system.

3.4 NONFUNCTIONAL REQUIREMENTS

In addition to the FRs, there are nonfunctional requirements (NFRs) that define *how* the system should behave.

NFR 1: EFFICIENCY The SDON system must be designed for high efficiency to meet *application users' application developers*', and *edge device*'s expectations.

For *application users*, the SDON system must use the overlay network optimally, ensuring that the application runs smoothly with e.g., minimal latency, fast response times, and optimal bandwidth use. The user should experience a seamless and uninterrupted service even under changing or restricted edge network conditions.

For *application developers*, the SDON system must provide APIs that allow the overlay network to be designed to maximize the use of underlying resources. This includes minimizing the overhead related to overlay network and edge device management. Additionally, in untrusted environments where distributed

applications commonly operate, the system must adapt edge device resources to improve utilization and align with the application's requirements.

For *edge devices*, the SDON system should ensure that the offered resources are optimally allocated and used. The system must minimize unnecessary resource consumption, such as reducing CPU, memory, and bandwidth usage. Additionally, it should dynamically adapt to changing workloads and network conditions, ensuring that resources are used in a cost-effective manner and aligned with both the application's requirements and the edge device's limitations [Mac07].

NFR 2: OVERLAY ROBUSTNESS The SDON system must feature a high degree of robustness to function correctly under stress conditions such as invalid inputs, resource unavailability, and communication failures [IEE90]. The system must maintain stable overlay networks, even amidst fluctuations in resource behavior and resource availability. This robustness ensures that consistent, "acceptable" behavior is preserved, successfully managing adverse or unexpected conditions [FMP05]. In case of a failure, the system should provide self-healing mechanisms that automatically detect and restore the affected components back to a normal state. Furthermore, the attack surface for potential attacks from external or internal participants must be minimized as much as possible. This robustness is particularly important in edge environments, as edge devices and edge networks display varying levels of reliability and may leave the system at any time or change behavior due to network transitions. Therefore, all resources in these environments must be considered unreliable and untrusted by default.

NFR 3: SCALABILITY is crucial for the SDON system to accommodate a broad spectrum of application deployment sizes. A system is considered scalable if its performance grows in proportion to the size of resources added [Hil90]. The system must perform efficiently across both small and large-scale environments. It should be capable of horizontal scaling (scaling out) by dynamically incorporating additional edge devices and resources into the overlay network to distribute workloads effectively [ST23].

In addition to the scalability of the edge devices and resources available to the system, scalability must also be maintained concerning the size of the requested overlay network in terms of links and nodes. Furthermore, central components that could create bottlenecks or lead to single point of failure (SPOF) must be avoided [Fis12].

NFR 4: EXTENSIBILITY is essential for the SDON system to accommodate to various distributed applications. It must be designed to facilitate easy customization, allowing for the seamless addition or replacement of components without disrupting existing operations [ST23]. The system must specifically support extensibility concerning constructing and managing different types of overlay networks and the associated mechanisms. Additionally, the system must be extensible to include measures that optimize resource utilization in edge environments and integration of a wide range of edge devices and network types.

NFR 5: SECURITY To operate securely across potentially untrusted environments, the SDON system must address key security goals, such as *confidentiality*, *integrity*, and *availability* [LK22].

Confidentiality prevents unauthorized access to transmitted data and can be achieved using techniques such as end-to-end encryption. End-to-end security is essential even when communication is relayed through third parties, ensuring that intermediaries cannot access the data. The encryption of the data must rely on session keys, ensuring that even a future compromise of long-term secrets (e.g., the private key) does not allow the reconstruction of a session key, thus achieving PFS [Roß11].

Integrity guarantees that data transmitted over the overlay network remains unaltered and protected from unauthorized modifications by third parties. Furthermore, it must be possible to reliably associate the data with a specific sender, even in indirect communication scenarios where end-to-end security must be maintained [Roß11].

Availability ensures that a system, its data, and functionalities are accessible to users whenever required. It also requires that resource access remains uninterrupted, even during failures, attacks, or high load, ensuring continuous operation and minimizing downtime.

3.5 CHAPTER SUMMARY

This chapter identified FRs and NFRs for a SDON system. A requirements analysis was conducted using a scenario to identify stakeholders, including application developers, application users, and SDON system maintainers, along with their needs. Based on these needs, requirements were identified, such as dynamic overlay network programming (FR 1), self-configuration (FR 2),

dynamic overlay services (FR 3), efficiency (NFR 1), overlay robustness (NFR 2), scalability (NFR 3), extensibility (NFR 4), and security (NFR 5).

4

Related work

In this chapter, related work relevant to this thesis is reviewed, focusing on approaches that facilitate the creation of distributed applications through overlay networking technologies. The chapter is structured as follows:

- *Section 4.1 establishes a classification for the literature analysis and presents related work following this classification.*
- *Section 4.7 evaluates prior work against the requirements of a software-defined overlay networking system identified in Chapter 3 to highlight gaps in prior work.*

4.1 CLASSIFICATION

The creation process of a distributed application can be divided into several life cycle phases (Figure 4.1). Related work is classified into five areas corresponding to the following phases [RF20]:

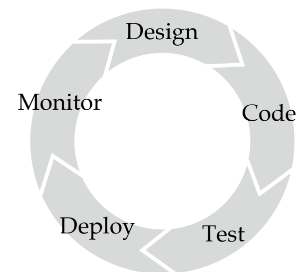


Figure 4.1: Overlay network development life cycle.

1. **Design**: Defines the architecture and component interaction based on the requirements. Focuses on aligning technical design with business goals, ensuring robustness, scalability, and maintainability (Section 4.2).
2. **Code**: Implements the design through coding. Produces reliable, scalable, and efficient code to meet requirements (Section 4.3).
3. **Test**: Verifies component interaction and checks for correct functioning. Involves writing and conducting tests to ensure all requirements are met (Section 4.4).
4. **Deploy**: Installs and configures the application across hosts. Ensures a smooth transition to production (Section 4.5).
5. **Monitor**: Monitors performance and reliability. Tracks metrics, detects issues and ensures stability by scaling or migrating components as needed (Section 4.6).

After categorization, this chapter presents the relevant work. The literature review includes approaches with varying scopes. Some works are extensive, addressing multiple challenges in developing distributed applications using overlay networking technologies, while others focus on solving specific aspects. The review period spans over three decades and covers several areas of distributed systems development.

4.2 DESIGN METHODOLOGIES FOR OVERLAY NETWORKS

The *design phase* of a distributed application overlay network involves translating the requirements into a detailed architecture that defines how the system's components interact. This phase is critical for aligning the technical aspects of the application with business objectives and performance goals. During this phase, architects and developers decide on the software's structure, technology stack, and data flows. The primary goal is to ensure the system is robust, maintainable, and scalable. [RF20].

In this context, works are identified and classified into overlay network taxonomies (Section 4.2.1) and declarative application modeling (Section 4.2.2).

4.2.1 *Decision-support through overlay network taxonomies*

Survey papers provide taxonomies that offer an overview and comparison of existing overlay network schemes, guiding the selection of an appropriate overlay network for specific application use cases. Notable work includes [Lua+05; AS04; KS10; Pas12; RM06], which each compare popular overlay networks like the structured overlay networks CAN, Chord, and Pastry [Rat+01; Sto+03; RD01] as well as the unstructured overlay networks Hyphanet (formerly known as Freenet), Gnutella, and FastTrack [Cla+01; KR02; KR04]. The papers compare the overlay networks based on different functionalities and performance metrics. The following aspects are identified as often being considered to assist in the selection of the correct overlay network scheme:

- Assess the overlay network system's *decentralization degree*.
- Describe the *operational architecture* of the overlay network system.
- Specify the overlay's *lookup query protocol performance* in terms of message complexity.
- Identify the required overlay network *system parameters* for operation.

- Evaluate the lookup *routing protocol performance* in the overlay.
- Analyze the *routing state* and scalability of the overlay network system.
- Describe the behavior of *node churn* and related self-organization.
- Look into the overlay network *security*.
- Examine the *robustness* and fault resiliency of the overlay network system.
- Review the *fairness* of resource allocation to assess the balance of responsibilities and dependencies among all overlay network participants.

Based on this work, architects and developers get a comprehensive overview of overlay network techniques, allowing them to make an informed decision about choosing the best-suited existing one or creating a new overlay. However, the benefit of these surveys may be limited if they do not address aspects of interest relevant to the specific application, such as how well the overlay network topologies map onto the physical network infrastructure or their behavior in mixed mobile, wireless, and ad-hoc network environments. Furthermore, [Lua+05] mentions that an overlay's behavior in real-world networks like the Internet is often not considered.

4.2.2 *Declarative application modeling*

Other papers present methodologies for modeling distributed applications declaratively, where developers can specify *what* the solution should accomplish rather than *how* it should be achieved, contrasting with imperative programming. While some of the following approaches are limited to modeling the system, other models support features such as automatic source code generation or automatic service management during overlay network operation that can be used in later software development life cycle phases. This section covers rule-based application specification, embedding service descriptions in source code, domain-specific language (DSL) descriptions, and modeling overlay networks as distributed databases.

▶ RULE-BASED APPLICATION SPECIFICATION is applied by approaches like FogBrain [FB20] and MARIO [Bro+20]. FogBrain provides an approach where the application, the included services, and the service interconnections can be specified via declarative rules. In MARIO, management policies that define triggers *when* and *how* an application should undeploy, migrate, or replicate services to keep

the application state aligned to a desired goal can be specified. Both approaches have been implemented using the Prolog logic programming language. Prolog employs a declarative programming paradigm, meaning the logic of computation is articulated through relations, manifested as sets of facts and rules that can be queried to resolve problems via logical inference [CM12].

Snippet 4.1 shows an example of a VR application specified with FogBrain. Line 1 specifies the application with identifier `vrApp` and three services `videoStorage`, `sceneSelector`, and `vrDriver` using an `application/2` fact. In the next line, a `service/4` fact is used to describe the `videoStorage` service with software requirements `mySQL` and `ubuntu`, as well as hardware requirements of 16 GB of memory. Lines 3 to 4 describe the two other services, and in lines 5 to 8, the interaction requirements between all services using `s2s/4` facts are specified. For instance, line 5 specifies that communications directing from service `videoStorage` to service `sceneSelector` requires a latency of at most 150 ms and a bandwidth of at least 8 Mbit/s.

Snippet 4.1: Example VR application specification in FogBrain [FB20].

```
application(vrApp, [videoStorage, sceneSelector, vrDriver]).
service(videoStorage, [mySQL, ubuntu], 16, []).
service(sceneSelector, [ubuntu], 2, []).
service(vrDriver, [gcc, make], 2, [vrViewer]).
s2s(videoStorage, sceneSelector, 150, 16).
s2s(sceneSelector, videoStorage, 150, 0.5).
s2s(sceneSelector, vrDriver, 20, 8).
s2s(vrDriver, sceneSelector, 20, 1)
```

Such rule-based approaches present a declarative continuous reasoning framework to support runtime management decision-making. Both approaches do not support application topology changes like adding/removing services.

- EMBEDDING SERVICE DESCRIPTIONS into the source code is an approach that is for example used by the multi-agent platform Jadex [BP12] and the distributed computing framework Ray [Any17]. Service properties, dependencies and interfaces are specified declaratively with method. Such descriptions can be interpreted by a suitable runtime environment, as with Jadex. This runtime environment ensures that the service is executed according to its configuration and that all dependencies are resolved. Therefore, the application developer is freed from tasks such as manual dependency management, service deploy-

ment, resource allocation, and runtime configuration adjustments. This design principle, known as *inversion of control*, improves modularity and flexibility by decoupling the execution of components from their implementation. This allows systems to be more dynamic and adaptable.

```
16 @Description("This agent provides a basic chat service.")
17 @Agent
18 @ProvidedServices(@ProvidedService(type=IChatService.class,
19   implementation=@Implementation(ChatServiceD2.class)))
20 @RequiredServices({
21   @RequiredService(name="clockservice", type=IClockService.class),
22   @RequiredService(name="chatservices", type=IChatService.class, scope=ServiceScope.PLATFORM)
23 })
24 public class ChatD2Agent
25 {
26 }
```

Snippet 4.2: Service descriptions are directly embedded to the source code in Jadex [BP12].

Snippet 4.2 illustrates this design principle with the `ChatD2Agent` agent taken from the Jadex tutorial. In Jadex, all services are part of an agent, which can be – like in this case – an empty Java class for a simple agent. The service descriptions are formulated by using Java annotations. The `@ProvidedServices` annotation in line 18 informs the Jadex runtime that this agent offers a service of type `IChatService` and that this service should be instantiated using the corresponding implementation `ChatServiceD2`. Conversely, the `@RequiredServices` annotation specifies the services required by the agent for execution, in this case, services of types `IClockService` and `IChatService`. The attribute `scope` in line 22 is used to specify the scope in which a required service should be looked up for. It is worth noting that in Jadex, a distributed application is organized into components and sub-components corresponding to this scope. While the `IClockService` is only searched within the current or immediate sub-components, the `IChatService` is searched across all components of the Jadex platform.

This design paradigm offers advantages in terms of application modularity and testability. However, in exchange, the developer loses some control over the application as the framework supervises the runtime environment. This can lead to increased dependency on the particular framework and restricted expandability and debugging.. These are aspects that engineers and developers of a distributed application should consider beforehand.

- DSL-BASED OVERLAY NETWORK BEHAVIOR MODELING. Approaches such as MACEDON, Mace, SEDA, and by Lopes [Rod+04; Kil+07; WCB01; Lop97] introduce DSLs where tailored language syntax and semantics simplifies the overlay network description, reducing errors, and increasing development efficiency. Further, such overlay-generic application programming interface (API) increases the interoperability between different overlay networks and applications, simplifying consistent experimental evaluation.

  The mentioned approaches have in common that they all implement overlay node's behavior using an event-driven finite state machine (FSM). Here, FSMs typically consist of *states*, *events* (triggered by actions like message reception or peer failures), *transitions* that define responses to these events, and *timers* for scheduling future processes.

  Approaches like Mace and Macedon support the generation of source code based on these high-level specifications. The generated code can be used unmodified in Internet settings, as all necessary functionalities, such as thread and timer management, network communication, and state serialization, are automatically provided. The generated source is API-consistent, allowing seamless integration into the application and ensure that overlay network changes do not require modifications elsewhere in the system.

  While DSL-based development can significantly reduce development effort, it may come with the cost of potentially reduced overlay network performance due to the required overlay network abstraction layer (the FSM). Additionally, the authors of MACEDON note that some overlay networks may not be expressible using a FSM. Nonetheless, such approaches are well-suited for rapid prototyping and evaluating various overlay network schemes.

- MODELING OVERLAY NETWORKS AS DISTRIBUTED DATABASES, as introduced by approaches like OverLog and Behnel et al. [Loo+05; BB05], provide a method for designing overlay network topologies as database management systems (DBMSs). This approach treats each node's knowledge about itself and its directly connected overlay network neighbors as a partial view of a distributed database containing all node information.

  Behnel et al. design overlay networks using a SQL-like declarative languages. Based on stored queries, a management system makes decision about relevant neighbors, attributes of interest, and messages transmitted. This approach makes it necessary, that all information will eventually reach all nodes. Behnel et al. suggest various methods, including broadcast, active lookup, overhearing routed messages, central discovery services, and epidemic communication.

```
1 CREATE VIEW chord_fingertable AS
2 SELECT   node.id
3 FROM     node_db
4 WITH     log_k = log(𝒦)
5 WHERE    node.supports_chord = TRUE AND node.alive = TRUE
6 HAVING   ring_dist(local.id, node.id) IN (2^i, 2^(i+1))
7 FOREACH  i IN [0, log_k)
8 RANKED   highest(1, ring_dist(local.id, node.id))
```

Snippet 4.3: Implementation of the Chord graph [Sto+03] in SQL-like overlay network specification language SLOSL [Beh07].

Snippet 4.3 demonstrates how a Chord graph can be formulated using the SQL-like overlay specification language (SLOSL) [BB07]. Chord is a DHT organizing nodes in a circular identifier space, with each node maintaining links to its successor, predecessor, and "fingers" – peers at exponentially increasing intervals for efficient routing [Sto+03]. In SLOSL, the clauses `CREATE VIEW`, `SELECT`, `FROM`, and `WHERE` function similarly to their counterparts in SQL. The `CREATE VIEW` clause initiates a node set view named `chord_fingertable`. Lines 1 to 2 utilize the `SELECT` and `FROM` clauses to define the view's attributes and data sources, specifically showcasing the `node.id` attribute and integrating it as a sub-view of the local node table, `node_db`. The `WITH` clause sets variables or options, notably by assigning `log_k` a default value representing the logarithm of the DHT key space size $\mathcal{K}$. Line 5 's `WHERE` clause filters nodes into the view based on their support for the Chord protocol and their status as "alive", excluding those considered dead by the local node. The `FOREACH` clause outlines a set of buckets corresponding to individual chord fingers, with the `HAVING` expression allocating nodes to these buckets. Line 8 then selects the highest node from each bucket to finalize the view setup.

Designing overlay networks as DBMSs allows developers to focus solely on the view configuration rather than the underlying overlay network protocol, effectively decoupling the design process. Behnel et al. acknowledge that while performance may not match that of a direct implementation, the benefits of rapid replication and flexibility justify its use. Further, manual configuration is still necessary to select appropriate algorithms, such as a gossip protocol, to maintain the currency and accuracy of the views. While the approach explained how a node locally decides what information is relevant to create and maintain its local view of the overlay networks, the efficient propagation of this information was out of scope.

4.3 CODING-SUPPORT FOR OVERLAY NETWORKS

The *code phase* in the life cycle of a distributed application overlay network covers implementing the overlay's functionality. During this phase, developers write code to realize the design specified in the previous phase. The primary goal of this phase is to produce reliable, scalable, and efficient code that meets the specified requirements. [RF20].

Two approaches are identified that support this life cycle phase in this context: Providing building blocks for generic overlay network functionalities (Section 4.3.1) and utilizing concurrent programming paradigms (Section 4.3.2).

4.3.1 *Distributed application programming environments*

Distributed application programming environments facilitate the development of distributed applications with overlay networks by providing adaptable building blocks designed to address common challenges in distributed systems. For example, JXTA [Gon01] and libp2p [Pro20] are software libraries that facilitate P2P communication and simplify distributed application development.

▸ THE JXTA P2P FRAMEWORK, originally developed by Sun Microsystems, includes a suite of six protocols and components utilizing these protocols to provide often-needed P2P functionalities. Within this framework, JXTA features two fundamental protocols designated as core protocols, vital for basic message routing and query resolution, while the remaining four protocols build on these two core protocols to provide higher-level functionalities [Tra+02]:

- **Endpoint routing protocol:** Facilitates route discovery for message transmission between peers, particularly through intermediary relays when direct routes are unavailable.
- **Rendezvous protocol:** Manages message propagation across peer groups by allowing peers to subscribe to and propagate messages within a peer group's rendezvous service.
- **Peer discovery protocol:** Enables peers to publish and discover advertisements related to peers, peer groups, modules, pipes, and content using resolver queries.
- **Peer information protocol:** Supports the retrieval of status information about peers, including status, uptime and capabilities, using resolver queries for propagation.

- **Pipe binding protocol:** Establishes virtual communication channels, or pipes, between peers and manages the binding of these pipes to physical peer endpoints. These pipes provide quality of service (QoS) guarantees and are used for application traffic.

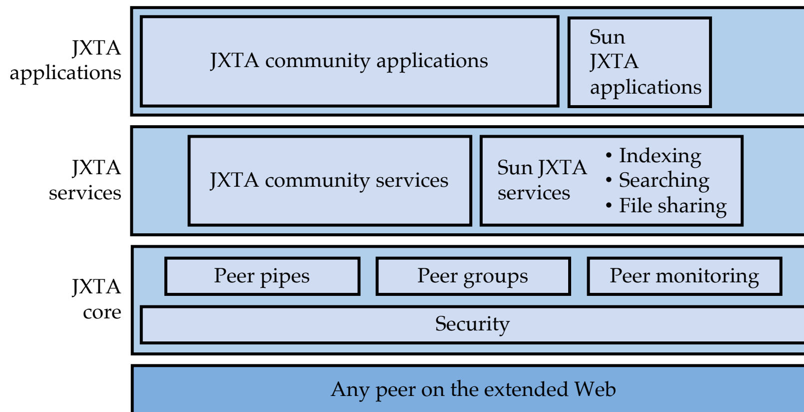


Figure 4.2: JXTA architecture.

The JXTA architecture displayed in Figure 4.2 is structured into three main layers: core, services, and applications. The core layer provides the fundamental protocols and mechanisms for P2P communication, including peer discovery, peer communication, peer group management, and security. Building on these core protocols, the services layer offers higher-level functionalities such as indexing, searching, and file sharing, facilitating more complex interactions and functionalities required by distributed applications. Finally, the applications layer consists of the end-user applications that leverage the core and services layers to perform specific tasks, providing user-facing functionalities built on top of the core and service components. Sun Microsystems envisioned a community developing over time to provide services and applications for the JXTA ecosystem.

▶ LIBP2P originated as the networking protocol for the InterPlanetary File System (IPFS) project [Ben14] and has since been abstracted into a standalone modular networking stack utilized across various projects for P2P communication. In contrast to JXTA, libp2p is aimed more at interoperability across different programming languages and versions, thereby providing developers with the flexibility to integrate P2P capabilities into diverse software environments.

The architecture of libp2p is designed to be modular, allowing developers to tailor the overlay network stack by selecting specific components that meet

their applications' requirements. This modularity extends to the transport layer, accommodating a range of protocols to ensure operability under various network conditions, including challenging scenarios such as NAT traversal.

libp2p comprises a suite of language-agnostic protocol specifications that support fundamental P2P functionalities like peer discovery, data storage and retrieval, and secure communications. Security features include peer identity verification through public key cryptography and encrypted communications to safeguard data exchanges.

In conclusion, JXTA and libp2p offer libraries that accelerate distributed application development by providing directly usable and well-tested components. These components primarily address lower-level functionalities, which require developers to engage in significant configuration. Integration efforts to achieve higher-level functionalities are tailored to specific requirements and can therefore introduce complexity.

4.3.2 *Concurrent programming paradigms*

While the approaches in the previous chapter are software libraries integrated into the application, this chapter introduces frameworks that utilize concurrent programming paradigms like actor-based and agent-based programming. Here, an application is modeled as a collection of software actors or agents who interact with each other to achieve given goals.

The actor-based model [Agh86] is a computational paradigm where actors are the fundamental computation units. All actors are independent software components that do not share data and communicate through asynchronous message passing with non-deterministic message delivery. They operate autonomously with location-transparent communication, relying on thread-based local behavior and a mailbox that buffers incoming messages, serving as the universal identifier for the actor [CN22].

An agent is an goal-driven entity capable of learning and adaptation, while an actor focuses on concurrency and state isolation through asynchronous message passing [Fer99].

This programming model fosters scalable application development by simplifying concurrency management, ensuring inherent fault tolerance, enhancing scalability, providing location transparency, and facilitating easier debugging and maintenance by encapsulating state and behavior within actors. These features make it a powerful and effective alternative to traditional multi-threaded systems that rely on shared data and locks [Lee06].

The principles of these approaches for overlay network programming are now discussed, based on the work of actor-based programming platforms Akka, Parallel Theatre, and ActorEdge [Lig09; Nig21; AZ17] and agent-based programming platforms Jadex, JADE, and JaCoMo [BP12; BPR01; Boi+13]. These approaches provide runtime platforms to host multi-actor/multi-agent applications running locally or distributed multiple systems connected through a network.

The following section contains further description of Parallel Theatre, which offers the most comprehensive features for distributed application development, including automatic actor migration between remote platforms and dynamic load balancing. In Parallel Theatre, the actor runtime environments are called "theatres". Each theatre consists of four layers: (1) The application layer (AL) contains actors assigned locally for execution. (2) The control layer (CL) manages scheduling and dispatching messages among AL actors. (3) The transport layer (TL) facilitates remote interactions through communication networks, typically using TCP sockets. (4) The time server layer (TS) ensures a common global time notion within the system.

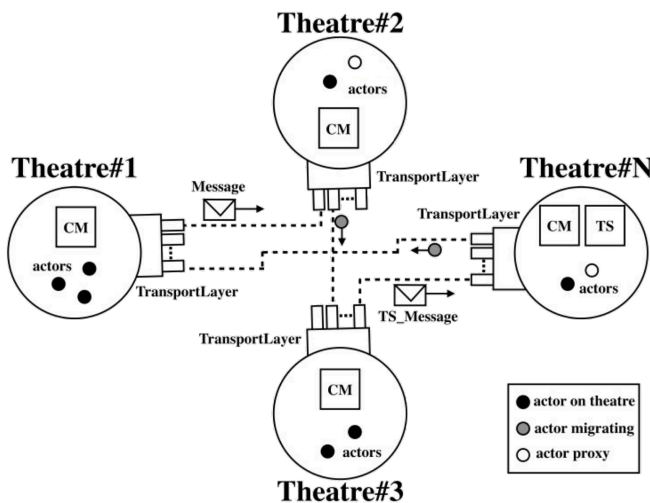


Figure 4.3: Distributed actor-based system consisting of N actor runtime environments called "theatres" [CNS20].

Figure 4.3 shows the relationships between theatres, the messages traveling through the network, and the actors within the system. Black circles (●) represent actors currently executing in a specific theatre. White circles (○) indicate actors that have moved away from their original theatre. Gray circles (◉) depict actors that are migrating to a new destination theatre.

As part of a load-balancing strategy, an actor can migrate between different theatres during its lifecycle. A proxy version is then created in the original

theatre when the actor migrates. This proxy is a placeholder and message forwarder, retaining the last known destination theatre. Messages directed to a proxy actor are transparently forwarded to the indicated remote theatre.

To summarize, concurrent programming paradigms like actor-based and agent-based programming offer an easy-to-use, lock-free, and concurrent computing model that supports the development of scalable distributed application overlay networks. However, these paradigms can introduce complexity in debugging and managing states across distributed systems, along with performance overhead, a steep learning curve for developers, and limited tooling.

4.4 TESTING FRAMEWORKS FOR OVERLAY NETWORKS

In the *test phase* of a distributed application overlay, it is verified that all components interact correctly. This phase is crucial for identifying defects and verifying all requirements. Tasks include writing and conducting automatic/-manual tests. [RF20].

Formal verification methodologies (Section 4.4.1) and simulation environments (Section 4.4.2) for overlay network systems have been identified to assist in this development phase.

4.4.1 *Formal verification of distributed systems*

Approaches like IronFleet, Verdi, or by Sergey et al. [Haw+15; Wil+15; SWT17] aim to verify the correctness of a distributed system formally. In theory, this verification methodology can potentially completely remove errors from distributed systems.

The formal verification methodology is examined by highlighting IronFleet, which outlines a methodology for constructing provably correct systems using a combination of formal state-machine refinement based on TLA[1] [Lam02] and formal program verification using Floyd-Hoare-style imperative verification [Hoa69]. This approach can be automated and machine-checked, proving the *safety* and *liveness* properties of distributed system implementations. The "safety" property states that something will do *not* happen, such as guaranteeing that a program started with correct input cannot terminate without producing correct output [Lam77]. The "liveness" property states that something *must* happen, such as guaranteeing that a program will terminate if its input is correct [Lam77]. IronFleet ensures that a distributed system implementation conforms to a high-level centralized specification (e.g., a distributed

1 *"The temporal logic of actions (TLA) is a logic for specifying and reasoning about concurrent systems."*—[Lam94]

key-value store behaves like a key-value store). Consequently, the approach can address specific concurrency issues, arithmetic errors, and protocol mismatches [Yua+14]. To achieve this, IronFleet decomposed the distributed system behavior into three layers corresponding to a high-level specifications layer, a distributed protocol layer, and an implementation layer, making the verification of practical distributed system implementations feasible, as illustrated in Figure 4.4. Each layer consists of a series of steps determined utilizing reduction [Lip75] based on a fine-grained behavior description of the application that must be supplied. Any behavior of a lower layer (e.g., `P0`, `P1`, ...) refines some behavior of the layer above (e.g., `H0`, `H1`, ...). The two states must satisfy the specification's refinement conditions for each refinement, as shown in a dashed arrow.

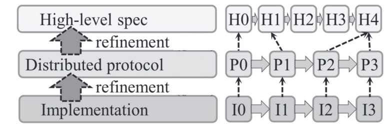


Figure 4.4: IronFleet divides a distributed system's behavior into layers of steps for verification [Haw+15].

Imperative verification is applied for the implementation layer running on each host, proving that the implementation correctly refines the protocol layer. For this, Dafny, an automatic program verifier for functional correctness [Lei10], is used as seen in Snippet 4.4. This capability automatically handles many low-level proofs, such as verifying the program for all possible inputs without assistance.

```
1 method halve(x:int) returns (y:int)
2     requires x > 0;
3     ensures y < x;
4 { y := x / 2; }
```

Snippet 4.4: Imperative verification example [Haw+15].

To summarize these formal approaches, verification complexity scalability relative to application size is an ongoing area of research. The authors of IronFleet note that exploring a large search space can be time-consuming. Additionally, extending guarantees, e.g., to include fairness properties beyond liveness is a topic for future work. "fairness" property extend "liveness" by ensuring that certain actions occur *infinitely* often within a system. A crucial requirement for formal verification is a fine-grained behavior description of the application.

4.4.2 *Simulation environments*

Simulation environments offer well-defined platforms for running and comparing different distributed application overlay networks. In this context, two simulation frameworks are highlighted: PlanetSim [Gar+05], which focuses on structured overlay networks, and OverSim [BHK07], which supports both structured and unstructured overlay networks. Both frameworks utilize a custom discrete event simulator (like OMNeT++ [Ope01] in the case of PlanetSim). A discrete event simulator models the operation of a system as a sequence of discrete events in time, where each event occurs at a specific point and changes the system's state [Rob14].

These simulators divide their simulation environments into three layers: the application layer, the overlay network layer, and the underlay layer. Each layer is separated by a well-defined API, allowing layers to be exchanged independently. This separation allows for the simulation of the same application with different overlay networks or underlay networks.

OverSim features a similarly flexible underlay layer that supports heterogeneous networks, enabling the configuration of bandwidth and latency properties. Additionally, it allows simulations to run on real networks. In contrast, PlanetSim offers a GUI for visualizing the overlay network topology at any time, aiding in the debugging of new overlay network protocols [Nai+06].

Before an overlay network can be executed within these frameworks, it must implement a given API. PlanetSim requires overlay networks to implement the Common API (CAPI) for structured overlay networks, as presented by [Dab+03], which offers an overlay-neutral interface based on key-based routing and consists of eight individual interfaces, as detailed in Table 4.1.

Once the overlay network is implemented and all underlying layers and experiment parameters (such as the initial state and desired overlay network size) are configured, the simulation can be started. The entire distributed system will be executed event by event. Simulators like OMNeT++ offer the flexibility to manually step through each event or run the simulation until a specified condition is met. Additionally, the environment can be configured to behave deterministically, facilitating easier debugging and comparison of results. OMNeT++ also provides an interactive simulation runtime GUI.

In conclusion, it is essential to implement the overlay network using the simulation framework's API first. When using CAPI, a widely adopted API for overlay network programming, the overlay network implementation will likely work directly in the simulator. However, the poor scalability of simulators often necessitates a significant amount of computing time [Law24]. Additionally, the usability of some simulators is a serious issue, as documentation is often inadequate [Nai+06].

4.5 STRATEGIES FOR OVERLAY NETWORK DEPLOYMENT

The *deployment phase* of a distributed application overlay network involves installing the application's components across multiple hosts. This phase ensures the application is correctly installed, configured, and operational in the target environment. This phase is crucial for transitioning from development to production. [RF20].

Interface	Description
`void route(key →K, msg M, nodehandle hint)`	Forwards message `M`, towards root of key `K`. Optional `hint` argument specifies the initial hop node for routing.
`void forward(key ↔K, msg ↔M, nodehandle ↔nextHopNode)`	Triggered at every node forwarding message `M`, notifies the application that `M` with key `K` is being forwarded to `nextHopNode`.
`void deliver(key →K, msg →M)`	Called on root node for key `K` when message `M` arrives.
`nodehandle[] local_lookup(key →K, int →num, bool →safe)`	Generates node list for next hops towards key `K`, maintains a safe fraction of non-faulty nodes if `safe` is `true`.
`nodehandle[] neighborSet(int →num)`	Returns up to `num` unordered nodehandles neighboring the local node.
`nodehandle[] replicaSet(key →k, int →max_rank)`	Returns ordered nodehandles for storing replicas of object `k` up to `max_rank`.
`void update(nodehandle →n, bool →joined)`	Informs application that node `n` joined or left neighbor set of the local node.
`bool range(nodehandle →N, rank →r, key ↔lkey, key ↔rkey)`	Returns ranges the node `N` is currently a `r`-root for. *true* if determinable, *false* otherwise.

readonly parameters are denoted as → `p`. read-write parameters as ↔ `p`. ordered set of type `T` as `T[]`.

Table 4.1: Common API for structured overlay networks [Dab+03].

As part of this effort, solutions have been identified that create more favorable environments for the deployment and operation of distributed applications. Overlay networks such as VPNs overcome some limitations of underlying physical networks by providing multicast, QoS support, DoS defense, and resilient routing [Sto+02; For04; KS10] (Section 4.5.1). Further, ISP-involved overlay network deployment models are presented (Section 4.5.2).

4.5.1 *Self-configurable mesh VPNs*

Mesh VPNs use a P2P network architecture, which improves resiliency, scalability, and performance compared to traditional VPNs that use a star topology. One approach is highlighted in which the control and data planes are decentralized, followed by two where the control plane is centralized.

- The Secure OverLay for IPsec Discovery (SOLID) [RSS10] approach is a self-configurable VPN based on IPsec, used to connect subnetworks. IPsec is a suite of protocols designed to ensure the confidentiality, integrity, and authenticity of data communications over IP [SK05]. Here, access to the overlay network is controlled via a central authority, which also assigns IP addresses to the individual participants. Apart from that one-time certification per gateway, SOLID requires no manual configuration. If a packet arrives at a gateway without knowing a route to the corresponding target gateway, it is cached, and the gateway sends a discovery request over the existing routes. A greedy search is used where each gateway forwards the message to the gateway closest to the destination address. The Chord [Sto+03] ring structure ensures the timely discovery of the correct gateway. Routing tables are further optimized by detecting public routable gateways and gateways within the same subnetworks that are candidates for direct paths. This overlay network for discovery and routing allows the system to achieve the necessary security, robustness, and scalability.

  Figure 4.5 illustrates the SOLID overlay network that is organized in a Chord ring [Sto+03]. Each subnetwork, capable of containing an arbitrary number of hosts, is connected to the overlay network via an IPsec gateway. Solid lines indicate proactively established IPsec tunnels, while dotted lines show tunnels that are routed through intermediate systems due to a nested configuration. Thick lines represent virtual connections between IPsec gateway processes on the same device, allowing multiple secured subnetworks to connect through a single IPsec gateway.

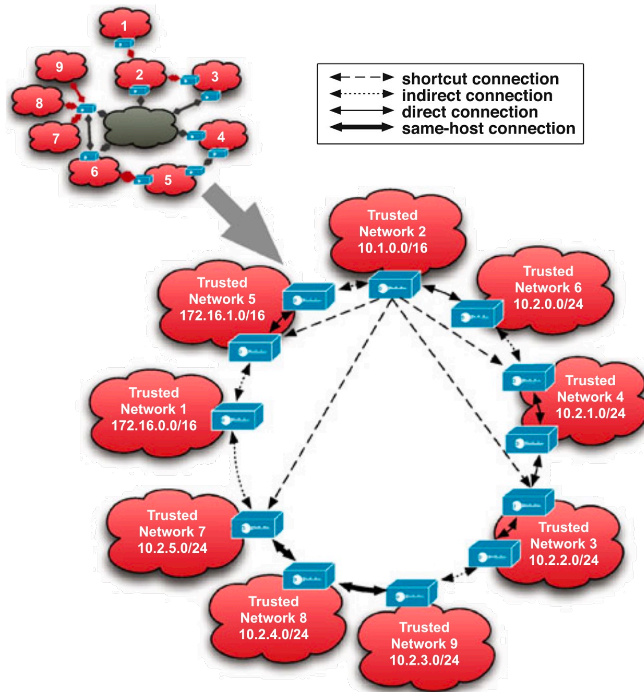


Figure 4.5: Mapping of a VPN topology to a SOLID overlay [RSS10].

- A CENTRALIZED CONFIGURATION APPROACH is presented by ZeroTier [Zer14] and Tailscale [Tai19]: These two providers offer proprietary services where they provide centralized infrastructure to semi-automatical customer VPNs. Via a web interface (or an API), the customer can configure their overlay network, such as routing tables and access rights. Subsequently, customers must install the provider's software on the respective hosts and associate it with the customer's token. The provider's software then automatically handles the construction and management of the overlays. Further, these approaches aim to optimize the underlay routes to minimize latency automatically.

  ZeroTier uses a two-layered overlay network. The first layer consists of a P2P network with encryption and NAT traversal, providing a fully routable flat network. On top of this, a second layer for the data plane, an Ethernet virtualization layer, is placed [Zer24a]. Further, ZeroTier provides a SDK [Zer24b] allowing to use their service as an application-level overlay. Tailscale relies on WireGuard for the data plane. WireGuard is a secure network tunnel protocol operating at layer 3, known for its lightweight design, performance, and use of modern cryptography [Don17]. In contrast, a custom protocol is used for the control plane, utilizing centralized coordination servers for rendezvous and configuration distribution [Tai20].

The presented work here helps deploy distributed applications by providing secure, fully-routable layer 3 overlays. Therefore, generic IP-based applications can be used without customization while relying on a more idealized network.

4.5.2 *ISP-involved overlay network deployment models*

[KS10] discusses overlay network deployment models with involvement from ISPs. While overlays usually do not impose requirements on the underlay, the approach presented here expects ISP participation to provide end-to-end QoS guarantees.

The deployment model proposed by [DZH03], aims to provide end-to-end QoS guarantees on an interdomain scale through an overlay network. This is achieved by purchasing necessary resources like bandwidth with specific QoS guarantees from underlying ISPs via bilateral service level agreements (SLAs). The authors motivate their approach because of the inherent structure of the Internet, which is a collection of autonomous systems owned by various administrative entities (like an ISP) that work on a best-effort quality model. They state that each administrative entity is primarily concerned only with the performance of its network, making it challenging to achieve end-to-end QoS across multiple such systems.

The overlay network uses service gateways that perform service-specific data forwarding and control functions. These gateways are placed between the individual networks, and the QoS guarantees between them are specified in bilateral SLAs. End-to-end QoS between networks is provided by these gateways, while within individual networks, it is managed by the respective administrative entity.

To conclude this approach, it must be stated that the primary challenge lies in efficiently purchasing resources from ISPs, which is a capital-intensive process and does not scale well with large and dynamic overlays due to the extensive number of required SLAs with numerous ISPs. However, there is the opportunity to achieve end-to-end QoS by ensuring that the underlying ISPs reserve the necessary resources according to the SLAs.

4.6 MONITORING AND MANAGING OVERLAY NETWORKS

The *monitor phase* of a distributed application overlay network involves continuously observing the application's performance, availability, and reliability. Key tasks include tracking system metrics, logging events, detecting anoma-

lies, and raising alerts. The goals are to ensure optimal performance, quickly identify and resolve issues such as node churn or link failures, and maintain the application's health and stability by migrating or scaling components to meet QoS goals. [RF20].

This section examines two contrasting approaches to application monitoring and management. The first approach involves a separate system that conducts monitoring and management, allowing the application to operate without being aware of these functions (Section 4.6.1). The second approach integrates monitoring and management directly into the application, enabling it to monitor itself and respond autonomously to failures, load changes, or evolving requirements (Section 4.6.2).

4.6.1 *Separate application orchestration*

This approach places the application orchestration into a separate system that is transparent to the application. This simplifies the application design by decoupling operational concerns, allowing developers to focus solely on application logic without the added complexity of incorporating monitoring and management functionality. It also enables specialized, dedicated systems to handle monitoring and management tasks, potentially leading to more robust and efficient performance. Additionally, this separation allows for greater flexibility, as the monitoring and management system can be independently updated, scaled, and optimized without requiring changes to the application itself. This modularity facilitates more manageable maintenance and upgrades and can enhance the overall reliability and scalability of the application infrastructure.

Kubernetes [Kub14] is an open-source orchestrator designed for deploying containerized applications. Initially developed by Google in 2014, it has since become the standard API for building cloud-native applications. Kubernetes is versatile and can be used on various scales, from small clusters of Raspberry Pi computers to large data centers filled with the latest hardware. [Bur+22]. While initially developed for cloud environments, several approaches have been proposed to adapt the principles of Kubernetes to IoT and edge environments as well [SAF23; Bar+23].

In Kubernetes, a distributed application must be packaged as containers, typically created with Docker [Doc13]. A container image is a binary package that includes all the files, libraries, and dependencies required to run a program. It also encapsulates the application's code and environment settings. The container image has well-defined interfaces, allowing the orchestration system

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-mariadb-pod
spec:
  containers:
  - name: nginx
    image: nginx:latest
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 30
      periodSeconds: 10
  - name: mariadb
    image: mariadb:latest
    resources:
      requests:
        memory: "128Mi"
        cpu: "500m"
      limits:
        memory: "256Mi"
        cpu: "1"
    livenessProbe:
      tcpSocket:
        port: 3306
      initialDelaySeconds: 30
      periodSeconds: 10
```

Snippet 4.5: Pod manifest for an Nginx and MariaDB pod with resource management and monitoring mechanisms.

to manage and run the containers without understanding the application's specifics or internal structure. This abstraction enables Kubernetes to efficiently handle application deployment, scaling, and management across a cluster of machines, ensuring consistency and portability across different environments.

To orchestrate an application, Kubernetes requires metadata regarding the resource requirements of each container, performance metrics, and appropriate responses to those metrics. Snippet 4.5 describes a simple pod example. Pods are the smallest deployable units of computing that Kubernetes can manage. Here, the pod consists of a nginx-based [Igo04] web server (lines 7 to 21) and a MariaDB-based [Mar09] database server (lines 22 to 35).

For resource management, Kubernetes allows the specification of minimal and maximum resources a single container utilizes. For each container, the `requests` sections (lines 10 and 25) specify the minimum amount of resources (memory and CPU) guaranteed to the container, while the `limits` sections (lines 13 and 28) define the maximum amount of resources it can use. If a container exceeds its CPU limits, it is throttled, while exceeding memory limits results in the container being terminated and restarted according to a specified restart policy. This mechanism helps maintain system stability and performance by controlling resource usage.

For monitoring, Kubernetes can be instructed with `livenessProbe` sections on how to check if an application within a container is deadlocked. For the web server, it performs an HTTP GET request starting 30 s after the container begins and repeating every 10 s. For the database server, it checks TCP connectivity on port 3306, starting 30 s after startup and repeating every 10 s. If a liveness probe fails, Kubernetes restarts the container to restore functionality according to a specified restart policy.

Further, Kubernetes allows scaling the number of pod instances running in parallel in response to changing loads or failures. Kubernetes can be instructed to maintain a given number of pods at all times and to scale horizontally up or down in response to specific criteria, such as CPU or memory usage. For example, if the CPU usage of a set of pods exceeds 80 %, Kubernetes can automatically create additional pod instances to distribute the load more evenly. Conversely, if the CPU usage drops below a certain threshold, Kubernetes can reduce the number of running pods to save resources. Kubernetes also manages load balancing through built-in service mechanisms.

A core concept of Kubernetes is the reconciliation loop, which ensures that the system's observed state matches the desired state. For example, it ensures pod resource consumption is within defined limits, liveness probes succeed, and pod replication is aligned with the desired configuration. The reconciliation

loop continuously monitors the system's current state and takes specified actions to return it to the desired state.

Even though management and monitoring run transparently for the application, using dedicated orchestration tools like Kubernetes requires the application to be containerized and its requirements to be explicitly specified. This additional step can make adoption more complex, but it ultimately provides robust management, scalability, and reliability for distributed applications.

4.6.2 *Integrated application orchestration*

This approach integrates orchestration functionalities directly into the application, enabling real-time, context-aware responses to failures, load changes, and evolving requirements. Having immediate access to its performance data, the application can use resources more efficiently and react faster to dynamic conditions. This tight coupling enhances resilience and adaptability, allowing the application to adjust its behavior autonomously to maintain optimal performance and reliability.

CONSTRAINTS

```
node_constraint(Host) :-
  comonnode{hostname:Host},
  alive(Host),
  light_loaded(Host),
  is_avail(Host),
  get_node_attr(Host,cpuspeed,Cpuspeed),
  get_node_attr(Host,freecpu,Freecpu),
  Cpuspeed*Freecpu/100>1.5.

group_constraint(NodeList) :-
  assemble_values(NodeList,location,Locs),
  length(NodeList,Len),
  Max is ((Len-1)//4)+1,
  ( for(I,1,4), param(Locs,Max) do
      count_element(I,Locs,Num),
      Num=<Max ).

path_constraint(LenList, Max) :–
  max(LenList, Max),
  diameterUtil(_,DiameterMax, _),
  Max<DiameterMax.
```

COST FUNCTION

```
migration_cost(Actions, MigrateCost) :-
   count_element(add, Actions, AddLen),
   count_element(remove, Actions, RmvLen),
   addCostParam(AddParam),
   removeCostParam(RmvParam),
   MigrateCost is AddParam*AddLen +
           RmvParam*RmvLen.
```

UTILITY FUNCTION

```
util_function(NodeList,Util,Params) :-

 % Compute utility values for different node attributes
 fiveminloadUtil(LoadMin,LoadMax,LoadWeight),
 assemble_values(NodeList,fiveminload,LoadList),
 util_value("<",LoadList,LoadMin,LoadMax,LoadUtil),

 % Omitting utility values for liveslices and freecpu, the same as above

 % Utility of max distance from fixed nodes to the overlay
 findall(P,fixednode(P),Fixed),
 minlatency(MinLat),
 maxneighUtil(_,NeighMax,NeighWeight),
 get_nearest_neighbor_list(Fixed,NodeList,NeighList),
 max(NeighList,MaxDist),
 util_value("<",[MaxDist],MinLat,NeighMax,NeighUtil),

 % Utility of overlay network diameter
 diameterUtil(_,DiamMax,DiamWeight),
 util_value("<",Params,MinLat,DiamMax,DiamUtil),

 % Weighted average of the utilities above
 weighted_avg([LoadUtil,SliceUtil,CpuUtil,NeighUtil,DiamUtil],
 [LoadWeight,SliceWeight,CpuWeight,NeighWeight,DiamWeight],Util).
```

Definition of util_value:

$\textbf{util_value}_{<} = \text{avg}_i \, ((x_{max} - \text{bound}(x_i))/(x_{max} - x_{min}))$

$\textbf{util_value}_{>} = \text{avg}_i \, ((\text{bound}(x_i) - x_{min})/(x_{max} - x_{min}))$

$\text{bound}(x) = \max(\min(x, x_{max}), x_{min})$

CONFIGURATIONS

```
fiveminloadUtil(0, 10, 2).   liveslicesUtil(0, 10, 1).   freecpuUtil(1, 4, 3).   maxneighUtil(0, 500, 2).   diameterUtil(0, 1000, 2).
addCostParam(0.012).   removeCostParam(0).
```

Figure 4.6: Publish/Subscribe application requirements expressed as a set of Rhizoma constraints [Yin+09].

Rhizoma [Yin+09] and GoPrime [Cap+16] and by Al-Oqily et al. [AK09], provide approaches where orchestration management is provided by a policy layer embedded in the application. To give a deeper insight into these approaches, this section focuses on Rhizoma and closely connects the application with a decentralized management system called Rhizoma, turning it into a self-deploying system similar to early "worm" programs. A computer worm is a self-replicating malware program that spreads across networks without needing to attach itself to a host program or requiring human intervention. The application specifies its resource requirements to the Rhizoma management system as a constrained optimization problem. Constraint logic programming defines a set of constraints an application must satisfy and a goal function that optimizes within those constraints. Rhizoma monitors the resources on which the application runs and decides at runtime if new resources are needed or if existing resources can be released to maximize its utility continually [Yin+08].

Figure 4.6 illustrates the requirements for a publish/subscribe application using Rhizoma constraints. These constraints encompass node, group, and network path constraints. **Node constraints** (top left of the figure) employ several predefined predicates, such as "alive" for nodes that must respond to ping requests and accept SSH connections and "light loaded" for specifying maximum memory pressure and five-minute load averages. **Group constraints** (second box on the left side) are applied to any group of nodes, ensuring, for instance, that nodes are evenly distributed across four geographical regions. **Path constraints** (third box on the left side) specify the desired network properties, such as setting a limit on the maximum diameter for the shortest path between any two nodes in the overlay network. Additionally, Rhizoma defines a **utility function** used to compute a utility value for each possible deployment, aiming to optimize this value. The function represents a weighted average of the deviations of different node parameters from their ideal states, taking into account factors such as the five-minute load on each node, available CPU resources, network diameter, and maximum latency. Moreover, a **cost function** is included to evaluate the cost of a specific deployment and the migration process to it. This is especially important for commercial cloud services, as it directly aligns with the provider's pricing model. Optimizing for actual costs is a key characteristic of Rhizoma. An example configuration of these function values is shown at the bottom of the figure.

Concluding, using knowledge-representation techniques in distributed systems is widespread in work on intelligent agents [Syc+96]. There has not been much related work in autonomous, self-managing distributed systems

since then, outside the malware community [Yin+09]. Approaches only support system-level metrics, not application-level metrics used in the optimized process.

4.7 CHAPTER SUMMARY

Related work was discussed in terms of the different development life cycle phases. Now, related work is classified against the requirements identified in chapter 3. The tabular overview in table 4.2 illustrates which related works fully (●), partially (○), or not at all (blank) meet the requirements. In some cases, a lack of data prevented a conclusive statement, indicated by a "?". This evaluation follows a summary highlighting research gaps and unaddressed issues in existing work.

Dynamic overlay network programming: Previous approaches, like [STS08; Rod+04; Buf08; Net20; DAS23], focus on separating the application from the implemented overlay scheme by providing a overlay-agnostic interface, such as the presented Common API for structured overlays [Dab+03]. This separation allows the overlay implementation to be switched transparently for the application, providing flexibility when different overlay schemes need to be evaluated or when the application's requirements change. However, these approaches share the limitation that the choice of overlay implementation must be made at the application's *build time*. This means that the developer must decide on a concrete overlay scheme before deployment and operation of the application. Consequently, the application cannot switch between overlay schemes at *runtime* and can only react to changing constraints within the current scheme.

The only approaches that theoretically support runtime adaption of overlay schemes are [Buf08; AK09; Yin+09]. However, these approaches primarily focus on overlay resource policy description. They allow for the theoretical description of high-level goals to be fulfilled by dynamically selecting an overlay scheme at runtime, but this capability is not actually addressed or implemented in the approaches.

Self-configuration: Some approaches support self-configuration only partially. However, this often comes with the limitation that self-configuration is fully achieved iff a specific overlay scheme is chosen at build time, or the approach supports only a specific overlay scheme (e.g., DHT) as seen in [Sto+02; For04; RSS10]. Therefore, approaches that meet this requirement often exclude those that fulfill the requirement of "dynamic overlay programming".

Table 4.2: Related work in overlay network construction systems.

System / ***Author***	**Year**	Dyn. overlay	Self-conf.	Services	Efficiency	Robustness	Scalability	Extensibility	Security
		Requirement							
JXTA [Gon01]	2001	●			○	●	○	●	●
i3 [Sto+02]	2002		●		●	●	●	●	●
MACEDON [Rod+04]	2004	○		○	○	?	●	●	?
UIP [For04]	2004		●		●	●	●		●
Behnel et al. [BB05]	2005	○	●	●	●	○	○		
Kumar S.D et al. [KB06]	2006		●	●		●	●		?
Buford [Buf08]	2008	●	●			●	○		
Overlay Weaver [STS08]	2008	○		○	○	?	●	●	?
Akka [Lig09]	2009			○		●	●	●	●
Al-Oqily et al. [AK09]	2009	●	●	●	○	○			○
Rhizoma [Yin+09]	2009	●	●	○		●	●		?
SOLID [RSS10]	2010		●			●	●		●
Jadex [BP12]	2012	○	●		●		○	●	○
Kubernetes [Kub14]	2014	○	○	●	○	●	●	●	○
ZeroTier One [Zer14]	2014		●		●		○		●
GoPrime [Cap+16]	2016		●	●		●	●		?
ActorEdge [AZ17]	2017		○				●		
EmbJXTAChord [BL18]	2018		○		●	●	●	●	●
Tailscale [Tai19]	2019		●		●		○		●
FogBrain [FB20]	2020		●	●		○	○		○
libp2p [Pro20]	2020				●	●	●	●	●
MARIO [Bro+20]	2020		●	●		○	○		○
OpenZiti [Net20]	2020	○	○		●	●	●	●	●
Parallel Theatre [Nig21]	2021		○				●	●	?
Idawi [Hog22]	2022				●	●	●	●	?
EdgeVPN [SAF23]	2023		●	●	●	●	●	●	●
INSANE [Ros+23]	2023		●	●		?	?	●	?
`Oakestra` [Bar+23]	2023	○	●	●	○	●	●	●	?
Diaz Rivera et al. [DAS23]	2023	○	●		●	●	●	●	●

● := fully met ○ := partially met (blank) := not met ? := unknown

It is also noteworthy that some presented approaches focus on specific functionalities, such as low-latency pairwise communication in the case of INSANE [Ros+23], or location transparency as seen in Jadex [BP12], Tailscale [Tai19], and ZeroTier [Zer14]. These approaches are self-configuring within their clearly defined working area.

Additionally, it should be mentioned that approaches like [Buf08; AK09; Yin+09] construct only a conceptual framework for self-configuration without considering this aspect in the context of dynamic overlays.

Overlay services: In terms of flexible overlay services, the observation is similar to self-configuration. Many approaches provide solutions that address specific sub-problems of overlay construction in untrusted environments, such as focusing on communication constraints, without offering extensions [Buf08; Yin+09; Net20; DAS23].

Efficiency and Robustness: It has often been observed that strong assumptions about the underlay in terms of control and availability are made, which only apply to limited environments like data center networks [Kub14]. Operation in heterogeneous and dynamic networks is often considered out of scope. Only a few approaches [BP12; For04; Hog22; BL18; Zer14; Tai19; SAF23] address operation in more flexible network environments e.g. by implementing NAT traversal or failover mechanics. Approaches like [Pro20; BL18] provide low-level building blocks for the construction of overlays and are thus capable of being efficient and robust in principle. However, whether these approaches fulfill these properties depends on the specific overlay scheme implemented.

Scalability and Extensibility: This is a fundamental requirement of many overlay schemes, which is why many approaches partially fulfill these requirements. Scalability and extensibility often come at the cost of reduced or nonexistent control over the overlay regarding dynamic overlay programming or self-configuration, which require a certain degree of central coordination. This central coordination can limit the system's overall scalability.

Security: Security is often not considered and is frequently omitted by many approaches, as indicated by the numerous "?" symbols for this requirement in table 4.2. When security is addressed, it is typically focused on controlling access to the overlay resources and ensuring the integrity of the overlay while operating in an untrusted underlay [Sto+02; For04; RSS10]. Approaches like [Zer14; Tai19] provide confidentiality by encrypting all data plane communications. Other approaches, such as [Gon01; Pro20], offer low-level building blocks that at least provide security primitives.

In summary of prior work, each system focuses on one or a few aspects of the overall topic, no system meets all the discussed requirements to the best of our knowledge. A recurring observation is the frequent absence of dynamic overlay reconfiguration in response to changing application requirements or resource constraints. Furthermore, support for dynamic overlay services is often lacking, which is necessary for resource management in heterogeneous environments. Existing applications frequently require modifications to be compatible with a given approach, or limitations in the application design process may arise.

There is a lack of solutions that can dynamically reconfigure overlays at run time without placing strong requirements on the physical network capabilities. A robust solution should overcome limitations through appropriate techniques, such as NAT traversal, and support additional overlay services, which many applications require. Overlay design is complex, so developers should be supported in constructing overlays by allowing them to define the desired outcome without specifying each step necessary to achieve it.

Overall, the need for a comprehensive approach that addresses these gaps remains an open research area.

5 System architecture for SDON at the edge

In the preceding chapters 3 and 4, the requirements for a software-defined overlay networking system are identified, and the relevant literature is reviewed. The review found that numerous existing systems facilitate the development of distributed applications by applying overlay networking techniques. However, while these systems address certain aspects of the broader topic, none fully satisfy all the identified requirements. This chapter, therefore, presents an architecture for a software-defined overlay networking approach that meets all requirements. The structure of this chapter is as follows:

- *Section 5.1 outlines the general system model of a software-defined overlay networking system.*
- *Section 5.2 describes the overall system architecture of a distributed middleware for a simplified development and deployment of overlay networks at the network edge. This includes the rationale for splitting the middleware into three conceptual layers, each dealing with one aspect of software-defined overlay networking.*
- *Section 5.3 introduces the communication layer. It overcomes restricted connectivity and the lack of trust in edge networks to link edge devices dynamically.*
- *Section 5.4 describes the service layer. This layer allows the flexible integration of services into the overlay, like custom routing, quality of service (QoS), or security. Therefore, it helps applications in making better use of edge resources.*
- *Section 5.5 introduces the software-defined overlay networking layer. This layer provides central control and automatically interprets high-level functional requirements to configure overlay networks. Thus, it simplify deployment of overlay networks to the edge.*
- *Section 5.6 summarizes this chapter.*

5.1 SOFTWARE-DEFINED OVERLAY NETWORKING SYSTEM MODEL

This section describes a system model that combines the advanced network management capabilities of software-defined networking (SDN) (Section 2.1) with the flexibility of overlay networks (Section 2.3) to operate independently of underlying edge network challenges. SDN systems allow network administrators to define the desired outcomes for the network rather than manually specifying individual configuration steps. These systems automatically configure network devices and paths to meet performance, security, and reliability requirements. Many SDN systems, like ONOS and Ryu [Ope24; Ryu17], require control of the underlying network infrastructure, which requires all networks involved to be operated by the same administrative domain, which is typically only provided in data center-like networks [LL14]. In contrast, overlay networks are often deployed in untrusted environments, spanning administrative domains without the need to control the network configuration. For this reason, overlay networks construct an abstraction layer above this constrained environment (Section 2.3). Overlay networks provide their own services without requiring changes to the existing network infrastructure. Therefore, a normal SDN system cannot be used in untrusted environments because it makes strong assumptions about underlying infrastructure control. Whereas a conventional SDN system requires network infrastructure configuration to adapt to requirements, a software-defined overlay networking (SDON) system creates an overlay network that does not require control of the underlying infrastructure.

In the remainder of this section, the impact of distributed application requirements on such a system is first discussed, followed by the challenges the system faces in untrusted edge networks. Finally, the next section presents an architecture for a SDON system that connect these two "worlds" of application requirements and edge network challenges.

Distributed applications requirements

Distributed applications demand a robust and flexible network environment where individual components can communicate seamlessly. Such a network environment is often not present at the edge. A system in which overlay networks are programmed is required to integrate them into the application. This approach allows the network to be created and used transparently to match the application's needs (FR 1). The overlay network deployment process involves

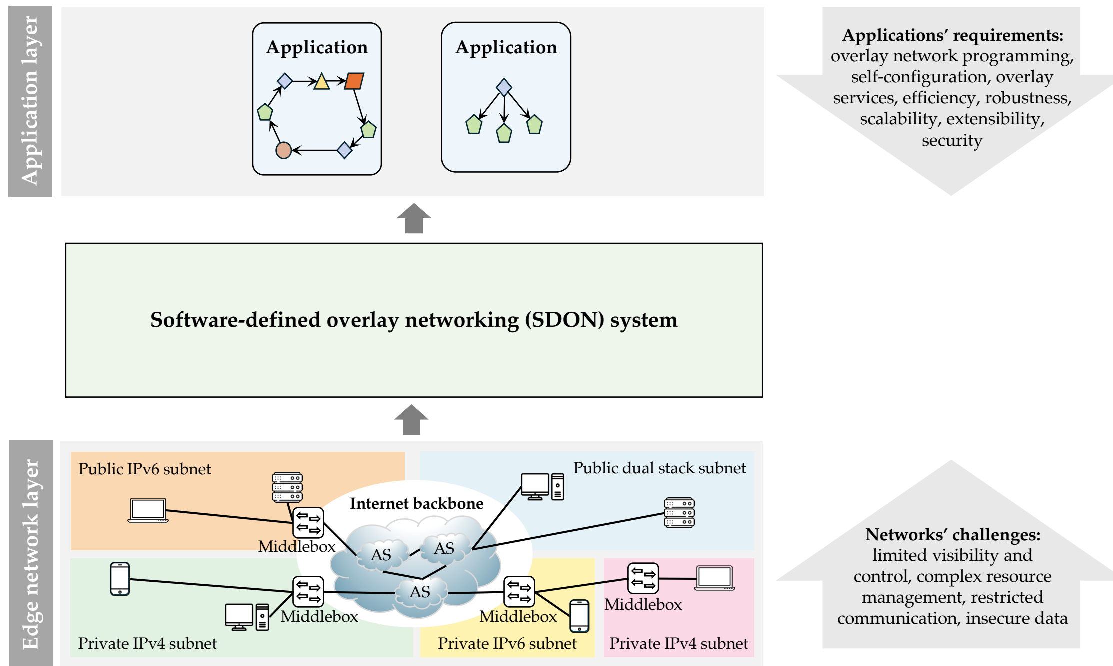


Figure 5.1: Model of a SDON system: Atop applications requiring well-behaving overlay networks. At the bottom, independent best-effort behaving devices. In between, the SDON system provides means to "connect" these two layers.

the application developer formulating high-level functional goals. Afterward, the system automatically **self-configures** the overlay by enforcing the necessary lower-level configuration steps to achieve the formulated goals (FR 2). Since no assumption can be made about the behavior of edge devices the overlay is deployed on, the system must **support overlay services** to provide additional services, like routing, QoS, or security in an otherwise best-effort and untrusted environment (FR 3). Furthermore, the system must be **efficient**, **robust**, **scalable**, **extensible**, and **secure** to optimize the utilization of device resources and to be capable of supporting a wide range of application deployment sizes (NFR 1–NFR 5).

Challenges of edge networks

The edge networks pose challenges to SDON, respectively when distributed applications are operated across multiple subnetworks under different administrative control, e.g. the Internet [TFW21]. Here, developers have **limited visibility and control** over the configuration and behavior of devices and networks. Here, reconfiguring devices and networks to improve application performance is constrained or entirely unfeasible. The types of devices en-

countered, such as smartphones, desktop computers, and servers, as well as the networks they operate on, including residential, corporate, cloud, public, and mobile networks, can vary significantly in their capabilities. Additionally, networks are dynamic and edge resources can be unpredictable, as they may suddenly exit the system or change their behavior without prior warning, making **complex resource management** necessary. The construction of an overlay networks needs coordination and synchronization. Although almost all devices are *somehow* connected with the Internet, **communication is restricted** between many devices [DA08; For04]. Incompatible protocols are used side-by-side, such as IPv4 and the successor IPv6, requiring IPv6 transition mechanisms (like NAT64, 4in6, or 6in4 [MBB11; DC98]). Those mechanisms introduce communication indirections between IPv4 and IPv6. Furthermore, the widespread deployment of middleboxes on the Internet render many devices unreachable without additional countermeasures like NAT traversal (refer to section 2.4 for more information about NAT) [Haa+16; WP17]. A middlebox is an intermediary network device that performs functions beyond the standard operations of an IP router, like gateways or firewalls [BC02]. Finally, all information transmitted in these environments must be considered as **insecure data** due to the absence of built-in security services and the public nature of the Internet. Thus, received information can be manipulated, including the sender and receiver information of the communication and its content, in the absence of security protocols.

The system model described is illustrated in Figure 5.1. The SDON system application layer at the top contains applications, each requiring a different overlay network. At the bottom, the edge network layer consists of devices that can host the applications and establish overlay networks for the distributed applications. The application will transparently use the overlay network. In between, the SDON system, tasked with addressing the needs of applications despite the challenges present in the underlying edge network.

The described system model can be used for the following two usage scenarios: i) Operation on dedicated devices that wait to host services of various distributed applications, which then transparently use the SDON overlay. This enables SDON system to realize an edge computing platform, on which any application can be deployed and operated. ii) Peer-to-peer operation, where SDON system is tightly coupled with a specific application. In this case, the user starts the application as usual, with SDON system transparently integrated into the application. While the user operates the application normally, an overlay established by SDON system is used transparently.

In the next section, an architecture for such an envisioned SDON system is presented by identifying required components, specifying their needed functionalities and defining their interfaces.

5.2 OVERVIEW ON THE DISTRIBUTED MIDDLEWARE SYSTEM

This section outlines the overall architecture of the conceptualized Software-defined overlay networking (SDON) system. Given the system model presented (Section 5.1), a realization as distributed middleware is suitable. A middleware is a software layer located between the application and the operating system, abstracting the complexities of the device resources to simplify the programming and management of distributed applications [Mah04]. The SDON middleware is distributed, meaning it can run on every device that wants to offer its local resources to run distributed applications that transparently build and use the overlay. Developers can use the middleware to provide applications equipped with an overlay network, giving them an idealized view and control of the underlying device resources. The middleware transparently handles all the necessary tasks regarding overlay networking.

Bringing software-defined networking to the edge

In the previous section, four main challenges were identified for overlay networking in untrusted edge environments: limited visibility and control, complex resource management, restricted communication, and untrusted resources. Each of these challenges must be addressed using appropriate techniques, which can then be combined to form the desired SDON middleware. For this reason, the process enabling SDON is divided into the following subsequent steps (Figure 5.2):

1. **Link edge device efficiently and securely:** Ensuring that as many devices as possible can securely reach each other is a key task in overlay network construction. Establishing connectivity enables individual devices to be used for flexible network topology creation. Additionally, equipping each device with a cryptographic identity ensures the secure identification of devices. This step provides the primitives for flexible, efficient, and secure overlay network construction.

2. **Integrate dynamic overlay networks seamlessly:** Control communication behavior between devices by overlay services. Edge networks

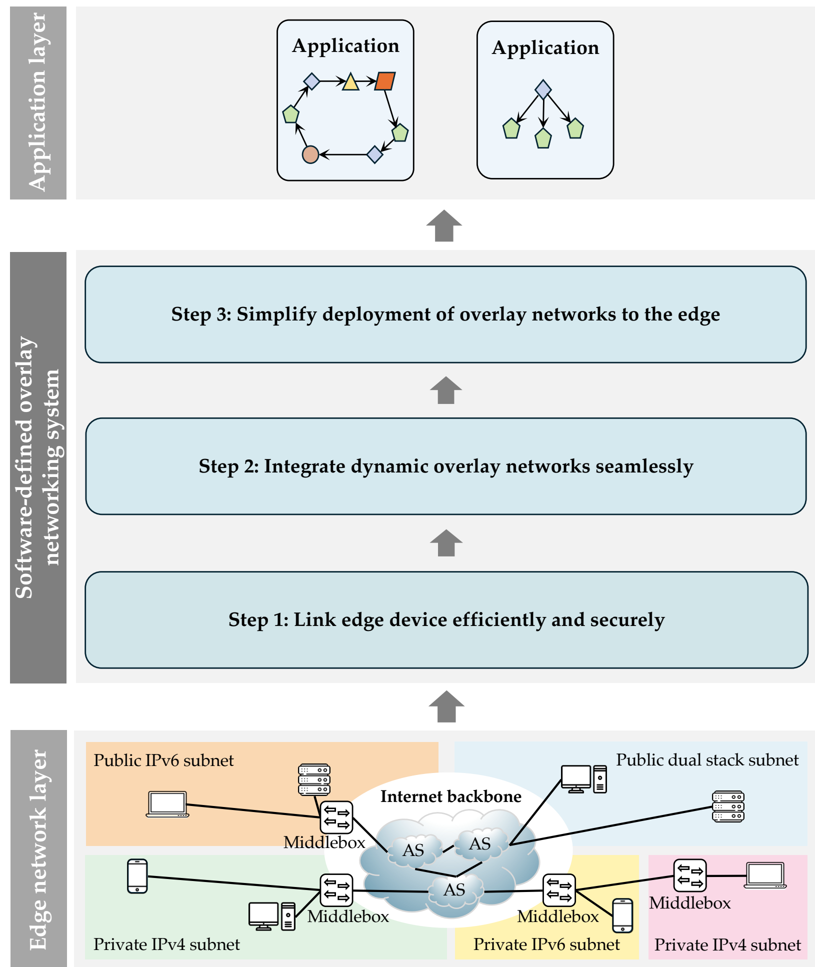


Figure 5.2: Software-defined overlay networking is conducted in three steps. First, edge devices are efficiently and securely linked. Second, requested overlay services are employed. Third, overlay network deployment is simplified by achieving central overlay control and automatically repairing unintended network changes.

typically operate on a best-effort service level, which does not meet the requirements of many applications to function properly. This step provides the ability to control the behavior of the overlay network by overlay services that, e.g., enforce QoS or other services that make the edge appear more reliable.

3. **Simplify deployment of overlay networks to the edge:** SDON enables flexible overlay network management using high-level functional requirements. This assists developers in managing overlay networks, as a network management system takes over many tasks related to device resource selection, overlay construction, and maintenance.

This three-step process shows how underlying device resources are transformed step by step to an overlay network managed by intents. Initially, the edge network layer consists of a collection of autonomous systems, each containing one or more subnetworks, operating on different IP versions and including heterogeneous devices and configurations, some of which are deployed behind middleboxes. As a result, communication is heterogeneous, restricted, and insecure. Therefore, a first layer constructs an overlay network, enabling secure connectivity between all devices. A second layer dynamically adds services to the overlay network. The third layer allows the overlay network to be centrally managed through high-level function requirements, enabling automatic resource selection, overlay construction, and maintenance.

Architecture overview

The corresponding overall middleware architecture is shown as an UML component diagram [OMG11] in Figure 5.3. The middleware is positioned between the application on top and the device's edge network interface at the bottom. Three sub-components are included within the SDON middleware component, corresponding to the three steps discussed earlier in this section.

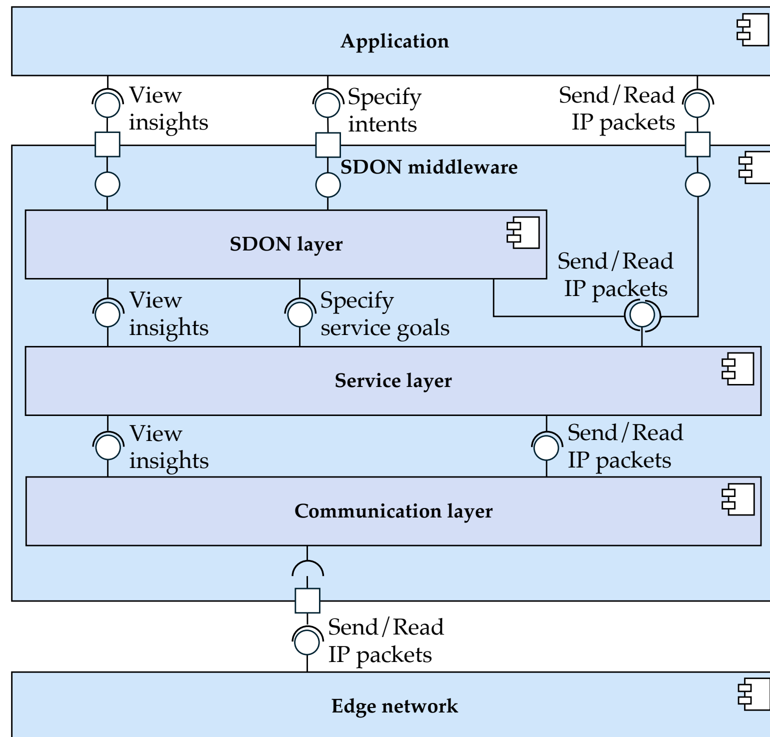


Figure 5.3: Overview of SDON middleware architecture as UML component diagram. The middleware utilizes the physical network to provide distributed applications with overlay networks specified by intents.

The remainder of this section is structured as follows: First, the required interfaces of the SDON middleware are presented. This is followed by a discussion of the provided interfaces. Finally, the sub-components are introduced.

Required interfaces

The SDON middleware requires one interface from the edge network:

- **Send/Read IP packets:** The edge network is utilized for control plane and data plane traffic, serving purposes such as overlay network management and application traffic. Therefore, the middleware must be provided with

an interface allowing sending and reading IP packets for communication with other remote devices running the middleware.

Provided interfaces

The SDON middleware provides a total of three interfaces to applications:

- **Send/Read IP packets:** While the required "Send/Read IP packets" interface handles underlay network communication, this interface is used for overlay network communication. The overlay network transparently processes all packets sent via this interface. This interface enables the application running on the local device to communicate with applications running on remote devices, allowing coordination and synchronization, thereby forming a distributed application. This interface processes all application communication and applies the overlay network routing and traffic-control.
- **Specify intents:** This interface is used to pass the application's requirements for the SDON system. These requirements are described as intents, where only the desired high-level goal is specified without detailing the necessary steps to achieve it. This abstraction simplifies the management of overlay networks [LF23]. While the application only specifies its intents, the middleware determines and automatically applies the necessary steps, such as neighbor discovery, latency measurements, link establishment, and dynamic adaption to node churn and alignment to the intents.
- **View insights:** This interface provides insights into internal middleware information. These internal details include the status of intent implementation, overlay network configuration, monitoring data, and device information. While intent-based networking (IBN), as facilitated by the previous interface, is typically designed to prevent applications needing to care about these internal details – since the SDON system abstracts and implements the provided intents automatically – it is advantageous in this case to provide applications with such an interface. IBN is typically applied in environments with a controlled availability of resources or the ability to deploy new resources (e.g., cloud environments). The SDON system is operated in potentially untrusted environments. Therefore, no assumptions about available resources can be made. The application can adapt to the edge network constraints using this interface if necessary.

Sub-components

The *SDON middleware* component includes three sub-components: the *communication layer* (Section 5.3), the *service layer* (Section 5.4), and the *SDON layer* (Section 5.5) that will be described in detail in the following sections.

5.3 COMMUNICATION LAYER

The *communication layer*, as depicted in Figure 5.4, provides fundamental communication capabilities, secure communication and mutual authentication among individual middleware devices. This component optimizes device resource utilization by maximizing device connectivity and identifying the most efficient edge network paths. Establishing a key infrastructure at this low layer, by mutually authenticating devices and ensuring confidential communication, provides the secure foundation needed for all subsequent steps.

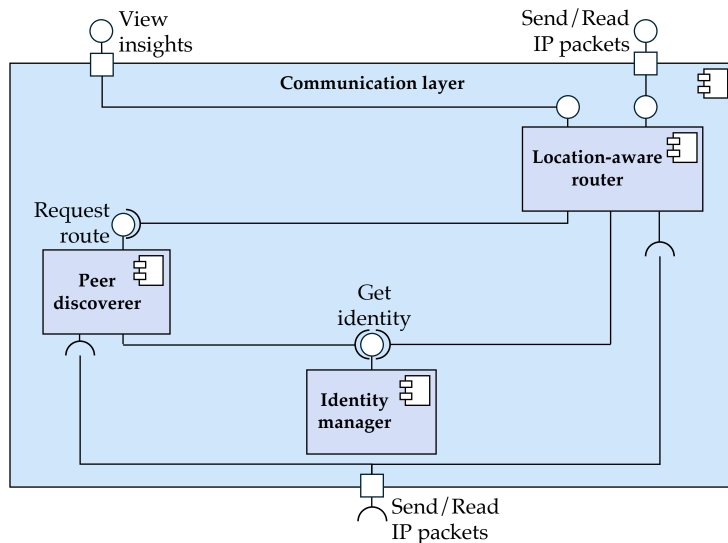


Figure 5.4: Communication layer architecture as UML component diagram. The layer enables secure communication between as many edge network devices as possible. It abstracts from communication barriers and provides a unified interface for communication.

After presenting this component's task of creating a secure communication overlay network, the required interfaces, provided interfaces, and sub-components are introduced.

Required interface

The *communication layer* requires the following interface:

- **Send/Read IP packets:** The edge network is utilized for peer discovery and communication with remote middleware devices. This interface is provided by the parent *SDON middleware* component (Section 5.2).

Provided interfaces

The *communication layer* provides two interfaces:

- **Send/Read IP packets:** This interface differs from the required "Send / Read IP packets" interface, as it processes communication overlay network messages, not edge network packets. All communication passed over this interface is routed and secured via the overlay network provided by this component. Communication overlay addresses are used instead of IP addresses for peer identification. The reasoning for introducing a separate overlay address schema is explained in Section 5.3. This interface is used by the parent *SDON middleware*.
- **View insights:** This interface provides internal information regarding the construction of the communication overlay network and edge network metrics. This interface exposes information such as discovered peers, paths to reach them, and corresponding path metrics. This interface is used by the *service layer*.

Sub-components

The *communication layer* component consists of three sub-components: *Identity manager* (Section 5.3.1), *peer discoverer* (Section 5.3.2), and *location-aware router* (Section 5.3.3), which are now described in detail.

5.3.1 *Identity manager*

The *identity manager* holds information about the identity of the locally running middleware. While IP addresses are used by the edge network to identify hosts, the communication overlay network employs its own addressing scheme for the following reasons:

- IP addresses are often unintentionally and unknowingly shared among multiple devices when deployed behind firewalls or NATs, making identification based on IP addresses impossible.
- These deployments also cause devices to be reachable at different addresses (e.g., WAN address vs. LAN address), depending on whether the sender is in the same subnetwork, rendering IP routing information context-sensitive.
- Further, a device's IP address changes over time when being mobile and roaming between networks or connected through dial-up Internet access, which assigns the device a new address every time a new connection is established. Here, it is not known which and when a new address will be assigned, making it difficult to identify a device over a longer period of time.
- IP addresses can be forged easily when no additional network security protocol, e.g., IPsec, is used.

Therefore, the *identity manager* employs virtual identities consisting of a public/private signing key pair, where the public part is used as the address for communication. Using a public key as a peer identifier is beneficial because it ensures both uniqueness and security. In contrast, the private part is used to authenticate and initiate encrypted communication [Cas+03]. A signing key pair allows the device to securely prove its ownership of an address to other entities and to ensure that messages originating from this address are recognized as legitimate.

Provided interface

- **Get identity:** This interface provides the middleware's local identity, including the address and corresponding cryptographic key material.

5.3.2 *Peer discoverer*

The *peer discoverer* is responsible for finding routable IP endpoints of remote middleware devices. This process involves exchanging routing information, overcoming communication barriers, verifying reachability, and providing secure communications. Establishing routability between as many devices as possible using most local network paths helps efficiently utilize available devices. Establishing secure communication is necessary as otherwise received

information, like the identity of a peer or application messages, can not be trusted. First, we explore the challenges of establishing routability and secure communications, and then present a protocol design that effectively addresses both issues.

Peer discovery

As stated in the system model (Section 5.1), the widespread deployment of NATs and firewalls render many devices unreachable from the Internet, which, in turn, prevents these devices from being used flexibly in overlay networks. Therefore, this component must provide means to overcome these communication barriers. Due to the heterogeneous behavior of NATs and firewalls, multiple traversal techniques have emerged, each with its rationale for use:

- The most successful NAT traversal method is *relaying* (Section 2.4.2.2). Here, communication with NATed devices is achieved by relaying all communication through a public reachable server. This method works with all NAT behaviors but has the drawback of making the relay a single point of failure (SPOF), a potential bottleneck, and a privacy concern. Further, this relayed communication increases the communication latency.
- Another method is *hole punching* (Section 2.4.2.3), which can establish direct network connectivity for most NAT behaviors. There is one exception: If both devices trying to communicate with each other are operated behind a NAT applying endpoint-dependent mapping and filtering (often named as "Symmetric NAT", see Section 2.4.1.2), this method will not succeed reliably. In this case, relaying is the only way to enable connectivity.
- *Port forwarding* (Section 2.4.2.1) – when supported by the NAT – changes the NAT behavior to endpoint-independent mapping and filtering. This allows the device to become reachable like a public server, increasing the likelihood of successful hole punching.

Individual protocols exist for each of these techniques. For relaying and hole punching, the Interactive Connectivity Establishment (ICE) framework [KHR18] is widely-employed, which combines the protocols STUN [Pet+20], TURN [Red+20], and a signaling protocol like SIP [Sch+02]. For port forwarding, protocols such as NAT-PMP [CK13], PCP [Win+13], and UPnP IGD [BPW13] are required, with the latter relying on SSDP, SOAP, and HTTP (refer

to section 2.4.2.1 for more information about port forwarding). As different NAT devices implement different port forwarding protocols, all protocols must be supported to ensure compatibility with many NAT devices.

Establishing secure peer communications

A cryptographic key agreement must be performed between both devices before a discovered network path can be used securely (Section 5.1). This key agreement is necessary to establish confidential and authenticated communication, and it involves the following steps:

1. Each device generates a pair of public and private keys for key agreement.
2. The public keys are then exchanged between the devices.
3. Each device computes a shared secret key using the received public key and its own public and private keys. This shared secret key is used to encrypt and authenticate subsequent communications.

For key agreement, protocols such as TLS using Diffie-Hellman key agreement are available [Res18; IT21; RTM22]. Since the key agreement process requires connectivity between the two devices, it cannot be performed before NAT traversal. As a result, key agreement protocols are applied after the NAT traversal process has been completed.

Secure peer discovery

For secure peer discovery, the NAT traversal techniques of port forwarding, hole punching and relaying must first be applied to achieve routability, followed by key agreement to secure the connection. Accomplishing this using existing protocols would involve many protocols, each conducting individual handshakes, resulting in additional load on the network and delayed communication establishment. This issue is illustrated in Figure 5.5. The messages 1 to 7 belong to the NAT traversal process (Section 2.4.2.3). This process requires at least 2 RTTs. Depending on the behavior of the NAT, up to two additional `ReachabilityCheck` messages might be necessary. Afterwards, with messages 8 to 9, the key agreement is conducted, resulting in an additional RTT. With this sequential process, devices are utilized less efficiently, which contradicts the efficiency nonfunctional requirement. Additionally, NAT traversal messages are not secured, which allows a potential attacker to perform a MITM attack. This can unintentionally result in an opening in the NAT for the attacker, enabling unauthorized access to the local network.

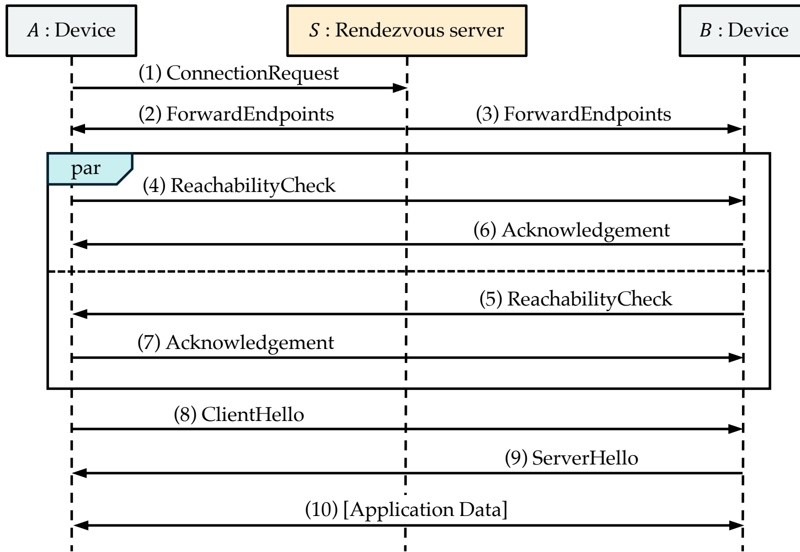


Figure 5.5: Sequential hole punching and key agreement. Messages 1 to 7 belong to the hole punching process and messages 8 to 9 to the key agreement process.

A more efficient approach includes creating a new protocol aware of the whole NAT traversal and key agreement processing, avoiding redundant handshakes. The communication used for NAT traversal can be used for key agreement, saving the subsequent handshake. Further, the NAT traversal messages can be used to initial application communication, further reducing connection establishment time. While the key agreement could be integrated into NAT traversal process, port forwarding cannot be changed as these would involve adjustments to middleboxes, which is impossible as control of underlying network infrastructure cannot be assumed.

The complete secure peer discovery process is shown in Figure 5.6. First, port forwarding is applied to help make any local NAT less restrictive to hole punching. Second, hole punching is applied, which requires both peers to exchange information. While existing protocols only exchange routing information, this signaling is also be used for key agreements. If hole punching succeeds, the discovered direct path is used. Otherwise, communication is relayed through a public reachable server. Unlike existing protocols, this new protocol would better use existing devices, thereby improving the efficiency of the middleware.

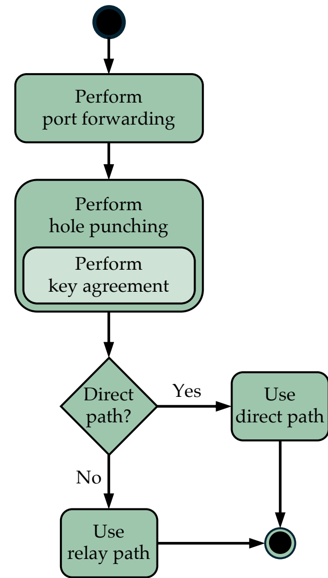


Figure 5.6: NAT traversal process applied by the peer discoverer component.

Required interfaces

- **Get identity:** The discovery process involves agreeing on cryptographic keys with peers, which needs access to the local address and identity's key material.
- **Send/Read IP packets:** The edge network is utilized for peer discovery. This interface is provided by the parent *communication layer* component (Section 5.3).

Provided interface

- **Request route:** This interface specifies to which peers a route is needed, triggering this component to start the discovery process.

5.3.3 *Location-aware router*

The *location-aware router* is responsible for routing using the most local discovered edge network path. The component maintains a table containing all known peers and their corresponding discovered paths. Continuously, all entries in the table are checked to verify peer reachability. Periodic health checks also collect path metrics like latency, jitter, and packet loss. Ensuring reachability and collecting metrics is necessary, as the *service layer* and *SDON layer* require this information to select suitable edge resources for overlay network construction. Further, this router signs and encrypts all outbound communication and decrypts and verifies all inbound communication to ensure an attacker has not forged received data. If a communication to an unknown peer is requested, this component temporarily buffers outgoing communication messages while a route is requested via the "Request route" interface.

Required interface

- **Request route:** This interface is required to request the discovery of a route to a given peer. Used when a communication to a peer with no discovered path occurred.
- **Get identity:** The routing process involves signing and encrypting communication, which needs access to the local address and identity's cryptographic key material.

- **Send/Read IP packets:** The edge network is utilized for route maintenance and processing outbound and inbound application communication. This interface is provided by the parent *communication layer* component (Section 5.3).

Provided interfaces

- **Send/Read IP packets:** This interface processes communication with peers. All communication passed over this interface is routed and secured, as mentioned in the component description. Communication overlay addresses are used instead of IP addresses.
- **View insights:** This interface exposes insights such as discovered peers, their routable IP endpoints, and associated route statistics.

In summary, the *communication layer* creates an overlay network that enables secure connectivity between as many devices as possible on the underlay using the best physical path, as shown in Figure 5.7. The dotted lines indicate the possible any-to-any connectivity, where physical paths are only established if communication occurs. The dashed lines indicate the mapping of devices between layers. This overlay network abstracts from many communication barriers incurred by heterogeneous physical network configurations and middleboxes. Further, secure communication and mutual authentication are provided, enabling devices to authenticate each other mutually and communicate securely.

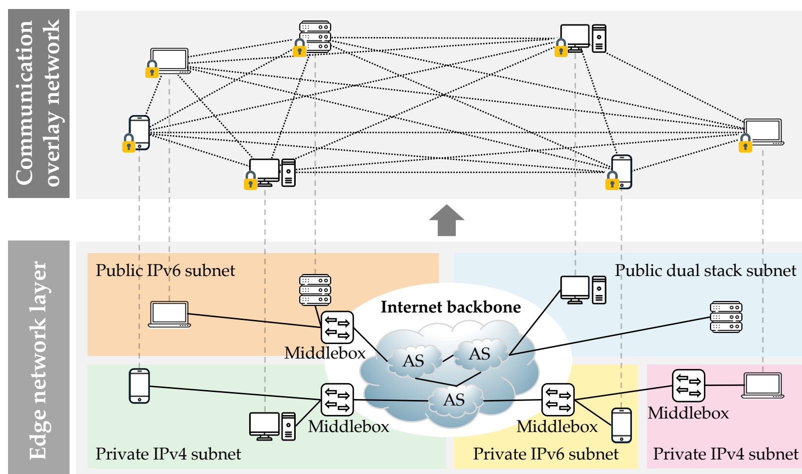


Figure 5.7: Communication overlay network provides secure connectivity between all devices. Heterogeneities and communication barriers of edge networks are abstracted. Dotted lines indicate available connectivity between devices, while dashed lines indicate the mapping of devices.

5.4 SERVICE LAYER

The *service layer* is built upon the *communication layer* (Section 5.3). While the *communication layer* enables connectivity between devices, it inherits the best-effort behavior prevalent in the edge networks. While some applications support running without any guarantees, many need higher requirements to function properly. Therefore, the *service layer* provides capabilities to control the overlay network behavior, like enforcing a given topology, QoS, or security. As a result, this layer elevates the best-effort overlay network to an overlay network, providing QoS guarantees. This component is designed to support a wide range of services to suit various application needs. Thus, goals can, for example, influence the communication between individual pairs of devices or the behavior and organization of groups of devices. Given the virtually limitless range of service requirements, as each application may have unique objectives, the middleware should offer a core set of foundational services while remaining flexible enough to accommodate the addition of new services as needed.

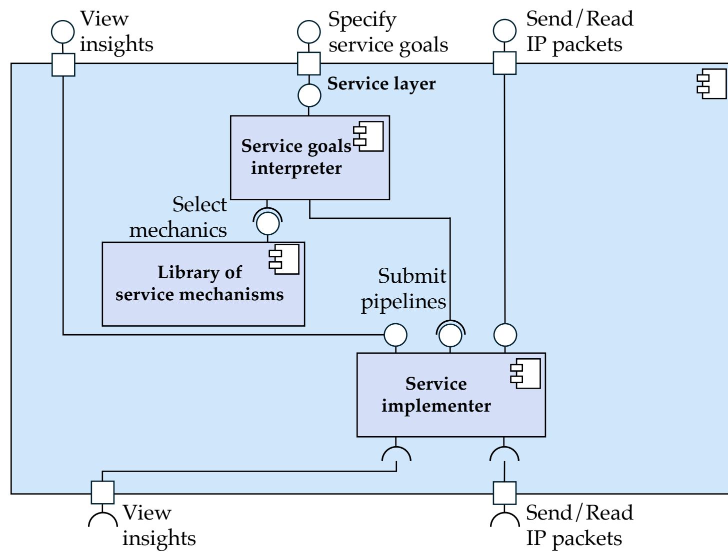


Figure 5.8: Service layer architecture as UML component diagram. The layer implements mechanisms to provide services many applications need to function correctly.

After describing this component's task of making the communication behavior configurable by dynamic integration of overlay services, the required interfaces, provided interfaces, and sub-components are presented.

Required interface

- **Send/Read IP packets:** This component requires access to the *communication layer*'s communication interface as some overlay services might need to modify it.
- **View insights:** This interface is necessary because some services require access to internal information about the underlying environment.

Provided interfaces

- **View insights:** This interface provides internal information regarding the applied overlay services. Information, such as the currently applied service and statistics generated by some of those services, is made accessible alongside insights provided by the *communication layer*.
- **Specify services:** This component provides an interface that accepts a list of required services that are integrated into the overlay and may only regard specific devices and their communication.
- **Send/Read IP packets:** This interface is equal to the one provided by the *communication layer* (Section 5.3), with the difference that all communication passing through this interface undergoes the communication modification enforced by some overlay services.

Sub-components

The *service layer* component comprises three sub-components: the *library of service mechanisms* (Section 5.4.1), the *service goals interpreter* (Section 5.4.2), and the *service implementer* (Section 5.4.3), each of which is now described in detail.

5.4.1 *Library of service mechanisms*

While service goals define the desired outcome (e.g., connectivity, confidential communication, support for distributed lookup overlay service), concrete mechanisms are required to enforce these goals (e.g., perform connectivity checks between edge devices, conduct latency measurement on network links, apply encryption sent over specific overlay links, let edge devices arrange in a

DHT). Thus, the *library of service mechanisms* maintains a library of available mechanisms to enforce service goals. These mechanisms may implement specific protocols, such as a cryptographic key agreement [DH22], authentication, error-control [Wel82], or codecs (e.g., gzip compression). More complex protocols, including examples like the structured Chord-DHT [Sto+03] and the unstructured membership management protocol CYCLON [VGS05], can be used to influence the arrangement of devices. Since service goals vary widely, the library also allows the addition of new mechanisms.

The interceptor design pattern is well-suited for implementing service mechanisms by adjusting the communication behavior on a lower level [Sch+00; RF20; ST23]. It enables modular processing of inbound and outbound messages through interceptors that can independently filter, modify, or analyze messages as shown in Figure 5.9. For example, an interceptor may encrypt outbound messages and decrypt inbound messages independently of any other interceptors, while other interceptors may perform functions such as authentication or replay protection. This independence allows easy addition, removal, or reordering of interceptors, as each interceptor focuses solely on its specific service mechanism.

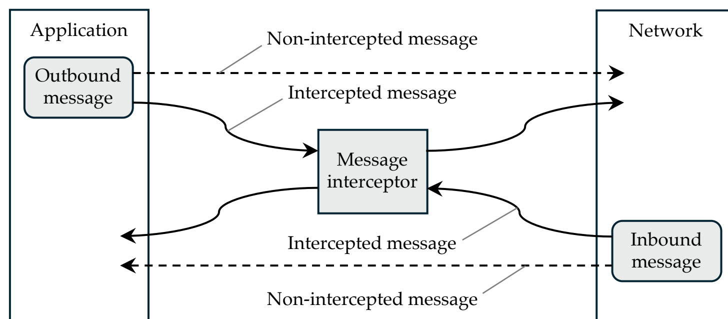


Figure 5.9: Interceptor design pattern used to implement various service mechanisms.

Provided interface

- **Select mechanisms:** This interface provides access to all service mechanisms available to achieve certain service goals.

5.4.2 *Service goals interpreter*

The *service goals interpreter* interprets the service goals received through its interface by mapping them to specific service mechanisms. It tracks available mechanisms in the library and which enforce what goals. Enforcing a goal may

include choosing one or combining multiple available mechanisms. Not all communication requires the same service goals. Therefore, setting different goals for each remote device and message is possible.

Service mechanisms are implemented as independently acting interceptors. This component connects these interceptors in sequence to form pipelines that ensure that all communication passes through such pipelines is processed sequentially by all interceptors, where each interceptor applies its corresponding service mechanism. For each peer, the middleware creates separate pipelines as shown in Figure 5.10. In this example, the processing pipeline for communication with peer A applies encryption and error control. For peer B, only error control is applied. For peer C, only encryption is applied. A list of the available service mechanisms in the middleware can be found in Section 6.3.1.2.

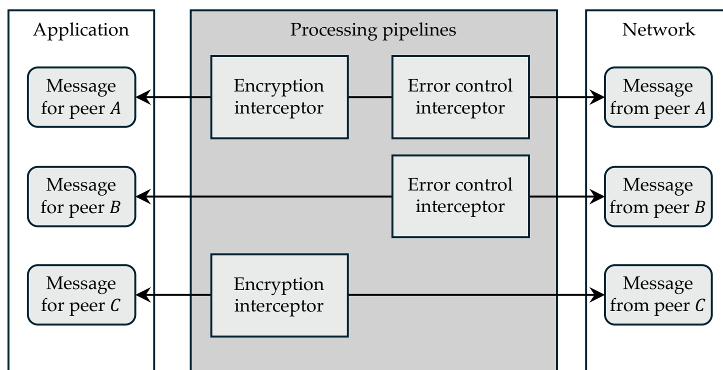


Figure 5.10: Interceptors are chained to create peer-level processing pipelines.

Required interfaces

- **Select mechanisms:** This interface is required to retrieve available service mechanisms from which this component selects to enforce service goals.
- **Submit pipelines:** After goals have been mapped to mechanisms, the mechanisms are chained together, creating processing pipelines. These pipelines are submitted to a component that processes messages through the dedicated pipeline before being passed further.

Provided interface

- **Specify service goal:** This component provides an interface that accepts service goals that must be applied to communication.

5.4.3 *Service implementer*

The *service implementer* ensures that the chosen service mechanisms are enforced to meet the specified goals. It tracks the inbound and outbound communications associated with each peer and processing pipeline, ensuring that the appropriate service mechanisms are applied to the communication stream before passing it along for further processing.

Required interfaces

- **Send/Read IP packets:** Access to the communication interface of the *communication layer* is necessary for this component to apply service mechanisms. This interface is provided by the parent *service layer* component (Section 5.4).
- **View insights:** This interface is essential because certain service mechanisms can benefit from internal information about the underlying environment. For instance, a reliable communication service mechanism might need details about the average latency of a route to configure the acknowledgment timeout for received messages correctly. This interface is provided by the parent *service layer* component (Section 5.4).

Provided interfaces

- **Send/Read IP packets:** This interface provides the application communication where the required service goals have been applied.
- **Submit pipelines:** This interface configures this component to know which communication belongs to what processing pipeline.
- **View insights:** This interface provides internal information regarding the applied service mechanisms. Further, the insights obtained via the required "View insights" interface are also exposed through this interface.

In summary, the *service layer* provides control of the overlay network topology and service guarantees in an otherwise best-effort environment, which does not adequately meet higher-level guarantees required by many applications to work correctly. The service overlay network created is shown in Figure 5.11: The communication overlay network has been "elevated" to a service overlay network with enforced network topology and communication behavior. Solid directed lines indicate established network paths by the *connectivity* goal.

The rounded boxes indicate different service goals applied, like reliability or confidentiality. To save space, the different goals are only depicted by different symbols, each type representing one goal. service goals are mapped to mechanisms that enforce the desired communication behavior. While the *communication layer* facilitates the communication itself, this layer determines the behavior of that communication.

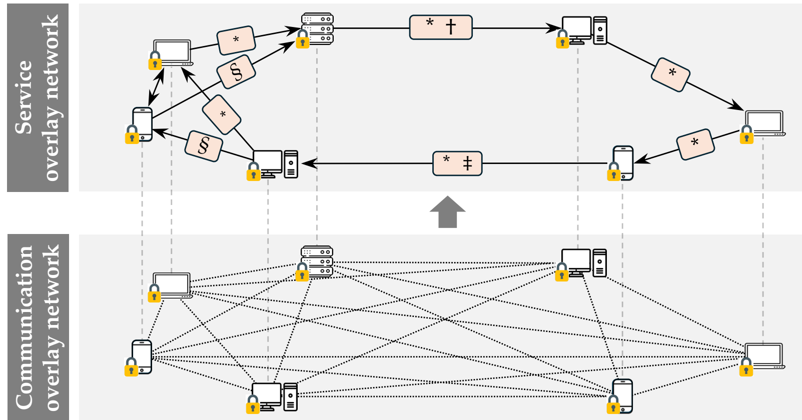


Figure 5.11: Service overlay network provides control of network topologies and service guarantees. Solid directed lines indicate the topology, while rounded boxes indicate service goals. Different symbols depict different goals. The dashed lines indicate the mapping of devices.

5.5 SDON LAYER

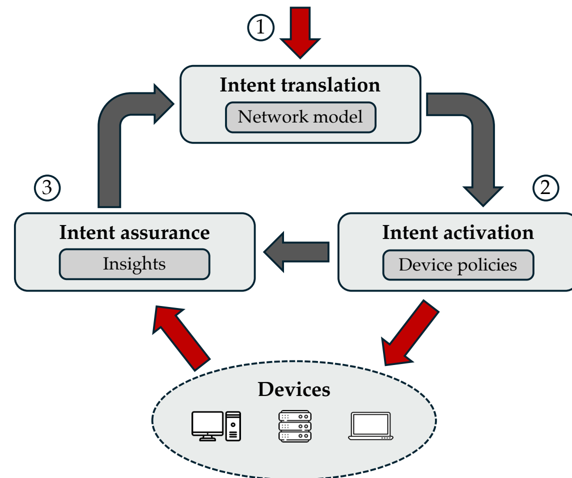


Figure 5.12: Software-defined overlay network programming.

The *SDON layer* enables overlay networking using intents that are high-level functional requirements, as shown in Figure 5.12. The system shown is similar to an IBN system and has been adopted for the use with overlay networks in untrusted environments [LF23]. The SDON system provides an interface ①, allowing applications to express intents – high-level functional requirements – which are then translated into a network model specifying resource constraints and service goals on all overlay network elements, such as nodes and links. This also includes specifying which application components should be deployed on which nodes. Based on this model, device configuration policies are created, and appropriate devices are selected ②. For this selection, information available for each device is considered. Not only is information provided automatically by the lower layers used, but information is also actively gathered from the devices (e.g., by requesting devices to benchmark peer bandwidth). The goal is to select the best available devices for each overlay network node that meet the desired properties. Once suitable resources are selected, the devices are configured according to the policy configurations. During the implementation of these policies, service goals are applied. After the deployment is completed, both the overlay network and the devices used are monitored by this SDON layer. Due to changes in edge resources or newly available information, it may become necessary to repeat the entire process and possibly deploy the overlay network on more suitable devices ③. This closed-loop mechanism ensures ongoing alignment between the intended and deployed overlay network.

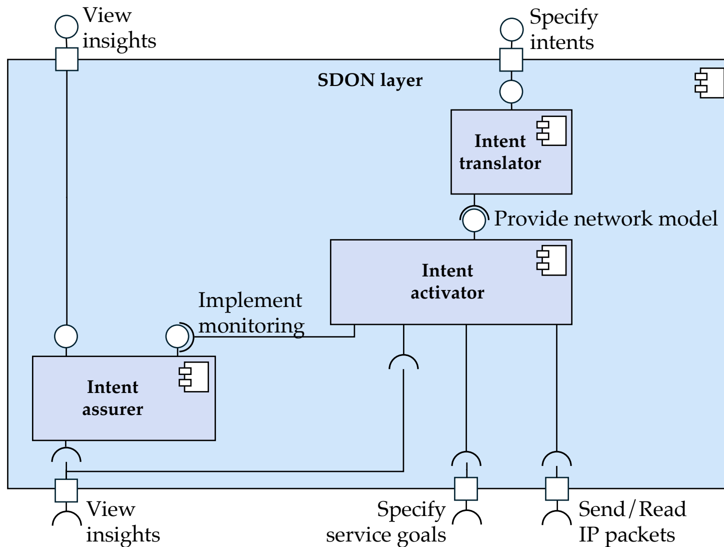


Figure 5.13: SDON layer as UML component diagram. The layer enables the construction and deployment of overlay networks based on intents, representing high-level functional requirements.

After describing this component's task, the required interfaces, provided interfaces, and sub-components are presented.

Required interfaces

The *SDON layer* requires three interfaces:

- **Send/Read IP packets:** This component requires access to an interface for overlay network communication, as activating intents requires coordination and synchronization with remote middleware devices.
- **Specify service goals:** The activation of intents requires configuring the behavior of overlay network communication. This is achieved by specifying service goals on devices and their communication.
- **View insights:** This interface is necessary because intent activation and monitoring require access to internal information about the layers below.

Provided interfaces

These interfaces are provided, which are both delegated to the parent *SDON middleware* component (Section 5.2):

- **View insights:** This interface provides internal information regarding the activation and assurance of intents. This includes information such as

what intent results in what device policies and what policies have been successfully applied. Besides this information, the insights provided by the lower layers are exposed as well.

- **Specify intents:** Through this interface, intents specifying the high-level functional requirements of the overlay network are received.

Sub-components

This component has three sub-components: *Intent translator* (Section 5.5.1), *intent activator* (Section 5.5.2), and *intent assurer* (Section 5.5.3).

5.5.1 *Intent translator*

The *intent translator* compiles the received intents into a network model that describes the desired overall overlay network behavior. This model includes the desired nodes, links, resource requirements, service goals, and instructions on which application components should run on each node. The network model is the foundation for matchmaking between overlay nodes and devices.

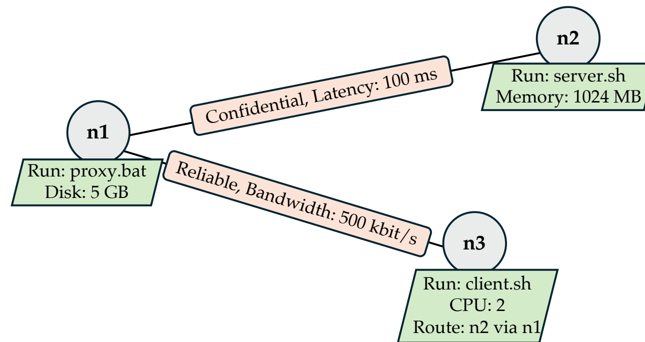


Figure 5.14: Software-defined network model describing a client-server application communicating through a proxy. Intents describe the desired topology, nodes, and links.

To improve the understanding of such a network model, an example of an application consisting of three nodes is shown in Figure 5.14. The model specifies that the program *proxy.bat* is to be executed on a node identified $n1$, the program *client.sh* on node $n2$, and the program *server.sh* on node $n3$. Node $n1$ is required to have at least 5 GB of disk capacity, while node $n2$ must have at least 1024 MB of memory. Node $n3$ is required to have at least 2 CPU cores and should route its messages to $n2$ via $n1$. Additionally, it is specified that nodes $n1$ and $n2$ should be connected by a link that ensures the confidentiality of the communications and maintains a latency not exceeding 100 ms. Nodes $n1$

and $n3$, on the other hand, are to be connected by a link that ensures reliable communications with a minimum bandwidth of 500 kbit/s.

Required interface

- **Provide network model:** An interface is required so that this component can hand over the network model, including all intents.

Provided interface

- **Specify intents:** Through this interface, intents are received from the application. This interface is delegated to the parent *SDON layer*, which is provided to the application (Section 5.2).

5.5.2 *Intent activator*

The *intent activator* uses the received network model to create a list of node-centered intents, as illustrated in Table 5.1. This list specifies which intents exist for each node, serving as a basis for matchmaking between overlay nodes and devices. A distinction is made between two categories of intents:

- **Resource constraints:** Resource constraints describe intents that require specific device capabilities, which cannot be intentionally enforced. For example, an operating system cannot be changed from Linux to Windows to enable support for running a batch program as requested by an application. Similarly, the bandwidth of a link between two nodes cannot be deliberately increased, but it may automatically increase in the future (e.g., due to reduced network congestion, which frees up more bandwidth).

  For this reason, only this category of intents is considered in the resource matchmaking process.

- **Service goals:** Service goals describe intents that can either fully or partially enforced. For example, two devices can implement an encryption protocol to enforce confidential communications, or an error control protocol to manage packet loss by resending lost packets in the edge network.

This component tries to select devices that best match the specified resource constraint intents. Resource matchmaking decisions are made based on the

Table 5.1: List of intents describing network model shown Figure 5.14.

Node	Intent	Value	Resource constraint	Service goal
$n1$	Run program.	`proxy.bat`	✓	
$n1$	Minimum disk size.	5 GB	✓	
$n1$	Confidential communication.	$n2$		✓
$n1$	Maximum latency.	$n2 = 100$ ms	✓	
$n1$	Reliable communication.	$n3$		✓
$n1$	Minimum bandwidth.	$n3 = 500$ kbit/s	✓	
$n2$	Run program.	`server.sh`	✓	
$n2$	Minimum memory size.	1024 MB	✓	
$n2$	Confidential communication.	$n1$		✓
$n2$	Maximum latency.	$n1 = 100$ ms	✓	
$n3$	Run program.	`client.sh`	✓	
$n3$	Minimum CPU count.	2	✓	
$n3$	Route	B via A		✓
$n3$	Reliable communication.	$n1$		✓
$n3$	Minimum bandwidth.	$n1 = 500$ kbit/s	✓	

information available at the moment of the process. Information may be incomplete, change, or become outdated, resulting in a different resource matchmaking process that better aligns with specified intents in the future. To address this, the next component employs a closed-loop mechanism that allows the resource matchmaking process to be rescheduled whenever new information or intents become available. Further, this component can trigger processes that will start to collect metrics needed to better comply with intents (e.g., perform bandwidth tests to find links that align with an intent). This device matchmaking is shown in Figure 5.15. Here, the intents from the client-server application that have already covered (Figure 5.14) are mapped to match available devices best. Overlay node $n1$ is mapped to device D because it is the only system that can run the `proxy.bat` service, which is a Windows batch script, and meets the other constraints. Node $n2$ is mapped to device A because it requires a connection with a latency not exceeding 100 ms to the device used for node $n1$. Node $n3$ is mapped to device C. It is the best remaining device because it is known to have a link to device D, although the exact bandwidth is not (yet) known. This selection may be based on incomplete information, as opportunistic matchmaking assumes that the desired conditions will likely be met.

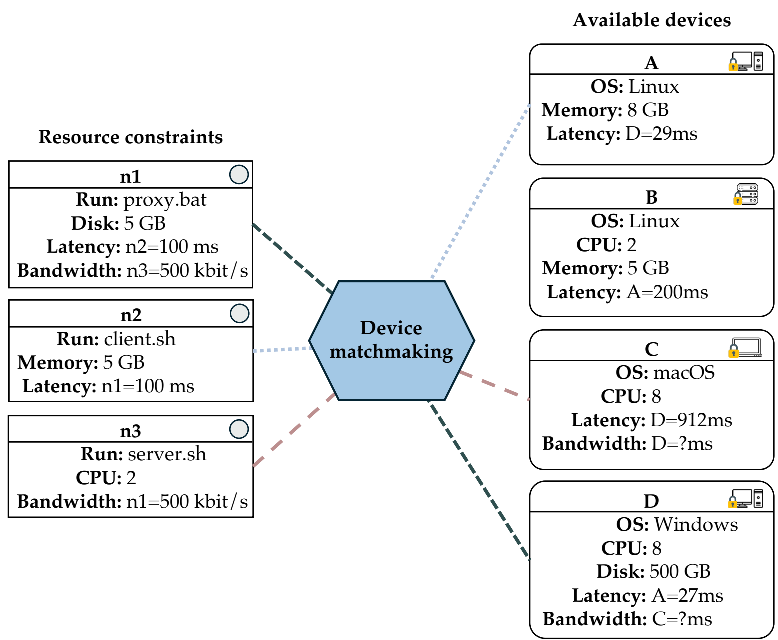


Figure 5.15: Device matchmaking applied by intent activator. Each node is mapped to best matching device.

After matchmaking, device configuration policies are created based on the corresponding intents. These policies are then sent using the communication

overlay to the *intent activator* components of the selected devices, which will then automatically apply the received configuration policies.

Required interfaces

- **Implement monitoring:** This interface is used to inform which intents are activated and which monitoring tasks should be conducted to continuously verify that the overlay network aligns with the expressed intents. The collected monitoring information is sent back using the same interface.
- **View insights:** This interface helps to get lower-level information in activating intents.
- **Specify QoS goals:** This interface is used to configure the local middleware by applying the desired QoS goals on the necessary peers and messages.
- **Send/Read IP packets:** This interface is required to communicate local decisions on resource allocation and desired QoS goals to other middleware devices. Therefore, this interface is also required to receive decisions a remote middleware made about the activation of intents.

Provided interface

- **Provide network model:** This interface is provided to receive the network model, including all intents.

5.5.3 *Intent assurer*

The *intent assurer* ensures that the overlay network remains aligned with the intended state. This is necessary as edge networks are dynamic, and device resources change over time. Further, overlay network deployment may have been conducted using limited information, revealing after some time that a different device matchmaking would result in better overlay network performance. Therefore, this component continuously monitors the overlay network by gathering passive and active analytics. The information to be gathered is directly specified by the intents (e.g., latency intents require monitoring the latency). The gathered information is sent to the *intent activator*, when thresholds are met and if the overlay networks needs to be reconfigured.

Required interfaces

- **View insights:** This interface helps to get lower-level information used to track the deployment progress of intents and monitor the state of successfully applied intents.

Provided interfaces

- **View insights:** This interface provides intent-level insights in addition to the existing insights received that are just passed through. This interface is delegated to the parent *SDON layer*, which is provided to the application (Section 5.2).
- **Implement monitoring:** This interface is needed to make this component aware of the intents that are in the process of being activated and to decide what measures are necessary to collect monitoring information to track intents.

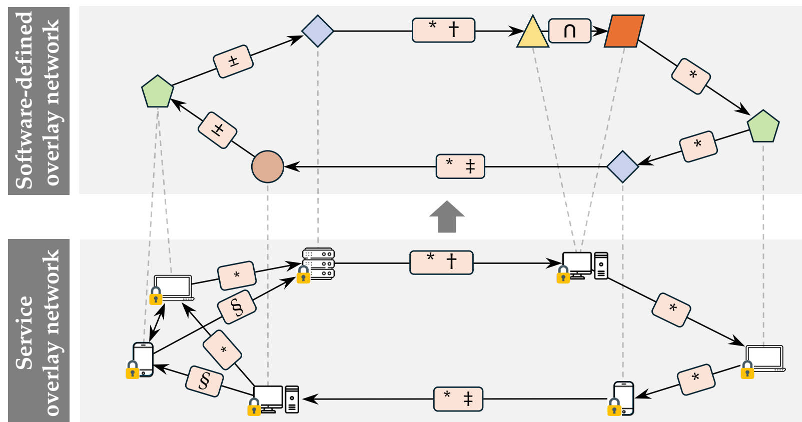


Figure 5.16: Software-defined overlay network. Based on high-level functional application requirements, an overlay network of nodes and links is created implementing the desired behavior.

In summary, the *SDON layer* provides all key system components to achieve IBN using overlay networking technology, facilitating the operation of distributed applications in untrusted environments, as shown in Figure 5.16. This component represents the final layer and last step of the middleware. The intended behavior of nodes and links is specified based on an overlay network model. This specification assists in selecting devices that can meet the requirements to achieve the desired outcome. Additionally, a closed-loop mechanism is applied to keep the overlay network aligned with the intents, responding to changes in application requirements or edge resources.

5.6 CHAPTER SUMMARY

In this chapter, the architecture for a SDON system was summarized, which fulfilled all functional requirements and nonfunctional requirements identified in Chapter 3. The requirements necessitate architecting the SDON system as a distributed middleware, positioned between the application and the edge network. The individual components, their tasks, and interfaces were identified. The SDON middleware includes a communication layer that dynamically and securely links individual edge devices by applying methods known from the P2P domain to overcome connectivity restrictions of edge networks. Further, the middleware includes a service layer, which allows the flexible integration of additional overlay services as needed by the application. These services can provide functionalities like custom routing, forwarding, QoS, or security. Finally, a SDON layer allows the programming of overlay networks using high-level functional requirements (intents), where the middleware then automatically identifies the required means to construct and maintain the desired overlay network. This architecture forms the basis for the prototypical implementation discussed in the next chapter.

6 Implementation of a SDON middleware

This chapter introduces the prototype of a software-defined overlay networking middleware. It implements the architecture developed in chapter 5. The prototype is a fully functional distributed middleware called drasyl. The structure of this chapter is as follows:

- *Section 6.1 gives an overview of the distributed middleware implementation.*
- *Section 6.2 presents the implementation of the communication layer.*
- *Section 6.3 details the implementation of the quality of service layer.*
- *Section 6.4 describes the implementation of the software-defined overlay networking layer.*
- *Section 6.5 summarizes this chapter.*

6.1 THE SDON MIDDLEWARE

The software-defined overlay networking (SDON) middleware has been implemented in Java due to its capability to support a wide range of software and hardware platforms, enabling the inclusion of as many devices as possible in constructing software-defined overlay networks at the edge. Therefore, the middleware is compatible with desktop computers, notebooks, servers running Windows, macOS, Linux, and smartphones with Android operating systems. Overlay networks can be constructed spanning across all these platforms.

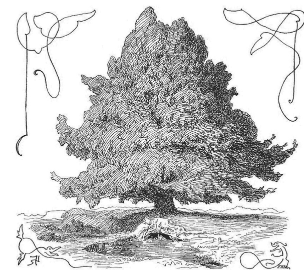


Figure 6.1: Yggdrasil (1895) by Lorenz Frølich.

The middleware implementation is called *drasyl* [dʀazyːl], named after Yggdrasil, the mythical tree from Norse mythology that connects all the realms of the universe [Lin01]. This name reflects the middleware's goal of connecting any devices, regardless of location and context, enabling overlay networking. drasyl was published as an open-source software project under the permissive MIT license to foster further research in this area. This dissertation refers to the version current at the time, which was also published as an open dataset [BR23]. The latest version of drasyl can be found on GitHub: `https://github.com/drasyl/drasyl`.

6.2 COMMUNICATION LAYER

This section presents the implementation of the *communication layer*, whose architecture has been developed in Section 5.3. Beginning with the *identity manager* (Section 6.2.1), where details about the implemented public-key signature system are provided. This is followed by the *peer discoverer* (Section 6.2.2), which describes the protocol implemented for communication overlay network construction. Finally, the *location-aware router* (Section 6.2.3) is presented, describing the mechanism used to verify peer reachability and routing.

6.2.1 *Identity manager*

As outlined in Section 5.3.1, public keys are used instead of IP addresses. Public keys provide uniqueness and security, while IP addresses do not offer these features. The *identity manager* uses an Ed25519 key pair as the device identity. Ed25519 is a high-speed and high-security public-key signature system, that is based on the Edwards-curve Digital Signature Algorithm (EdDSA) [Ber+12]. This signature scheme has been chosen because it best meets the requirements as it provides the following properties:

- **High security:** This signature system provides a 2^{128} security level, offering a similar difficulty to break as a 3000-bit RSA key [Ber+12]. A security level high enough to be used to connect devices securely.
- **Fast performance:** A quad-core 2.4 GHz Intel Westmere processor from the year 2010 can verify 71 000 signatures per second and sign 109 000 messages per second. Key generation happens almost as quickly as signing. [Ber+12]. This makes the signature system usable even on lower-power devices running the middleware.
- **Small keys:** The public key uses just 32 byte, and a signature takes up 64 byte, making them compact compared to other schemes providing a similar security level. This compactness reduces the middleware's overhead of the network protocol, as every middleware message must contain the recipient and sender's public key in its header. Every byte saved in the header increases the available space for application payloads contained in messages.
- **Flexibility:** Ed25519 keys are convertible to X25519 keys for use in key agreement protocols. This allows peers to establish Authenticated encryption with associated data (AEAD)-secured communication from the

first message by knowing only the peer's address, which represents the peer's Ed25519 public key. AEAD encrypts data while also protecting the integrity of both the ciphertext and associated, non-encrypted data, providing confidentiality and protection against unauthorized modifications [Rog02]. Message complexity can be reduced because signaling for agreeing on a key is no longer required, increasing security as communication can be encrypted from the very first message [Tho21].

When the middleware is started, it checks if an identity is present. If not, a new random identity is automatically generated locally and stored to be found on the next start. An example of a middleware's address is shown below:

```
c0900bcfabc493d062ecd293265f571edb70b85313ba4cdda96c9f77163ba62d
```

6.2.2 *Peer discoverer*

This component's architecture section 5.3.2 identifies the need for a custom protocol that efficiently combines NAT traversal and key agreement functionalities. This section presents the implementation of this protocol by first providing an overview and then all protocol messages (Section 6.2.2.1) and processes (Sections 6.2.2.2 and 6.2.2.3). Finally, the protocol is summarized by giving further details on how this protocol establishes the communication overlay.

The protocol is implemented by using UDP as the 4 protocol because UDP offers more flexibility than TCP. The features provided by TCP, such as reliability, flow control, congestion control, and error detection, are not always necessary, which leads to otherwise unnecessary overhead. Additionally, UDP hole punching is less constrained than TCP hole punching (Section 2.4.2.3). In this protocol, the standard UDP hole punching process is enhanced by piggybacking the key agreement on the signaling required for NAT traversal. As a result, the key agreement is completed no later than NAT traversal is finished, saving at least 1 RTTs. A second and third RTT can be saved in some scenarios because signaling traffic can securely transfer application data. While standard UDP hole punching uses only a single rendezvous server and uses it solely for signaling, this new protocol supports multiple rendezvous servers, which can also serve as relays when direct communication is not possible (or if the device does not desire it). Multiple rendezvous servers increase the scalability of the middleware, and using rendezvous servers as relays reduces the additional message complexity needed to contact a dedicated additional server.

6.2.2.1 *Messages*

Each message is AEAD-secured using the stream cipher XChaCha20 for encryption, with Poly1305 authenticator. XChaCha20, a variant of ChaCha20, supports random nonces, which reduces protocol complexity by eliminating the need for stateful nonce tracking and synchronization [Arc20]. The message format builds up on a public header, a private header and a body, as shown in Figure 6.2. The complete message is authenticated. The private header and the body are encrypted and can only be accessed by the recipient.

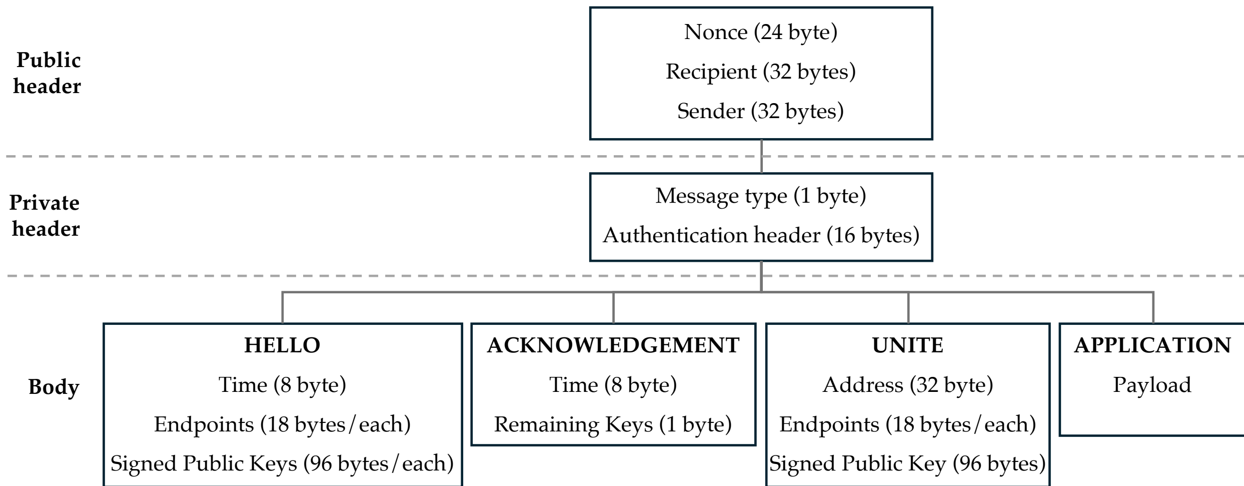


Figure 6.2: The communication layer protocol contains four message types. Each contains a public header, a private header, and a message type-dependent body.

The **public header** includes three fields: A 24-byte uniformly random *Nonce*, used as a message id and for encryption, followed by the *Recipient*'s and *Sender*'s public key, both fields are 32 byte long.

The **private header** contains two fields: A 1-byte long field for the *Message type* and a 16-byte long *Authentication header*.

The **Body** has the following message-type-dependent fields:

- `HELLO`: This message is used to register at a rendezvous server or to check connectivity to a peer.

  It consists of:

  - An 8-byte *Time* field, containing the sender's current time in milliseconds as a Unix timestamp when the message is sent.
  - An *Endpoints* field, listing IP endpoints, each 18 bytes long.

  - A *Signed public keys* field, listing signed public keys, each 96 byte long.

  Only the *Time* field is used for peer communication.

- `ACKNOWLEDGEMENT`: This message acknowledges the receipt of `HELLO` messages.

  It has:

  - An 8-byte *Time* field containing the timestamp from the corresponding `HELLO` message.
  - A 1-byte *Remaining Key* field indicating the number of unused public keys stored at the rendezvous server, only used when the message is sent from a rendezvous server.

- `UNITE`: This message is sent by the rendezvous server to exchange connection information between two middleware devices.

  - A 32-byte *Address* field containing the public key.
  - An *Endpoints* field with a list of IP endpoints, each 18 byte long.
  - A *Signed Public Key* used for key agreement.

- `APPLICATION`: This message is used to send application data and, with an empty payload, to initiate a connection information exchange at a rendezvous server.

  - A variable-length *Payload* field containing application data.

6.2.2.2 *Server registration*

Middleware devices register with a rendezvous server to become available for peer discovery initiated by other devices. Further, the registration is used to place connection information on how the device believes it can be reached, together with signed public keys used for key agreement on the server. While the connection information will help during NAT traversal, the public keys are used for key agreement.

The registration process is described in the UML sequence diagram [OMG11] shown in Figure 6.3. The registration is refreshed periodically as long as the middleware device is running. This is necessary to notify the rendezvous server that the middleware device is still online, to update the connection information stored on the server, to upload additional public keys, and to ensure that any existing middlebox keeps the device reachable.

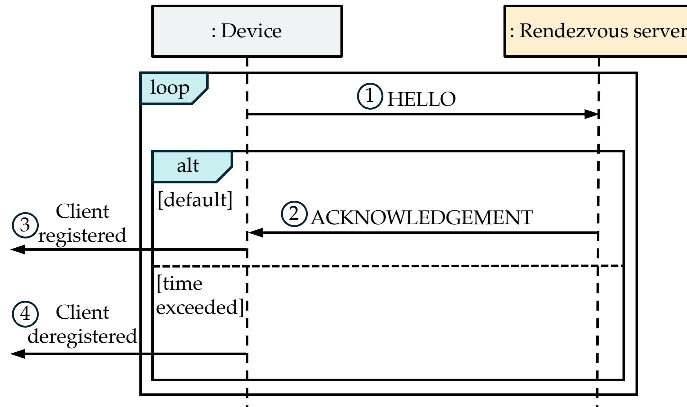


Figure 6.3: Registration process.

- The middleware device sends a ① `HELLO` message to the rendezvous server. The `HELLO` message contains the current timestamp, contact information and public keys.
- The rendezvous server stores the received contact information and public keys. Additionally, the server records the IP endpoint from which it received the middleware device message. Receipt of the message is acknowledged by responding with an ② `ACKNOWLEDGEMENT` message. This response includes the received timestamp.
- Upon receiving the `ACKNOWLEDGEMENT`, the device is successfully registered to the communication overlay ③. The device also learns the RTT to the rendezvous server using the received and current timestamp.
- If no `ACKNOWLEDGEMENT` is received within a given time, the device is no longer registered to the communication overlay ④.

To unregister, the device stops sending new `HELLO` messages, resulting in being considered offline by the rendezvous server after some time. The rendezvous server maintains a list of all currently considered online clients, including their transferred contact information, keys, and IP endpoint the registrations were received from and registration time.

6.2.2.3 *Peer discovery process*

Middleware devices perform peer discovery to establish connectivity using the most local available network path. This process requires both devices to be registered with the rendezvous server. It is shown as a UML sequence diagram in Figure 6.4.

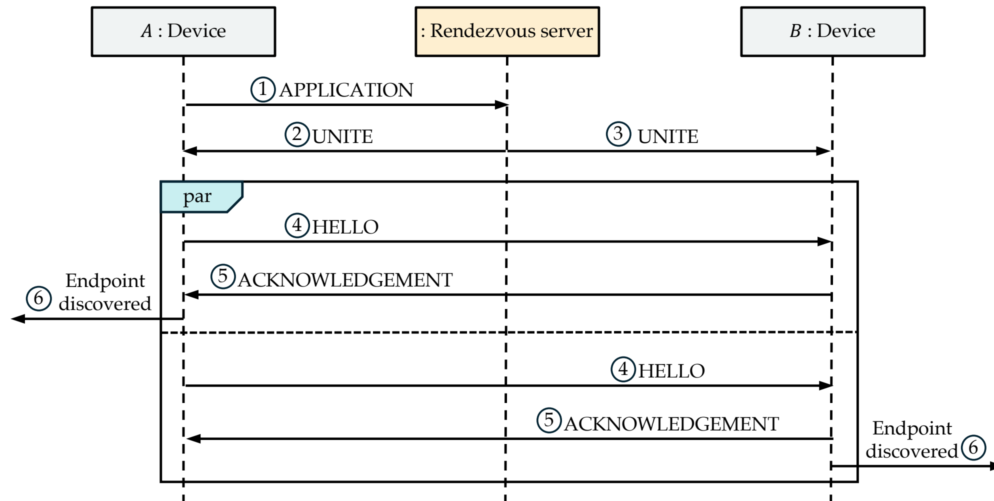


Figure 6.4: Discovery process.

- When middleware device A wants to communicate with middleware device B, it initially lacks information on how to reach B. Therefore, it contacts a rendezvous server to request information to reach B by sending an empty ① `APPLICATION` message addressed to middleware device B.
- The rendezvous server now sends all contact information alongside a signed public key for B to A using a ② `UNITE` message. The server sent A's routing to B in parallel using another ③ `UNITE` message to enable hole punching. Both sent public keys are immediately removed from the server, to not be used for another key agreement process.
- Both devices will check if the received own public key for the key agreement has already been used. Additionally, the received peer's public key is verified to ensure it is signed by the peer. Both devices will check if the public keys received for the key agreement have already been used. If not, both devices can generate their part of the shared secret and try to reach each other in parallel on all endpoints contained in the `UNITE` message by sending ④ `HELLO` messages. On receipt, a middleware device will respond with an ⑤ `ACKNOWLEDGEMENT` message. Once a middleware device gets an acknowledgment, it will know that the endpoint used for the `HELLO` message can be used to reach the other middleware device ⑥. The timestamps in both messages are also used to calculate the RTT.

The augmented protocol requires only four message types and two protocol processes "Server registration" and "Peer discovery". Depending on if one or both middleware devices are behind a middlebox and one of the middleboxes

applies endpoint-dependent mapping, up to three (4) HELLO messages must be sent to "punch" a hole into the middlebox (see Section 2.4.1.2). If hole punching failed, the active relay is used for communication with the peer. In this situation, middleware device *A* periodically repeats the discovery process as it never gives up on establishing a direct connection.

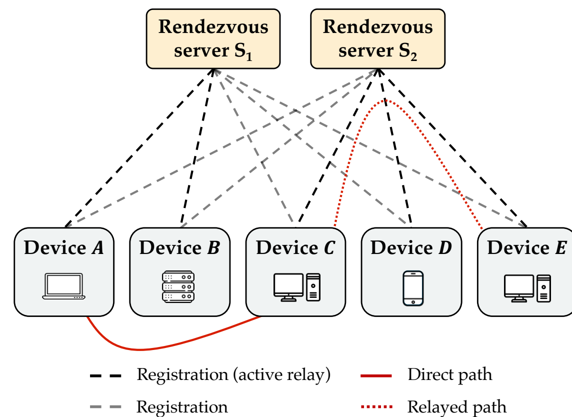


Figure 6.5: Communication overlay established by the communication layer to enable secure discovery of middleware peers.

Figure 6.5 illustrates the overlay network constructed by this protocol. Middleware devices register with at least one rendezvous server and select the one with the shortest latency as the active relay. The active relay is used when direct communication is impossible (like devices *C* and *E*). If hole punching succeeds, a direct connection is possible (like devices *A* and *C*).

6.2.3 *Location-aware router*

This component maintains a table with all known peers and their corresponding IP endpoints, discovered by the *peer discoverer* (Section 5.3.3).

The following factors were considered for the implementation of the IP endpoint health check: middleware devices can leave the communication overlay at any time or may be assigned a new IP address. Therefore, a health check is necessary to verify that the discovered endpoint is still alive. Furthermore, the discovered endpoint might point to a middlebox that has created a temporary mapping (refer to section 2.4 for more information about middleboxes and their mapping behavior). Middleboxes strive to destroy mappings due to inactivity. Therefore, traffic must be sent through this mapping to keep it alive. Thus, a periodic health check is needed to verify that a peer is still reachable, measure the latency, and prevent any middlebox from garbage-collecting the used mapping. A proper time interval must be selected for this as too many

health checks increase the network's overhead. Too few checks will result in the destruction of middlebox mappings and increase the time before a peer can be assumed offline. The challenge with middleboxes is that no assumption can be made regarding when a mapping will be cleared. According to [Hät+10], most middleboxes retain UDP bindings for at least 120 s of inactivity.

The health check is shown as a UML sequence diagram in Figure 6.6 and has been implemented as follows (Figure 6.6):

- Each IP endpoint is periodically checked by sending a ① `HELLO` message including the current time in milliseconds as a Unix timestamp.
- The contacted peer has to response by sending back an ② `ACKNOWLEDGEMENT` including the just received timestamp. As all messages are AEAD-secured the response can be verified and the timestamp is used to calculate the RTT ③.

If no response is received within a given time, the message is considered lost. If no response is received for a series of messages, the endpoint is considered dead and will be removed from the routing table ④. The component assumes an endpoint is finally dead if at least three consecutive responses fail to arrive.

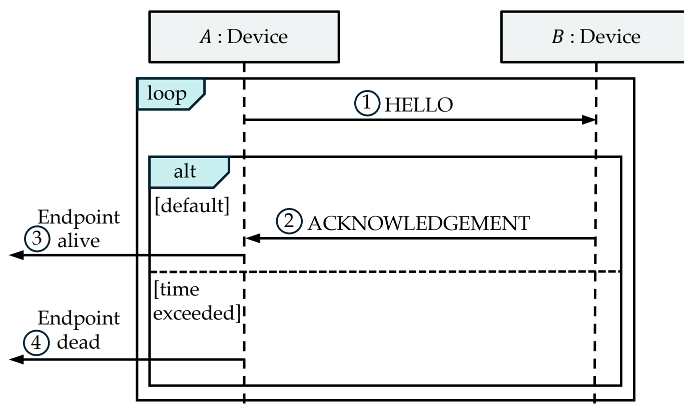


Figure 6.6: Endpoints are checked.

In summary, the communication layer implements a peer discovery protocol creating a communication overlay making all middleware devices routable. Whenever a middlebox device needs to communicate with an other device, the most local discovered network IP endpoint is used for communication. Hole punching is used to provide direct connectivity in most cases, with a fallback to relayed communication.

6.3 SERVICE LAYER

This section presents the presented of the *service layer*, whose architecture has been developed in Section 5.4. The presentation begins with the *service mechanisms library* (Section 6.3.1), which demonstrates the application programming interface (API) that supports the implementation of a wide range of mechanisms. Additionally, a selection of the already implemented mechanisms is presented. Next, the *service goals interpreter* (Section 6.3.2) is introduced, showing how the mapping of service goals to service mechanisms is performed. It also explains how these mapped mechanisms are subsequently used to construct a communication processing pipeline. Finally, the *service implementer* (Section 6.3.3) is discussed, describing how the constructed pipelines are applied and how messages are assigned to the correct pipelines.

6.3.1 *Library of service mechanisms*

First, the *library of service mechanisms* provides an API allowing the implementation of mechanisms to enforce service goals. Second, a collection of already implemented mechanisms is provided, from which the application can select (Section 5.4.1).

6.3.1.1 *Service mechanism API*

Service goals are enforced by mechanisms implemented as message-level interceptors that control how communication is processed between the application and the network.

Interface	Description
`void messageOutbound(Object msg)`	Invoked when an outbound message is passed to the interceptor.
`void messageInbound(Object msg)`	Invoked when an inbound message is passed to the interceptor.
`void pipelineActive()`	Invoked once, when pipeline for peer communication has been created.
`void pipelineInactive()`	Invoked once, when pipeline for peer is about to be terminated.

Table 6.1: `MessageInterceptor` API for implementing mechanisms to enforce service goals.

In drasyl, each interceptor is represented by a Java class that implements the `MessageInterceptor` interface, providing the methods shown in Table 6.1. The `outboundMessage()` and `inboundMessage()` methods alter outbound or inbound messages, while `pipelineActive()` and `pipelineInactive()` perform connection establishment or teardown handshakes. All methods can optionally be implemented, depending on the mechanism to be implemented.

```java
public class EncryptionInterceptor extends MessageInterceptor {
  private final byte[] encryptionKey;
  private final byte[] decryptionKey;

  EncryptionInterceptor(byte[] encryptionKey, byte[] decryptionKey) {
    this.encryptionKey = encryptionKey;
    this.decryptionKey = decryptionKey;
  }

  void outboundMessage(Object msg) {
    if (msg instanceof byte[]) {
      byte[] cyphertext = encrypt(msg, encryptionKey);
      super.outboundMessage(cyphertext);
    } else {
      super.outboundMessage(msg);
    }
  }

  void inboundMessage(Object msg) {
    if (msg instanceof byte[]) {
      byte[] cleartext = decrypt(msg, decryptionKey);
      super.channelRead(cleartext);
    } else {
      super.channelRead(msg);
    }
  }
```

Snippet 6.1: Implementation of a service mechanism that enforces confidentiality using encryption.

An example implementation of an interceptor for encrypting application communication before it is transmitted over the edge network, ensuring confidentiality, is provided in Snippet 6.1. The example interceptor is called `EncryptionInterceptor` and implements the `MessageInterceptor` interface.

The `outboundMessage()` method (line 10) first checks if the outbound message `msg` is a byte array. If `msg` is a byte array, it is encrypted using the provided encryption key. This key could, for example, be provided by a separate interceptor that has performed a key agreement in advance. The encrypted message is then passed to `super.outboundMessage()` to forward it along the pipeline for further processing or transmission (line 13). This enables handling by the

next outbound interceptor in the pipeline, eventually leading to transmission over the network. Using `outboundMessage()` ensures that each interceptor can add processing logic while maintaining a consistent and organized flow of message handling. If `msg` is not a byte array, it is forwarded unaltered (line 15).

Similarly, the method `inboundMessage()` (line 19) checks and decrypts inbound messages using a provided decryption key before passing it further by calling `super.inboundMessage()` (line 22).

6.3.1.2 *Implemented overlay service mechanisms*

Several overlay service mechanisms have been implemented to enforce service goals that are often needed. All these mechanisms have been implemented using the API presented in the previous section.

AUTHENTICATED COMMUNICATION Authentication has been implemented by signing messages with the sender's private Ed25519 key (Section 6.2.1), enabling the verification of the sender's identity and securely ensuring the integrity of the transmitted messages against forgery.

COMPRESSED COMMUNICATION Compression has been implemented by applying the Zstandard (Zstd) lossless data compression algorithm to all messages sent over the network [CK21]. Compression reduces the amount of data transmitted in exchange for CPU time, improving the network efficiency and data throughput.

CHORD-BASED ARRANGEMENT All nodes participating in an overlay network can apply the Chord protocol [Sto+03] to establish a DHT, which can be used for efficient, scalable distribution of keys across nodes in a distributed system. Nodes and keys are arranged in a virtual ring, allowing for quick data retrieval, and dynamic scaling with minimal reorganization.

ENCRYPTION Encryption has been implemented using the XChaCha20 stream cipher with Poly1305 for message authentication. Both peers perform an X25519 key agreement to establish this, generating a shared secret to derive encryption keys.

ERROR CONTROL Error control has been implemented using Automatic Repeat reQuest (ARQ), which ensures in-order delivery of messages and triggers retransmissions when necessary to prevent data corruption or loss. Addition-

ally, checksums are employed to detect accidental errors during transmission, maintaining the integrity of the transmitted messages.

CYCLON ARRANGEMENT All nodes participating in an overlay network can apply the CYCLON protocol [VGS05], which conducts a peer sampling mechanism where each node maintains a continuously updated list of peers. CYCLON achieves this by periodically exchanging partial network views with randomly chosen peers, ensuring robust and dynamic peer sampling.

PRIORITIZATION Prioritization is implemented using a priority queue, where each message is assigned a priority level (based on factors like message type, urgency, or the time it was created). Messages are enqueued based on their priority, with the highest priority messages processed first, ensuring that critical messages are handled before less important ones.

ROUTING Routing has been implemented by allowing the specification of custom relays, enabling overlay messages to be routed through a predefined path of overlay nodes rather than the default direct network route.

In summary, the implemented mechanisms are not intended to be exhaustive in meeting all the needs of applications. However, they demonstrate that this component's API can represent various mechanisms.

6.3.2 *Service goals interpreter*

The *quality of services (QoSs) goal interpreter* uses the implemented service mechanisms to enforce service goals specified by the application. It creates processing pipelines, including one or multiple service mechanisms. (Section 5.4.2). First, the available service goals are presented. Next, the mapping of these goals to mechanisms is explained. Finally, the interface for implementing processing pipelines is presented.

6.3.2.1 *Service goals*

Service goals describe the desired outcomes. To illustrate this function, goals are selected based on two criteria: (1) goals providing the most desired features by applications, (2) goals that describe different dimensions, goals describing overlay links between node pairs, goals describing paths between groups of

nodes, and goals describing the overlay topology. This section presents goals that can be enforced using the available mechanisms (Section 6.3.1.2).

RELIABILITY Communication in unreliable environments is inherently unpredictable. While some applications can tolerate message loss, others require reliable data transfer to ensure guaranteed delivery and error-free reception.

INTEGRITY Integrity ensures that transmission data is protected from unauthorized alterations. While reliability addresses errors that occur accidentally during transmission, integrity focuses on preventing intentional attacks such as tampering or data corruption. This is needed by applications where data authenticity and accuracy are essential.

CONFIDENTIALITY Confidentiality ensures that transmission data is accessible only to authorized parties, protecting it from eavesdropping. This is needed to maintain privacy and when the application handles sensitive information, such as personal information. If this confidentiality flag is set, the system uses symmetric encryption to ensure data can only be decrypted by the receiver having the corresponding decryption key, even when relayed through a rendezvous server.

EFFICIENCY Communication has a cost, in terms of required communication resources. This means the network between two devices has only a limited bandwidth, determining how much data can be transmitted over time. Increasing efficiency is essential when applications are operated in constrained environments with limited resources.

PRIVACY The privacy goal ensures that sensitive data is handled with strict access control, allowing users to determine who can access their information. Additionally, it is essential to protect not only the content of the communication but also the identities of the communicating parties, ensuring that both the data and the interaction remain confidential.

DISTRIBUTED LOOKUP Distributed lookup aims to efficiently locate and retrieve data or resources across multiple nodes, ensuring scalability, low latency, fault tolerance, and balanced query load while minimizing network traffic and maintaining data consistency.

LOAD BALANCING The goal of load balancing is to evenly distribute workload across multiple resources, ensuring optimal performance, preventing overload, minimizing latency, maximizing resource utilization, and enhancing system reliability and fault tolerance.

6.3.2.2 *Service goals mapping*

While goals specify the desired outcome, mechanisms are needed to enforce them. Therefore, goals must be mapped to mechanisms. Table 6.2 shows possible strategies to enforce each service goal by one or multiple service mechanisms.

		Service mechanism							
		Authentication	Compression	Chord arrang.	Encryption	Error control	CYCLON arrang.	Prioritization	Routing
Service goal	Reliability					✓			(✓)
	Integrity	✓			(✓)	(✓)			
	Confidentiality				✓				
	Efficiency		✓					(✓)	✓
	Privacy				✓				✓
	Distributed lookup			✓					(✓)
	Load balancing			✓			✓		(✓)

✓ := fully enforced (✓) := cond. enforced (blank) := not enforced

Table 6.2: Mapping between service goals and mechanisms. The service layer in the drasyl middleware selects a single mechanism or a combination of mechanisms to enforce one goal.

6.3.2.3 *Pipeline configuration*

Once service goals have been mapped to mechanisms, the processing pipeline is created. The pipeline consists of a chain of one more multiple `MessageInterceptors` and itself is called `ProcessingPipeline`.

Snippet 6.2 shows the exemplary initialization of a pipeline using the provided `ProcessingPipeline` interface. This pipeline enforces the goals *reliability* and *confidentiality* by using *error control* and *encryption* mechanisms (lines 3 to 4).

Snippet 6.2: Creation of a processing pipeline that enforces service goals reliability and confidentiality.

```
class ReliableAndConfidentialPipeline extends
    ProcessingPipelineInitializer {
    void initChannel(Pipeline p) {
        p.addLast(new EncryptionHandler());
        p.addLast(new ErrorControlHandler());
    }
}
```

6.3.3 *Service implementer*

The *service implementer* ensures that every incoming and outgoing message passes through the respective processing pipeline. A separate pipeline is maintained for each peer. If a message is sent to or received from a peer without an existing pipeline, one is implicitly created based on the service goals. Regardless of the processing performed by the pipeline, each message is formatted as an `APPLICATION` message with the appropriate public, private header, and body (see Section 6.2.2.1).

A virtual network interface is established on each edge device to implement the "Send/Read IP packets" interface. This virtual interface can be used as any other interface connecting a physical network. By creating such a TUN interface [Bor23b], IP applications can communicate transparently through the overlay network without modification. While Linux and macOS provide native support for TUN devices, Windows requires the use of Wintun[1].

1 https://www.wintun.net/

Configured with the IP address and netmask specified in the network model, the TUN interface ensures correct network addressing. The SDON middleware receives IP packets sent to this interface, processes them according to the overlay network configuration, and transmits them to the appropriate destination SDON device. Conversely, packets received via the overlay network are delivered to the local application through the TUN interface.

6.4 SDON LAYER

This section presents the implementation of the *overlay layer*, which has been architected in Section 5.5. The introduction begins with the *intent translator* (Section 6.4.1), which provides a Lua-based programming interface for creating overlay network models that incorporate the intents specified by an application. Lua is a lightweight, high-level scripting language specifically designed for

embedded use in applications [IFF96]. Next, the *intent activator* (Section 6.4.2) is introduced, showcasing the SDON controller that provides logically centralized software control of the underlying edge resources being programmed. The controller's greedy-based algorithm for selecting the best currently available resources for activating the overlay intents is demonstrated. Finally, the *intent assurer* (Section 6.4.3) is discussed, highlighting the closed-loop mechanism used to verify that the overlay aligns with the desired intents in the presence of changing resources.

6.4.1 *Intent translator*

Intents submitted to the system cannot be directly used to configure the underlying edge resources. Therefore, intents need to be converted into a network intent. The *intent translator* defines the network model and API for submitting intents to the system. (Section 5.5.1).

The goal is to describe a network with virtual nodes and virtual edges. Both the network as well as the nodes and edges can be assigned intents. Since the system is designed to be extensible, it must allow for adding new properties.

6.4.1.1 *Network model*

Following the requirements mentioned above, the data model has been created and is presented as an entity-relationship (ER) model [Che76] in Figure 6.7. The "root" of each network model is the *network* entity, which can contain any number of *network intents* and *nodes*. Links have a start and an end, connecting two nodes. Nodes and links can each have any number of associated node intents or link intents. Now, the available node, link, and network intents are presented, with the flexibility to extend them as needed. Intents either result in influencing the resource allocation progress applied by the intent activator (Section 6.4.2) or the service goals being enforced provided by the service goals interpreter (Section 6.3.2.1).

NODE INTENTS These node intents are available to be used individually or in combination.

- `cpu`: The minimal number of CPU cores. Applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).
- `memory`: The minimum amount of memory in MB. Applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).

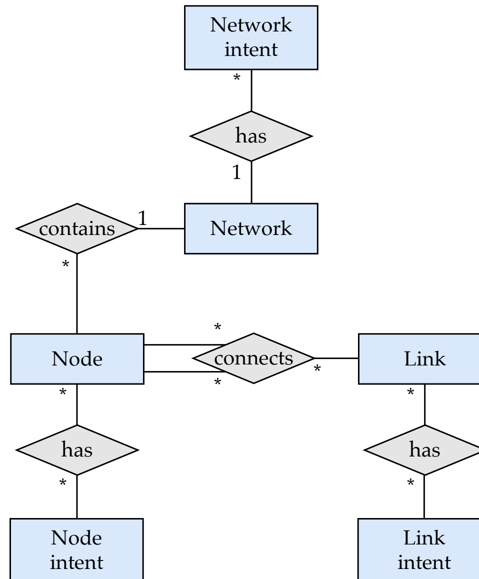


Figure 6.7: Network model describing intent-based overlay networks as an ER model.

- `disk`: The minimum disk size in GB. Applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).
- `privacy`: Enables encryption. Optionally, a list of routes can be supplied to control which devices see communication. Enforced by the *service goals interpreter* (Section 6.3.2.1).
- `distributed_lookup`: Enables node to perform distributed lookup protocol. Enforced by the *service goals interpreter* (Section 6.3.2.1).
- `load_balancing`: Enables node to perform balancing protocol. Enforced by the *service goals interpreter* (Section 6.3.2.1).

LINK INTENTS The following link intents can be used individually or in combination:

- `reliability`: Enables reliable communication if set to `true`. Enforced by the *service goals interpreter* (Section 6.3.2.1).
- `integrity`: Ensures communication integrity when set to `true`. Enforced by the *service goals interpreter* (Section 6.3.2.1).
- `confidentiality`: Ensures communication confidentiality when set to `true`. Enforced by the *service goals interpreter* (Section 6.3.2.1).

- `efficiency`: If set to `bandwidth`, communication is optimized to minimize bandwidth usage. If set to `latency`, communication is optimized to minimize transmission latency Enforced by the *service goals interpreter* (Section 6.3.2.1).
- `latency`: The maximum latency in ms. Applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).
- `jitter`: The maximum jitter in ms. This parameter is applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).
- `loss`: The percentage of packet loss, expressed as a percentage (%). This parameter is applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).
- `bandwidth`: The minimum bandwidth in kbit/s. Applied by the *intent activator* to influence the resource allocation process (Section 6.4.2).

NETWORK INTENTS Both node and link intents can also be enforced on the network level, activating them on every node and link if not locally overwritten. Furthermore, additional network intents control the proactive collection of edge device and network capabilities:

- `measure_latency`: Enables latency measurements between SDON devices. Each device periodically performs latency measurements with randomly selected other devices. The results of these measurements are collected at the SDON controller and are used to optimize the overlay network.
- `measure_cpu`: Enables one-time CPU benchmarks on all SDON devices to measure the CPU performance. The benchmark includes searching for prime number within a specified interval. The results of these measurements are collected at the SDON controller and are used to optimize the overlay network.

6.4.1.2 *Network modeling API*

An overlay network scripting API has been created using Lua to create network models. Lua is language-agnostic, making integrating with many different programming environments easy. This decouples the model from the concrete implementation.

Function	Description
`create_network(intents)`	Creates a new network model with specified intents.
`network:add_node(node_id, intents)`	Adds a new node to the network model with specified identifier and intents.
`network:remove_node(node_id)`	Removes node with specified identifier from the network model.
`network:clear_nodes()`	Removes all nodes from the network model.
`network:add_link(node1, node2, intents)`	Adds a new link to the network model between specified nodes and with specified intents.
`network:remove_link(node1, node2)`	Removes link between specified nodes from the network model.
`network:clear_links()`	Removes all links from the network model.
`network:get_nodes()`	Returns list of all nodes of the network model.
`network:get_links()`	Returns list of all links of the network model.
`network:set_callback(callback)`	Sets a callback function that is called every time new insights have become available.
`node:get_insights()`	Returns list of all insights of the node.
`link:get_insights()`	Returns list of all insights of the link.

Table 6.3: Lua-API describing desired overlay networks.

Table 6.3 shows the Lua interfaces. First, a network model is created by calling `create_network()`. Then, nodes and links with the desired properties are added to the model using `add_node()` and `add_link()`, respectively. Further, a callback can be defined using the `set_callback()` function. That callback is called every time a new insight becomes available, allowing the network model to change in response to changes in the underlying device resources. This callback can alter the network model in response to changes in the edge resources or to implement intents that are otherwise impossible with the available options.

Snippet 6.3 presents an example of using the modeling API to create the overlay network shown in Figure 5.14. A conditional intent has been added using the callback functionality (lines 7 to 17): If latency of link $A \leftrightarrow B$ exceed latency of 100 ms, traffic from node C to B is routed through a direct link instead of a path via A. If latency is below 100 ms, traffic via A.

```
network = create_network()
network:add_node("A", {disk = "5G"})
network:add_node("B", {memory = "1024M"})
network:add_node("C", {cpu = 2, routes = {B = "A"}})
network:add_link("A", "B", {encrypted = true, latency = "100m"})
network:add_link("A", "C", {reliable = true, bandwidth = "500k"})
function my_callback(network, devices)
  linkAC = network:get_link("A", "C")
  nodeC = network:get_node("C")
  if get_insights(linkAC)["latency"] > "100m" then
    network:add_link("C", "B")
    nodeC.routes[B] = "B"
  else
    nodeC.routes[B] = "A"
    network:remove_link("C", "B")
  end
end
network:set_callback(my_callback)
```

Snippet 6.3: Example usage of network modeling API describing network shown in Figure 5.14.

6.4.2 *Intent activator*

The *intent activator* allocates underlying edge resources and configures nodes and links to activate the network model (Section 5.5.2).

A centralized SDON controller supervises the task of activating intents by first asses available resources, which includes the application of reactive/proactive means to collect information on the available resources. All other nodes are middleware devices that register to one SDON controller forming a system as shown in Figure 6.8 using the protocol shown in Figure 6.9. Through registration, they offer resources to the controller, who can use these to create desired overlay networks and deploy application components.

The protocol shown relies on the virtual overlay network provided by the communication layer, enabling connectivity between the controller and all nodes. Further, all SDON overlay protocol messages are processed by pipelines, enforcing confidentiality, integrity, and reliability.

Multiple controllers can be added to the system, distributing the load of managing multiple applications among them. However, each application and each edge device is always managed by one controller at a time. An edge device transfer between controllers is possible. One controller with excess resources can hand over devices to a controller that needs more edge resources.

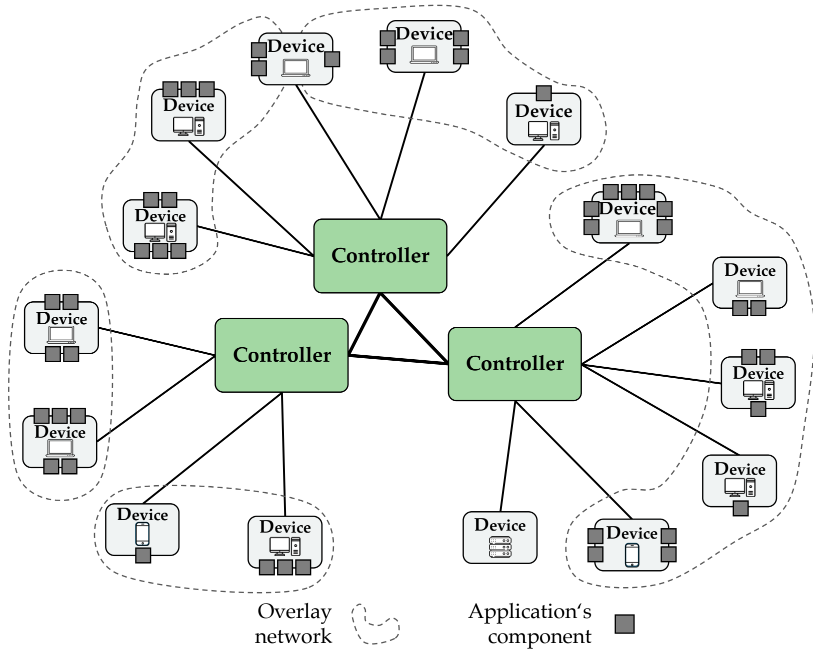


Figure 6.8: SDON overlay network. Devices register with a controller and offer resources to be part of an overlay where application components are deployed on the individual nodes.

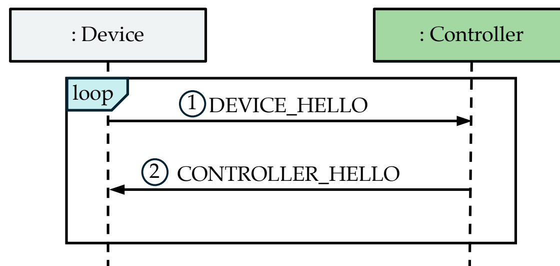


Figure 6.9: SDON layer protocol. Nodes (re)register periodically with a controller, to which the controller replies.

6.4.2.1 *Resource allocation algorithm*

Algorithm 1 Matchmaking of overlay nodes to SDON devices.

```
1:  function MATCHMAKING(N, D)
2:      M ← ∅
3:      for all n ∈ N do
4:          best_match ← ∅
5:          min_distance ← ∞
6:          for all d ∈ D do
7:              distance ← |CURRENT_STATE(d) − DESIRED_STATE(n)|
8:              if distance < min_distance then
9:                  min_distance ← distance
10:                 best_match ← d
11:             end if
12:         end for
13:         M ← M ∪ {(n, best_match)}
14:     end for
15:     return M
16: end function
```

A greedy algorithm selects the best device resources based on the available knowledge of the resources during allocation. The algorithm is implemented as follows: For each node $n \in N$, with N being the set of all nodes in the model, the desired state d_n is defined. The matchmaking process is outlined in Algorithm 1. Matchmaking requires traversing all overlay nodes N (line 3). Each node's and its connected links' requirements are compared with available SDON devices D to find the most suitable device $d \in D$ (line 6). For each node $n \in N$, the desired state of n is compared with the current state of d and then n is mapped to the d closed to the desired state (lines 7 to 11). At the end, the matchmaking returns a list of mappings M containing pairs (n, d), each mapping a node n to a device d (line 15).

The distance between the two states is calculated as follows. Each state is represented as a vector $s = (p_1, p_2, \dots, p_n)$, where each component p_i represents a resource property to be considered. The vectors are subtracted component-wise to quantify the distance between the desired state s_d and the current state s_c. This results in a difference vector $\Delta s = s_d - s_c$. Each component is weighted by a corresponding priority vector $w = (w_1, w_2, \dots, w_n)$ to enable prioritization among the resource properties. The weighted distance is then calculated by

multiplying the difference vector component-wise with the priority vector, resulting in:

$$\Delta s_{\text{weighted}} = w \circ \Delta s = (w_1 \cdot (s_{d1} - s_{c1}),\ w_2 \cdot (s_{d2} - s_{c2}),\ \dots,\ w_n \cdot (s_{dn} - s_{cn}))$$

This allows certain resource properties to have a greater influence on the overall distance measurement (e.g., when latency is more important than jitter).

6.4.2.2 *Configuration distribution*

In this step, the controller distributes the overlay node intents to the assigned SDON devices. This is achieved using the ① `CONTROLLER_HELLO` message seen in Figure 6.9, where the list of desired intents is attached to.

6.4.3 *Intent assurer*

The SDON controller supervises the provision and ongoing application of distributed intents by periodically receiving `HELLO_NODE` messages containing information about the node as well as the currently applied intents. These periodic updates potentially alter the current state of each node, c_m, which may render the existing resource allocation suboptimal. To address this, the controller regularly executes the algorithm described in the previous section to ensure that the overlay utilizes resources optimally and, if necessary, automatically reconfigures the overlay.

6.5 CHAPTER SUMMARY

This section presented the implementation of a distributed middleware for software-based overlay network programming.

First, the implementation of the communication layer was presented. A novel network protocol was introduced, which efficiently combined multiple NAT traversal techniques with a Diffie-Hellman key agreement, allowing edge devices to establish secure links in minimal time. This was used to construct an IP overlay, which creates the illusion of an unrestricted network for the application.

Second, a service layer was introduced that allowed the flexible integration of overlay services. This was implemented by applying the interceptor design pattern to all communication processed by the overlay. Individual interceptors can alter overlay behavior to integrate desired services without interfering with

other service interceptors. Services can be limited to specific overlay elements and can be changed during runtime. These services include functionalities like custom routing, forwarding, QoS, or security.

Third, the SDON layer was introduced, which provided global visibility and control of the overlay by introducing a network controller. The controller provides a Lua scripting API where application developers can program desired overlay networks by specifying the desired number of nodes, the topology, and node and link behavior. The presented controller automatically identifies the required steps and most suitable edge devices to create the desired overlay network. The controller supervises the ongoing alignment of the current overlay network state with the desired state by monitoring all edge devices. A closed-loop mechanism allows the automatic detection and repair of any unintended network change in response to unpredicted changes of edge resources.

7 Evaluation

The prototype of a software-defined overlay networking system presented in Chapter 6 is evaluated through several experiments, each focusing on one of the three layers: communication, dynamic overlay services, and software-defined overlay networking. Although each experiment targeted a specific layer, the lower layers were also implicitly tested. The evaluations are performed to demonstrate that the prototype meets the specified requirements. The same middleware is used throughout all evaluations, and it is successively extended with new functionalities, which then become the focus of each respective evaluation. Therefore, the experiments demonstrate how the middleware can be applied across a variety of use cases. This chapter is structured as follows:

- *Section 7.1 describes the evaluation of the protocol that enables secure connectivity between devices in restricted networks.*
- *Section 7.2 shows the performance evaluation of edge offloading where the middleware was integrated into a pre-existing computation offloading application.*
- *Section 7.3 presents the evaluation of two edge computing applications using the middleware to simplify edge deployment and centrally-controlled overlay network optimization.*

7.1 EVALUATION OF THE SECURE CONNECTIVITY PROTOCOL

In this section, the protocol used for achieving secure connectivity in restricted edge networks is evaluated. The evaluation setup is described first (Section 7.1.1), followed by the metrics (Section 7.1.2), the results (Section 7.1.3) and a discussion of the findings (Section 7.1.4). Parts of this section are based on the publication [BRF23a].

The evaluation demonstrates that the protocol implemented in the communication layer can securely connect devices in heterogeneous network environments. Therefore, the proposed protocol has been compared with the state of the art to show the expected 2 RTTs reduction in connection establishment time. Additionally, it is shown that this approach is compatible with actual applications and transparently provides secure communication channels. Further, with this experiment, we address the following research questions:

> **RQ 1.1** How to achieve connectivity between edge devices efficiently?

> **RQ 1.2** How to ensure that edge devices are linked securely?

7.1.1 *Experimental setup*

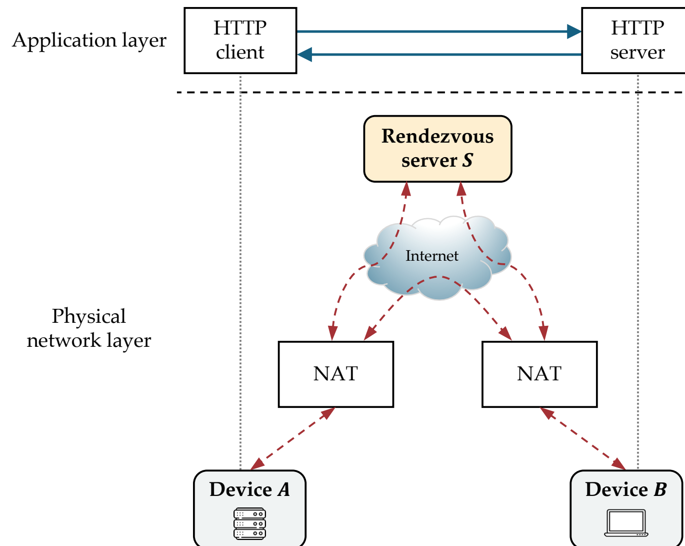


Figure 7.1: Application for communication layer evaluation. The application consists of a HTTP-client on device A communicating and a HTTP-server on device B. On the underlay, both nodes are connected through NATs with the Internet. The middleware transparently provides secure reachability between client and server.

The setup used for this experiment is shown in Figure 7.1. In this evaluation, the middleware runs a client-server HTTP application consisting of one client and one server. In this application, the client wants to send a single request to the server and receive the response. The HTTP client runs on device A, and the HTTP server runs on device B. To prove NAT traversal capabilities, each device is operated behind a NAT. Therefore, before the device A can securely reach the device B, the overlay network must perform NAT traversal and key agreement. Rendezvous server S, a public Internet host, handles the signaling required for hole punching and key agreement. In this setup, both devices are registered with the rendezvous server and have not yet established a direct connection.

An early snapshot of the middleware prototype has been used for this evaluation using a TLS-based key agreement (Section 2.5). TLS supports two distinct key agreement modes: The first one is used when peers communicate for the first time. This handshake mode is called "full handshake". The second mode

uses key material received through an earlier handshake or a separate channel, which is known as the "0-RTT handshake". The prototype used for this experiment has been published [BRF23b].

This experiment is conducted as a real-world experiment, where both devices and the rendezvous server are deployed on actual hardware within the Internet (Figure 7.2). Device A is located in Helsinki, Finland; device in B in Ashburn, Virginia, USA; and rendezvous server S in Nuremberg, Germany.

Figure 7.2: Location of server and devices of communication layer evaluation. Device A is located in Helsinki, Finland; device B is in Ashburn, Virginia, USA; and the rendezvous server S is in Nuremberg, Germany.

To demonstrate that the proposed approach is more effective than existing approaches, the client-server application establishes a connection once using the communication layer, with both devices and the server implementing the communication layer protocol, resulting in a piggybacked TLS handshake. The same experimental setup is used to perform hole punching and key agreement as conducted by previous approaches. In previous approaches, the TLS handshake is conducted subsequently after the hole punching process is completed. These two scenarios are repeated for both TLS handshake modes to compare how the system behaves during initial and successive connection establishments. Therefore, four real-world experiments are conducted, comparing the proposed approach and existing approaches for different TLS handshake modes:

- $E1$ Hole punching with subsequent full TLS handshake (state of the art).
- $E2$ Hole punching with piggybacked full TLS handshake.
- $E3$ Hole punching with subsequent 0-RTT handshake (state of the art).
- $E4$ Hole punching with piggybacked 0-RTT handshake.

7.1.2 *Experimental metrics*

This experiment aims to demonstrate that the proposed approach is effective and performs faster than existing methods when two peers want to communicate with each other. To achieve this, the experiment first examines whether a connection between the two peers can be established. Next, it measures the number of RTTs required before the time to first application byte sent (TTFBS). Additionally, the time elapsed from the client's connection request to the reception of the first byte is recorded. These metrics comprehensively evaluate the proposed approach's connectivity success rate and communication efficiency. Each of the four mentioned experiments (Section 7.1.1) was repeated $n = 1000$ times to obtain a reliable mean value for the TTFBS.

7.1.3 *Results*

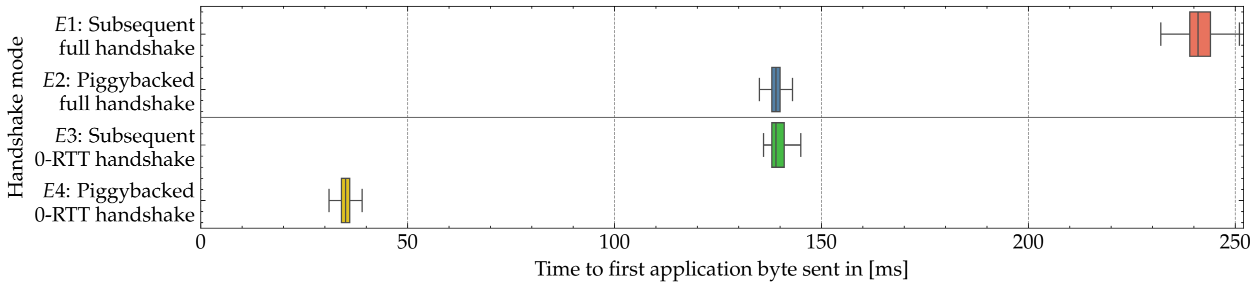


Figure 7.3: Distribution of the time to first application byte sent for $n = 1000$ experiments until two software-defined overlay networking (SDON) devices can start to communicate securely after a connection has been requested. Broken down into the two available TLS handshake modes and whether the handshake was done subsequently or piggybacked.

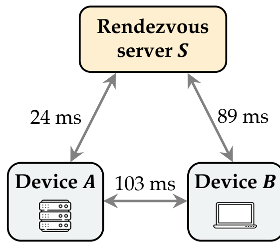


Figure 7.4: Observed median ping times between server and devices during experiment.

During the experiments, the following median ping times between the server and devices were observed (Figure 7.4):

$$A \leftrightarrow B\colon 103\,\text{ms} \quad A \leftrightarrow S\colon 24\,\text{ms} \quad B \leftrightarrow S\colon 89\,\text{ms}$$

Figure 7.3 plots the median time to the first application byte sent in dependence on the conducted handshake mode:

*E*1 A median TTFBS of 241 ms was observed for hole punching with a subsequent full TLS handshake. This result is consistent with the expected value, as the total handshake time can be derived by adding the ping times: *A* experiences 1 RTT with *S* and 2 RTTs with *B* ($24\,\text{ms} + 2 \times 103\,\text{ms} \approx 230\,\text{ms}$). The value observed sets the baseline for full TLS handshakes.

*E*2 Hole punching with a piggybacked full TLS handshake resulted in a median TTFBS of 139 ms, which is 102 ms less compared to *E*1. The observed value comprises 1 RTT between *A* and *S* and 1 RTT between *A* and *B*, corresponding to the expectation. This experiment demonstrates that 1 RTT can be saved during a full TLS handshake.

*E*3 Subsequent hole punching followed by a 0-RTT handshake results in a median time of 139 ms. This observation matches the previous experiment, with *A* having 1 RTT to *S* and another 1 RTT to *B*. The observed value sets the baseline for zero round-trip time (0-RTT) TLS handshakes.

*E*4 Hole punching with a piggybacked 0-RTT TLS handshake results in a TTFBS of 35 ms. Data transmission can begin after 1 RTT between *A* and *S*, aligning with expectations. These results indicate that saving 1 RTT can reduce the connection establishment time by 134 ms.

The experiment results confirm statements regarding general functionality and reduced peer discovery time by 1 to 2 RTTs. Since the TLS handshake is implemented similarly in other protocols, such as QUIC or Datagram Transport Layer Security (DTLS), similar results are expected. The results of this minimal setup can also be applied without loss of generality to large-scale middleware deployments, as the actual connection setup process between devices remains consistent.

7.1.4 *Discussion of results and evaluation conclusion*

This section discusses various aspects of the proposed approach and concludes with a summary.

handshake optimization Further handshake optimizations are technically impossible with current NAT behavior. All TLS handshake messages are already piggybacked, and no further optimization of the hole punching process is feasible.

tls security In this approach, the `ClientHello` message of the TLS handshake is relayed through the rendezvous server *S*, which might be seen as a potential security concern. However, TLS is explicitly designed to handle message transmission through potentially untrusted intermediaries, ensuring that security remains unaffected.

appending application data Following the completion of the TLS handshake, the augmented hole punching process allows for the transmission of encrypted application data via subsequent hole punching messages, thereby reducing connection establishment time. While this capability is technically possible with previous hole punching methods, it is generally discouraged due to the risk of data being sent to incorrect hosts during the hole punching process. Existing methods, which do not incorporate a TLS handshake, risk leaking unencrypted data if messages are sent to incorrect hosts. This issue arises when `ReachabilityChecks` are addressed to private endpoints obtained from *S*. Since many LANs use the same private address spaces, a `ReachabilityCheck` from *A* might reach an incorrect host within *A*'s private network, which shares the same private IP address as *B*.

AMPLIFICATION ATTACKS Existing hole punching methods are susceptible to amplification attacks. In such an attack, a malicious rendezvous server *S* could send *A* a large list of incorrect endpoints for *B*, causing *A* to send repeated `ConnectionRequests` to all these endpoints until a timeout or retry limit is reached. This attack can be mitigated in the proposed approach if TLS is configured to use a public key cipher, which is commonly the case. A public key cipher enables both peers to sign their endpoint lists before sending them to *S* for registration, allowing *A* to verify the `ForwardEndpoints` message upon receipt.

MAN-IN-THE-MIDDLE ATTACKS Man-in-the-middle attacks during the hole punching process are also possible, where malicious hole punching messages are injected. For instance, *B* might receive a `ForwardEndpoints` message with endpoints pointing to a malicious client *C*, resulting in *B* performing hole punching with *C* instead of *A*. This situation would create a hole in *B*'s NAT, exposing *B* to direct attacks from *C*. The proposed approach can mitigate this risk by configuring TLS to require client certificates, allowing *B* to authenticate *A*'s identity when receiving the `ClientHello` message. Another potential attack involves injecting `Acknowledgments` for non-functional endpoints, causing a client to lock into an endpoint and preventing communication. This attack can be countered using the piggybacked 0-RTT handshake, which allows authentication of all `Acknowledgments`.

In summary, the presented approach saves between 1 to 2 RTTs compared to existing methods. A full TLS handshake saves 1 RTT, while all subsequent 0-RTT TLS handshakes save 1 to 2 RTTs. This approach is designed for use with protocols such as QUIC and DTLS 1.3, which rely on UDP and use TLS-based handshakes for security. Particular benefits are seen for applications where the overlay network frequently needs to link new edge devices. Additionally, the hole punching process is secured through the piggybacked TLS handshake.

7.2 EDGE COMPUTING EVALUATION USING THE MIDDLEWARE

In this section, a performance evaluation of edge offloading is presented. The middleware is integrated into a pre-existing computation offloading system, enabling execution on the edge. This section begins with a description of the evaluation setup (Section 7.2.1), followed by the evaluation metrics (Section 7.2.2), and concludes with the evaluation results (Section 7.2.3). Parts of this section are based on the publication [Bor+23].

The evaluation shows that a pre-existing application that uses TCP and UDP for communication in unrestricted networks can be operated in restricted edge networks through an IP overlay network provided by the service layer. This overlay integration simplifies conducting a performance evaluation of various edge computing scenarios, where the service layer constructs the required overlay for each scenario. Additionally, it shows the successful creation of overlays across various environments, including edge, cloud, grid, residential, and mixed settings. With this experiment, we address the following research questions:

> **RQ 2.1** How to integrate overlay networks into applications without imposing restrictions on application design and functionality?

> **RQ 2.2** How to improve the use of heterogeneous edge resources through dynamic overlay networks?

7.2.1 *Experimental setup*

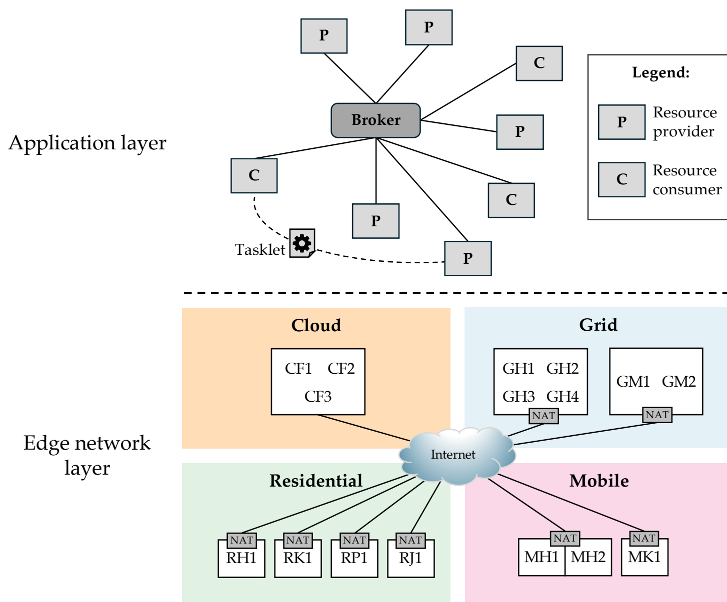


Figure 7.5: Application for service layer evaluation.

In this evaluation, the middleware runs the Tasklet computation-offloading system (Section 2.6) as shown in Figure 7.5.

Edge computing environments vary widely. An edge computing application is considered with heterogeneous devices that are connected differently across several scenarios. Different overlay service goals are applied in the individual experiments. Furthermore, this experiment demonstrates that existing software can operate with the middleware without modifications, thereby gaining the advantages provided by the middleware and improving application performance. This evaluation demonstrates how the middleware supports the execution of scientific experiments. Additionally, it is shown how various network environments impact communication and, consequently, the performance of edge computing. This experiment is conducted as a real-world experiment with an actual implementation.

An overview of the devices (Section 7.2.1.1), environments that are integrated with the middleware in different scenarios (Section 7.2.1.2), and computation tasks (Section 7.2.1.3) is provided in the following.

7.2.1.1 *Devices involved in experiment*

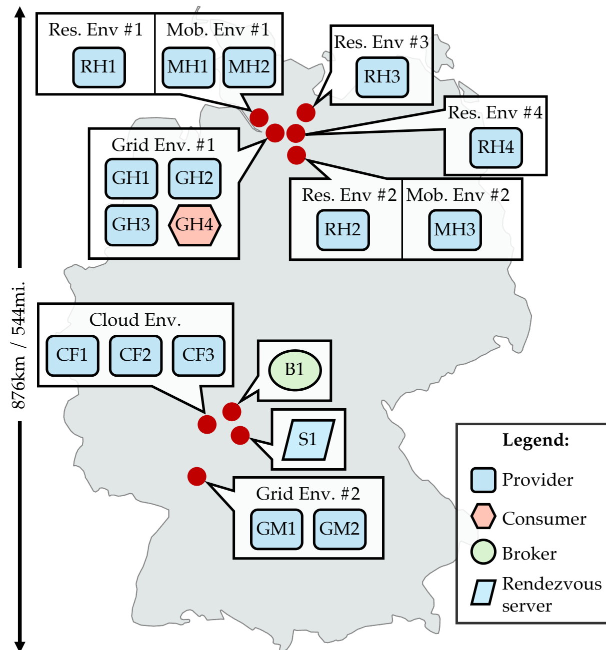


Figure 7.6: Location of all environments and nodes with their respective roles.

For this evaluation, sixteen heterogeneous devices were used at seven locations in Germany, near Hamburg, Frankfurt, and Mannheim, as shown in Figure 7.6. One device acted as a consumer that offloaded computations to

Environment	ID	Network/ISP	NAT type	CPU (cores/threads)	RAM	OS
Cloud	CF1	AWS Frankfurt	-	c5.xlarge (2C/4T)	8 GB	Ubuntu 20.04
	CF2			c5.xlarge (2C/4T)	8 GB	Ubuntu 20.04
	CF3			c5.xlarge (2C/4T)	8 GB	Ubuntu 20.04
Residential	RH1	Deutsche Telekom	Port-restricted	Xeon E3-1270v6 (4C/8T)	60 GB	Ubuntu 20.04
	RH2	Vodafone		M1 2020 (8C/8T)	16 GB	macOS 12.1
	RH3	O_2		Xeon E3-1225v2 (4C/4T)	32 GB	Ubuntu 20.04
	RH4	Deutsche Telekom		Xeon E3-1225v2 (4C/4T)	12 GB	Windows 10
Grid	GH1	University of Hamburg	Port-restricted	Xeon E3-1225v2 (4C/4T)	16 GB	Ubuntu 20.04
	GH2		Port-restricted	Xeon E3-1225v2 (4C/4T)	32 GB	Ubuntu 20.04
	GH3		Port-restricted	Xeon E3-1225v2 (4C/4T)	32 GB	Ubuntu 20.04
	GH4		Port-restricted	Xeon E3-1225v2 (4C/4T)	16 GB	Ubuntu 20.04
	GM1		Symmetric	Core i7-8700K (6C/12T)	32 GB	Windows 10
	GM2		Symmetric	Core i7-8700K (6C/12T)	32 GB	Windows 10
Mobile	MH1	Deutsche Telekom (4G LTE Band 7)	Symmetric	Core i5-6300U (2C/4T)	8 GB	Ubuntu 20.04
	MH2	Deutsche Telekom (4G LTE Band 7)		Core i5-8350U (4C/8T)	8 GB	Ubuntu 20.04
	MH3	O_2 (4G LTE Band 7)		Ryzen 7 3700X (8C/16T)	64 GB	Ubuntu 21.10
Broker	B1	DigitalOcean	-	Xeon 62xx (1vCPU)	1 GB	Ubuntu 20.04
Rendezvous server	S1	Vultr	-	5th Gen. Core (1vCPU)	1 GB	Ubuntu 20.04

Table 7.1: Devices, networks/ISPs, operating systems and NAT types used in the evaluation [Röb22].

fifteen providers. Additionally, two other hosts served as a broker and a rendezvous server, respectively. The devices varied in type (desktop or mobile systems), Internet connection (cellular or wired), and communication restrictions imposed by the networks to which the devices belong, such as firewalls or NATs blocking inbound connections, NATs rendering hosts unroutable, and ISP policies blocking P2P connections entirely, as detailed in Table 7.1.

Devices without NAT-imposed restrictions can be reached directly, while devices with such restrictions must be made reachable using the NAT traversal techniques first, which is transparently applied by the communication overlay. Some NATs restrict any P2P connections, requiring communication to occur via a public rendezvous server as a last resort. The devices were placed in different environments, allowing evaluation of the Tasklet system's performance in various scenarios. Providers were either located in the cloud, on a local or remote grid, in private households (residential), or in cellular networks as mobile devices.

In the following sections, devices are referred to by systematic identifiers. For consumers and providers, the first letter indicates the environment (C for "cloud", R for "residential", G for Grid, M for "mobile"). The second letter represents the location (H for Hamburg, F for Frankfurt, M for Mannheim). B denotes the broker, and S denotes the rendezvous server (see Figure 7.5).

The *cloud environment* (CF1–CF3) was located in the AWS data center in Frankfurt. All devices in this cloud environment were provided with public IP addresses and configured with no firewall, allowing these devices to be directly routable from any other device. The *residential environment* devices (RH1–RH4) were distributed in and around Hamburg and equipped with broadband Internet connections. Each residential network was isolated from the Internet through an NAT, which applied network address translation and blocked all inbound connection attempts. Therefore, to enable bilateral connection establishment with devices in these residential environments, the overlay network must be used to overcome the barriers imposed by the NAT. The filtering policies applied by the NAT permitted P2P connection establishment. The first *grid environment* was located in a university network in Hamburg (3 providers, GH1–GH3; 1 consumer, GH4). A second grid environment was placed in a university network in Mannheim (2 providers, GM1–GM2). Each university network was isolated from the Internet through a network-wide firewall that blocked all inbound connections. Within each grid, devices could reach each other, but bilateral connection establishment with devices from the Internet was only possible using the overlay network. Two *mobile environments* with three providers (MH1–MH3) were also utilized. Both mobile environments were equipped with

a carrier-grade NAT, assigning each mobile device a private, unroutable IP address and preventing any inbound connection attempts. Additionally, all mobile devices were isolated from each other, making P2P communication between these devices impossible. The overlay network allowed these devices to be reachable from the Internet, but P2P connections between two mobile devices were restricted by the strict filtering policies of the carrier-grade NAT. For communication between mobile devices, relaying through `S1` was required.

The broker `B1` and the rendezvous server were located near Frankfurt, Germany, in data centers operated by DigitalOcean and Vultr. Both were configured with public routable IP addresses and were not protected by a firewall.

7.2.1.2 *Edge computing scenarios*

Seven scenarios were evaluated, each representing an individual experiment. In each scenario, a subset of providers was used for execution, except for the "combined" scenario, where all available resources were utilized.

Each scenario required a different overlay network topology to be enforced. Further, the quality of service (QoS) goal "reliability" was set for all communication. In the following sections, each scenario is briefly presented.

CLOUD This scenario resembles traditional cloud offloading. Three Amazon EC2 `c5.xlarge` instances were rented in the same data center/availability zone in Frankfurt (`CF1`–`CF3`). Since no network barriers, such as firewalls, exist in this scenario, the consumer `GH1` can communicate directly with the providers. Therefore, the overlay network did not need to make any hosts reachable in this scenario.

RESIDENTIAL This scenario models typical residential setups (`RH1`–`RH4`). Each environment is located near Hamburg, Germany, and within different NATs. As a result, the overlay network must first make all providers reachable. Without this, the providers would not be available as offloading targets. The relaxed filtering policies applied by the residential gateways allow for P2P communication.

GRID-LOCAL In this scenario, a grid environment within an institute or company network is assumed (`GH1`–`GH4`). All devices, including the resource consumer, are located within the same LAN. This environment represents the ideal setting for P2P communication, as there are no network barriers on any side.

GRID-RELAYED The Grid-Relay scenario is a variation of the Grid-Local environment. In this scenario, all LAN-based communication between peers is intentionally disabled using a "routing" QoS goal. As a result, all communication is relayed through the public server `S1`. This setup allows for the comparison of the impact of the absence of the overlay network, as direct communication between consumers and providers is not possible.

GRID-REMOTE In this scenario, the consumer (`GH4`) is located in a different location than the grid's providers (`GM1` and `GM2`). Although these locations provide a high-quality Internet connection, institute/company-wide NATs require that the providers be made reachable via the overlay network first.

MOBILE In this scenario, offloading in mobile environments is assessed (`MH1`–`MH3`). Two of Germany's three Tier 1 mobile carriers, Deutsche Telekom and O_2, are covered. Mobile environments are typically very P2P-hostile, as mobile operators deploy symmetric NATs that prevent establishing any direct link between two mobile devices. As a result, communication between mobile devices must be relayed. Furthermore, due to the cellular connection, this environment can experience significant disruptions and jitter. Therefore, this environment represents the worst-case scenario for computation offloading.

COMBINED This scenario combines the Cloud, Residential, Grid-Remote, and Mobile environments. The goal of this scenario is to evaluate how highly heterogeneous environments impact the results. This scenario introduces the risk of observing negative performance effects on overall completion time due to stragglers.

7.2.1.3 *Tasks*

During the experiment, 16 800 tasks were executed across seven scenarios. Each task was split into 12 individual sub-task and was repeated 100 times. The result of a task is returned to the application once all 12 sub-tasks results have been obtained. The offloading experiments have been conducted with four different types of tasks. They are divided into two CPU-intensive (option pricing & a color key filter) and two bandwidth-intensive tasks (image convolution & ray tracing). The influence of network connectivity or computational power on offloading performance can be determined using this spectrum. Since all four application types show the same trend, only two tasks are presented in

this thesis: *Option pricing* (CPU-intensive) and *Image convolution* (bandwidth-intensive).

OPTION PRICING This task represents a CPU-intensive computation with low network load, using a Monte Carlo method to price a European call option. A Monte Carlo experiment uses random sampling to estimate mathematical or statistical outcomes. A European call option is a financial contract allowing the holder to purchase an underlying asset at a predetermined price, but only on a specific date in the future, without any obligation. The compiled code for this task is approximately 5 kB in size, takes two integers as input parameters, and outputs a single floating-point number. Consequently, the network load can be considered negligible for this task type.

(**a**) Source image. (**b**) Filtered image.

Figure 7.7: Source image and filtered output image of the image convolution task for edge kernel.

IMAGE CONVOLUTION An image convolution filter was implemented to exemplify tasks with high network load and moderate CPU usage. This filter constructs a 9×9 matrix for each pixel to detect the edges of images. Since this task uses images with the same resolution for both input and output parameters, the network load is higher and strongly dependent on the size of the image. In the experiment, the image was 1867 pixels wide and 1050 pixels high, resulting in 2 million pixels and 6 million RGB values that needed to be transferred twice (source input image and filtered output image) per computation (see Figure 7.7).

7.2.2 *Experimental metrics*

In this experiment, both qualitative and quantitative metrics have been captured. The qualitative metrics assessed whether the middleware successfully constructed the desired overlay network and whether task offloading was effectively implemented. On the other hand, the quantitative metrics involved investigating the behavior of different edge environments by measuring the

timings associated with transferring the task from the consumer to the provider ("offload task") and the timings of transferring the results back to the consumer ("return result").

7.2.3 *Results*

Figure 7.8: Average transmission times in milliseconds for a task of a CPU-intensive (left) and bandwidth-intensive (right) application.

Figure 7.8 and Figure 7.9 plot the average transmission times in dependence on the edge scenario. The figures summarize the results of the experiments for the CPU-intensive option pricing (left) and bandwidth-intensive image convolution (right) tasks. The figures show the average transmission times observed for offloading the task and returning the result. Transmissions were routed either directly through P2P connections or via the rendezvous server S1, depending on the specific environment. Since computation times strongly depend on the CPU performance of the respective device, the focus is placed only on transmission-related operations (offloading a task from a resource consumer to a provider and returning the result) to provide insights into environment-related performance. The outcomes of the results will now be presented and discussed:

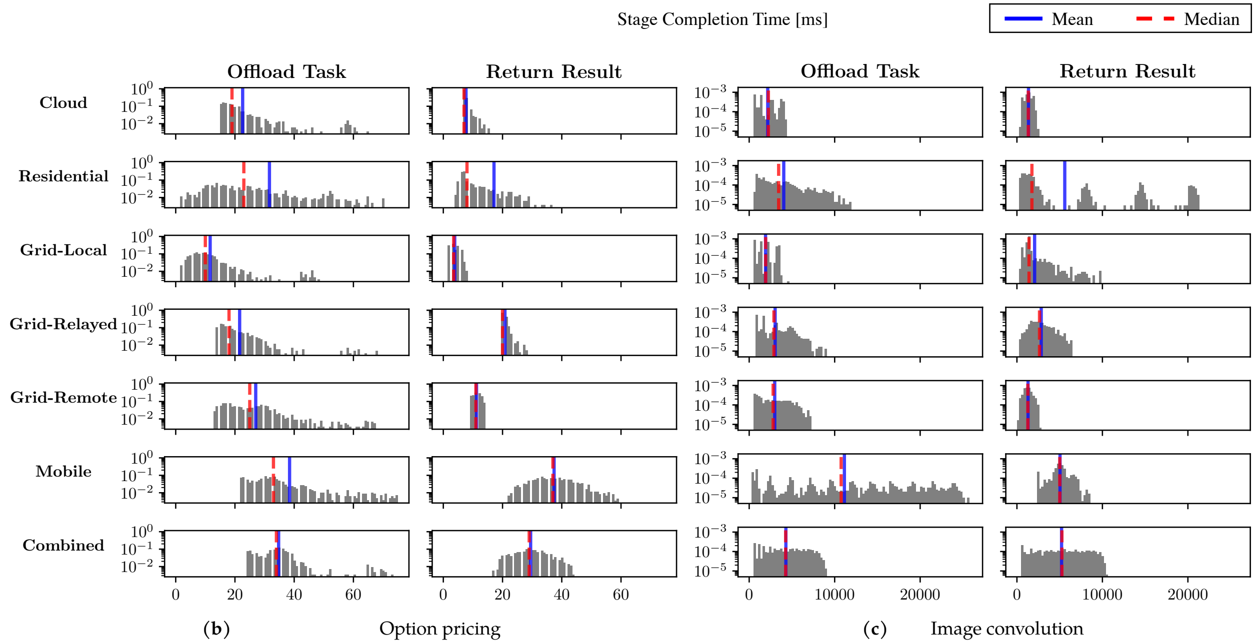


Figure 7.9: Histograms showing the density of the completion times in milliseconds of the individual life cycle stage in each scenario.

7.2.3.1 *Option pricing (CPU-intensive)*

The *Cloud* scenario performs very well, even though the providers are located hundreds of miles away from the consumer. This strong performance is attributed to the large and reliable symmetrical network connections of the cloud servers. However, a lower performance bound exists due to network latency caused by the long distances.

The *Residential* scenario's median performance is slightly higher, and the mean is 50 % higher than that of the cloud scenario. A more significant variance in performance is observed, attributed to the heterogeneity of the residential devices. While some tasks were completed within 5 ms, others took over a minute due to stragglers. This issue with stragglers can be mitigated by offloading the same task to multiple consumers in parallel and returning the first available result. Applying this scheduling strategy is expected to improve performance, demonstrating that P2P-connected edge resources can achieve competitive performance.

It is observed that the *Grid-Local* scenario achieves the best overall performance. This result aligns with expectations, as this environment is optimal for offloading. Grid-Local is twice as fast as the Cloud environment, demonstrating that with comparable CPU performance, a significant performance increase is possible in locality-aware computation offloading systems.

The results of the *Grid-Relayed* scenario demonstrate the impact of missing P2P connections compared to the Grid-Local environment. Since all communication must be routed through a public rendezvous server, the performance is generally lower in this scenario, with approximately 22 ms compared to 11 ms.

The *Grid-Remote* scenario performs worse than the Cloud environment but better than the Residential environment. Apart from a few stragglers, the execution time shows low variance, attributable to the homogeneity of the devices present in the grid. This scenario further demonstrates that edge computing can be effectively applied to institutional resources, offering competitive advantages over traditional cloud resources.

The *Mobile* scenario exhibits the worst mean and median performance of all scenarios. This aligns with expectations given the nature of cellular communication, the resulting network latency, and the shared network capacity among all carrier customers within a radio cell. It is notable that the proximity of consumers and providers does not result in improved performance.

In the *Combined* scenario, all providers from the Cloud, Residential, Grid-Remote, and Mobile environments were combined. The overall performance is comparatively poor, despite this scenario offering the most resources. This is attributed to the previously mentioned splitting of tasks into 12 sub-tasks and the significant impact of stragglers on performance. This issue can be mitigated by using a different scheduler that prioritizes faster devices or employs parallel offloading.

7.2.3.2 *Image convolution (Bandwidth-intensive)*

Due to the fast connection between the consumer and the providers, the *Cloud* environment consistently performs well for data-insensitive tasks. This environment is not significantly affected by the type of application.

The *Residential* scenario performs slower than the Cloud environment, with the task offloading stage displaying a higher variance in completion times. Four clusters can be identified in the histogram for the return result stage, each representing one of the four residential providers in this scenario. The asymmetrical downlink and uplink of the providers significantly affect these environments, particularly when returning the result. If the consumer is also situated in a residential environment, the uplink will have a strong influence on the offloading task stage.

Grid-Local exhibits the best performance for the offloading task stage and the second-best overall performance. Although the times are comparable to

the Cloud environment, it is observed that stragglers negatively impact the performance during the result return stage.

In the *Grid-Relayed* environment, it is evident that relaying traffic significantly impacts network times. For instance, the time required to offload a task is increased by 33 %, and the time to return the result is increased by 46 % compared to the direct traffic observed in the Grid-Local scenario.

In the offload task stage of the *Grid-Remote* environment, the mean is approximately 1000 ms slower, and the median is nearly 900 ms slower compared to the Grid-Local environment. This decrease in performance is due to the providers being several hundred kilometers/miles away from the consumer.

The *Mobile* scenario represents the weakest-performing environment for these task types. Compared to the CPU-intensive task, there is a considerable variation in the offloading task stage times. This result aligns with expectations that the varying quality of radio communications significantly impacts bandwidth-intensive tasks. In contrast, the response result does not exhibit such significant deviation, which can be explained by the more restricted uplink of the asymmetric cellular connection.

The *Combined* environment is also evident in this scenario. It should be noted that, compared to returning the result, the offloading stage is slightly faster due to the asymmetrical uplink of some devices.

7.2.4 *Evaluation conclusion*

This evaluation assumed a computation offloading system that utilizes cloud, grid, and (idle) edge computation resources. The following main challenges has been addressed in the evaluation: First, the heterogeneity of hardware, software, connectivity, and reliability of the involved devices needed to be overcome to create a unified computing platform. Second, different network topologies and QoS goals have been successfully enforced. Third, the network environment context information have been collected that can be used to support QoS mechanisms. The system was deployed in a real-world setup with peers located in various cloud, residential, grid, and mobile environments.

It was demonstrated that residential and grid resources are only marginally slower than cloud resources regarding network times, making them cost- and energy-efficient alternatives. The additional resources from the edge were shown to positively impact application performance, indicating that these edge resources can serve as cost-effective alternatives to cloud resources.

Furthermore, the experiments indicated that the absence of P2P communications could significantly degrade overall application performance by up to a factor of 5.

In this evaluation, it has been shown how an existing application – the Tasklet system – that normally only functions in unrestricted networks, can be operated in restricted networks without requiring changes to the application. This enables the application to access resources that could not previously be used.

7.3 EVALUATION OF CENTRALLY-OPTIMIZED OVERLAY NETWORKS

This chapter describes the evaluation of an approach to simplifying the deployment and operation of overlay networks at the network edge. The evaluation assumes complementing edge computation scenarios and is divided into two complementary sub-evaluations. Both evaluations focus on constructing an overlay network based on high-level functional requirements. The underlying resources are heterogeneous, and initial knowledge about these edge devices is limited. In both evaluations, the overlay network is optimized and redeployed over time as more data is collected. The first evaluation considers heterogeneous link latencies between edge devices, while the second evaluation considers heterogeneous edge device computing performances.

The evaluation demonstrates that the implemented SDON layer can do overlay network management based on high-level functional requirements. Therefore, the implementation of the SDON layer, including all components, has been used in two scenarios: One requiring overlay link resource optimization, the other requiring overlay node resource optimization. Further, with this experiment, we address the following research questions:

> **RQ 3.1** How to program overlay networks that assist application developers to use edge resources efficiently?

> **RQ 3.2** How to simplify overlay network deployment and management at the network edge?

7.3.1 *Latency-based routing optimization*

This evaluation focuses on optimizing the overlay routing based on link latencies. This section first describes the evaluation use case, setup (Section 7.3.1.2), followed by the metrics (Section 7.3.1.3), the results (Section 7.3.1.4).

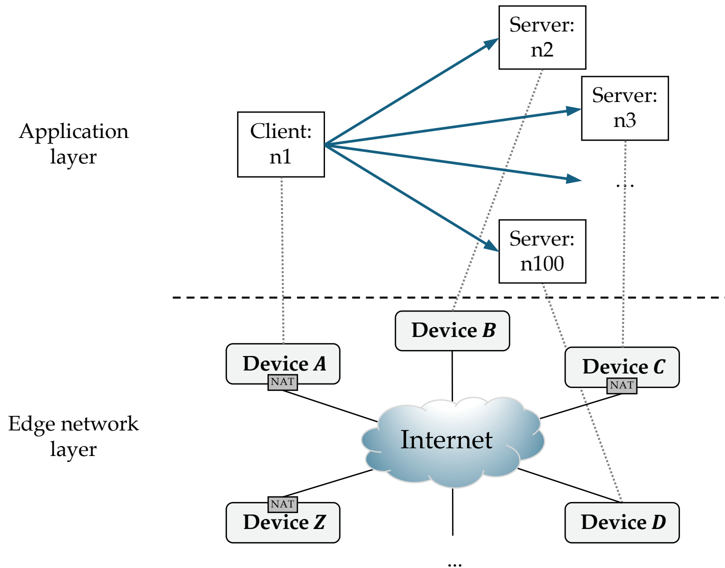
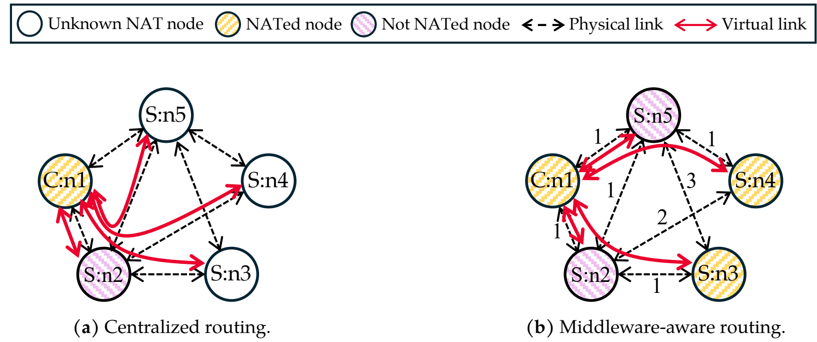


Figure 7.10: Real-time application used for first evaluation of SDON layer.

7.3.1.1 *Use Case*

This use case assumes a widely distributed real-time application as shown in Figure 7.10, where a client (`n1`) wants to communicate with a set of servers (`n2`–`n100`), each placed in a different edge network. The client and most of the servers are placed behind separate symmetric NATs (Section 2.4), while some servers are not NATed and, therefore, publicly reachable. Since hole punching is not practically feasible over two symmetric NATs, the only option for the client to reach NATed servers is relayed communication (Section 2.4.2).

ROUTING ALGORITHM Instead of just rely on the rendezvous server, the application uses a custom routing, using one of the accessible servers as a relay. Achieving the best performance requires the application to determine which servers are accessible and which are only accessible via a relay. In the latter case, the application must find which node can act as the relay with the lowest latency and configure the routing table accordingly.



(a) Centralized routing.

(b) Middleware-aware routing.

Figure 7.11: In (a), client `n1` communicates with all servers `n2`–`n5` through `n2` acting as a relay. In (b), faster paths are selected due to additional knowledge about link latencies (edge labels) and NAT information.

Application developers often prefer a simpler implementation approach where all nodes route through a centralized relay, as shown in Figure 7.11a. This use case is chosen to demonstrate how the middleware facilitates the programming of an overlay network that performs these functions, as illustrated in Figure 7.11b.

Snippet 7.1: Network model for NAT-aware routing.

```
network = create_network({measure_latency='0.2|60'})
initial_relay = 'n2'
network:add_node('n1', {ip='10.2.1.1', run='client.sh'})
for i = 2, 100 do
  network:add_node('n'..i, {ip='10.2.1.'..i, run='server.sh'})
  network:add_link('n1', initial_relay)
  network:add_link(initial_relay, 'n'..i)
  network:get_node('n1').routes['n'..i] = initial_relay
  network:get_node('n'..i).routes['n1'] = initial_relay
end
network:set_callback(function(network, devices)
  network:clear_links()
  for i = 2, 100 do
    fastest_relay = fastest_path('n1', 'n'..i)
    network:add_link('n1', fastest_relay)
    network:add_link(fastest_relay, 'n'..i)
    network:get_node('n1').routes['n'..i] = fastest_relay
    network:get_node('n'..i).routes['n1'] = fastest_relay
  end
end)
```

NETWORK MODEL

Snippet 7.1 demonstrates the network model, providing functions to optimize routing.

- In line 1, a new network model is created. The parameter `measure_latency` is set to `0.2|60` resulting in the SDON controller enforcing all SDON devices to randomly select a subset of 20 % of other devices for latency testing, repeated every 60 seconds. Otherwise, no latency information would be available to optimize the overlay network. The results of these tests, which fail for non-routable devices, thereby identifying NATed devices, are automatically aggregated at the SDON controller.
- The initial relay `n2` is specified in line 2.
- In line 3, the client node `n1` is added, setting the overlay IP address to 10.2.1.1.
- In lines 4 to 10, servers `n2–n100` are added. Initially, the client uses `n2` as a central relay to reach all servers (see Figure 7.11a).
- Lines 11 to 20 define a function triggered upon receiving new latency information. This function allows the identification of the fastest relay and updates the topology and routing tables accordingly (see Figure 7.11b).
- Although the exact code for selecting the fastest relay in line 14 is not shown for brevity, the process involves creating a latency matrix that includes all latencies, enabling the selection of the fastest relay.

7.3.1.2 *Experimental setup*

In this experiment, Mininet [LHM10] is used to emulate a network consisting of 103 nodes, with one serving as a rendezvous server, another as the SDON controller, and the remaining 101 nodes acting as SDON system nodes. One SDON system node functions as the application client, while the other 100 act as servers. 92 % of the SDON system nodes are placed behind NATs, reflecting real-world end-host statistics in edge networks [Haa+16]. To equip the Mininet environment with NATs exhibiting the desired symmetric behavior, a software was developed that implements several NAT behaviors [Bor23a]. Latencies based on the King dataset [GSG02] are incorporated, providing real-world latencies between a set of 1740 Internet hosts, with a random subset of 101 hosts drawn to configure the latencies of the paths between the client and servers. The controller is configured with the SDON system settings specified in Snippet 7.1, where `measure_latency` is set to `n|60` with $n \in \{\frac{0}{8}, \frac{1}{8}, \dots, \frac{8}{8}\}$,

resulting in nine individual settings. Thus, for each setting, n relay candidates are considered for every client-server path. When $n = \frac{0}{8}$, centralized relaying is used, whereby all paths use the same predefined relay.

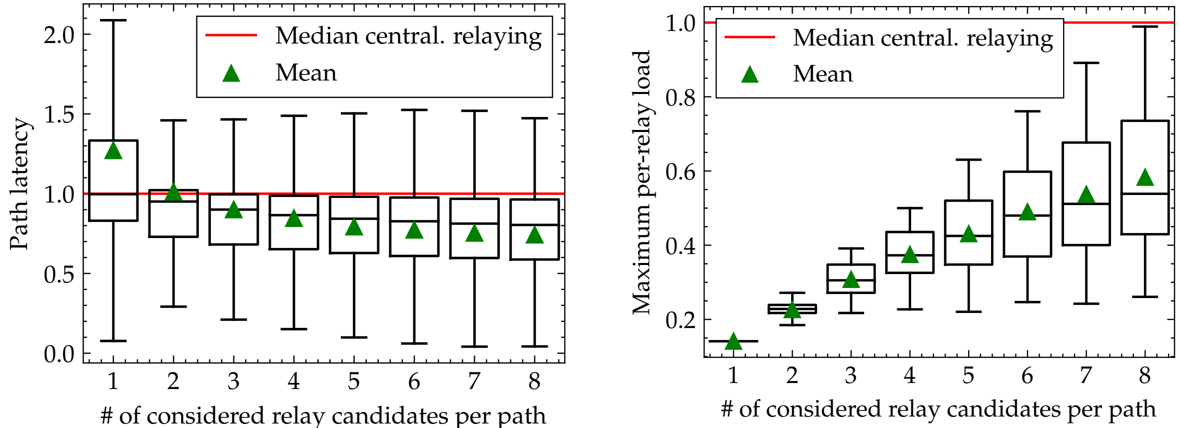
An experiment run begins with the rendezvous server and controller being started, followed by all nodes registering with both. Subsequently, the controller instructs the nodes to perform latency measurements with a random subset (as defined by the above parameter), to then choose the fastest relay, and to configure the corresponding routing tables. Once routing tables are populated, the application client contacts all servers in sequence. The experiment is repeated 100 times, testing all nine possible settings in each iteration. For each iteration, a new random subset of latencies from the King dataset is drawn, resulting in a total of 900 runs.

7.3.1.3 *Experimental metrics*

`fping` is used to measure latency for each client-server path, calculating the average from 10 ICMP pings. Additionally, `tcpdump` is employed to monitor relay traffic, using the number of UDP packets handled by each relay as a load metric.

7.3.1.4 *Results*

The results are shown in Figure 7.12.



(a) Path latencies in dependence on the number of considered relays.

(b) Maximum per-relay load in dependence on the number of considered relays.

Figure 7.12: Median client-server-path latency and maximum per-relay load in dependence on the number of considered relays per path. Results have been normalized to centralized relaying within the same experiment iteration.

PATH LATENCIES Figure 7.12a shows the median path latencies for client-server communications depending on the number of considered relays. These latencies are plotted on the y-axis and normalized to the median latency for centralized relaying within the same experiment iteration. This normalization is performed to compare all latencies with centralized relaying, which is commonly used due to its ease of implementation for communication across NATs. The x-axis indicates the number of relays considered for each client-server path to identify the fastest relay through latency measurements. The solid line represents the median latency of centralized relaying, while the triangle marks the mean latency for each setting.

The results indicate a logarithmic improvement in median and mean path latencies as more relay candidates are considered. The highest scattering is observed when only one relay is considered, which can be attributed to the fact that, in this setting, a random relay is effectively chosen since only one relay is tested, preventing the selection of the best option. Consequently, this setting shares the same median latency as centralized relaying, whereas, in all other cases, the median latency is improved. Including a second and third relay improves performance by 5 % with each additional candidate. This improvement decreases to 2 % and eventually to 1 % for each subsequent candidate. Overall, performance can be enhanced by 20 % when all eight available relays are considered.

MAXIMUM PER-RELAY LOAD Figure 7.12b shows the median of the maximum per-relay load depending on the number of considered relays. These loads are plotted on the y-axis and are normalized to the median load for centralized relaying observed within the same iteration. In all other respects, this sub-figure is structured in the same manner as the left-hand sub-figure.

The results show a logarithmic increase in load as more relay candidates are considered. This growth occurs because the likelihood of using a globally optimal relay increases. Additionally, the scattering of the load also increases due to this effect. For all eight settings, the load per relay remains within the range observed when a central relay is used. Addressing the second experiment's research question, the inclusion of a second relay increases the load by 67 %. This increase reduces to 40 % and 21 % with each additional candidate. Overall, performance can be enhanced by 20 % when all eight available relays are tested.

Considering the results of both figures in Figure 7.12, a favorable trade-off between reduced latency and moderate load is achieved when half of the relays are tested for each path.

7.3.2 *CPU-based node arrangement optimization*

This evaluation focuses on the overlay node arrangement based on node CPU performances. This section first describes the evaluation use case, setup (Section 7.3.2.2), followed by the metrics (Section 7.3.2.3), the results (Section 7.3.2.4).

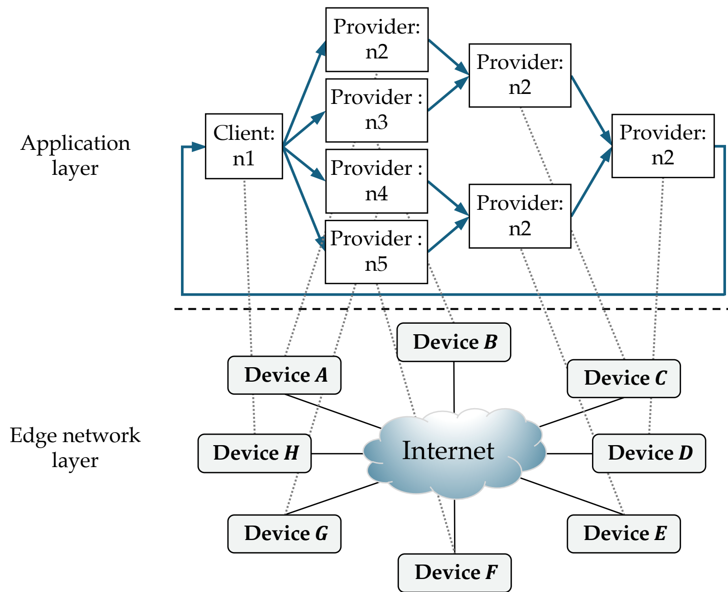


Figure 7.13: Computation-offloading application used for first evaluation of SDON layer.

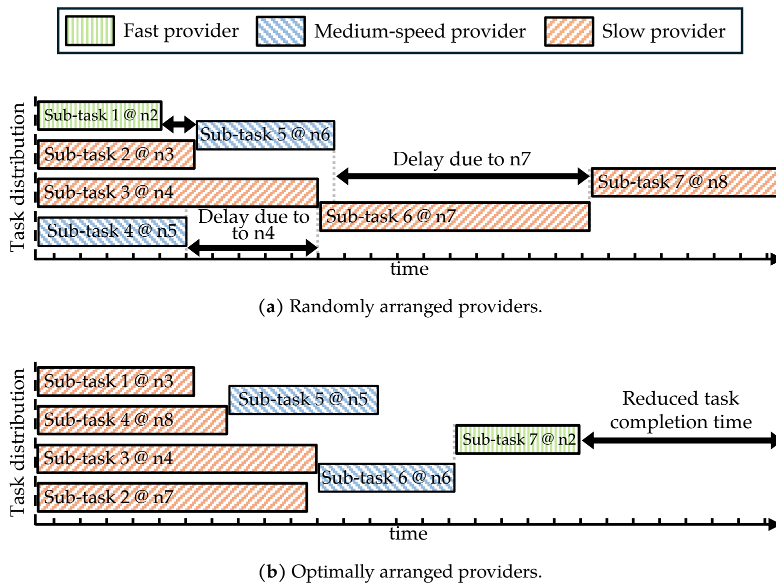
7.3.2.1 *Use Case*

This use case assumes a computation offloading system where a node (denoted as the consumer) wants to offload a batch of similar computations to a group of other nodes (denoted as providers). Each computation involves finding the largest element in a set, which requires a pairwise comparison of the elements. Here, parallel reduction is employed, where the set is evenly divided among the providers. Each provider then calculates its local maximum and forwards it to the next provider. This step is repeated until only one element – the global maximum – has been identified. Each provider compares exactly two elements, with the computational load per comparison being equal. This type of computation results in a pipeline that resembles a binary tree, where each provider acts as a node and the final result is received back from the root. Consequently, each provider must wait for the completion of the previous two providers, always waiting for the slower of the two providers. Therefore,

it is beneficial that the two preceding nodes have as similar performance as possible. This problem is illustrated in Figure 7.14. Here, we see how the timings of subsequent sub-tasks impact overall completion time. In Figure 7.14a, the providers n2–n8 are not optimally arranged. The computation task in this figure compares eight elements, with two elements sent to nodes n2–n5 respectively. These, in turn, send their results to the subsequent providers n6 and n7. Here, n6 must wait for n3 and n7 must wait for n4 before the sub-task can begin. This causes sub-task 6 to be significantly delayed, affecting the overall completion time.

ARRANGEMENT ALGORITHM Achieving the best performance requires the application to assess individual providers' performance and arrange the computation pipeline that minimizes delays due to stragglers. An optimized pipeline is shown in Figure 7.14b: The fastest provider is placed at the last stage. The second and third providers are placed in the stage before. The slowest providers are placed in the first stage, whereas the slowest providers in this first stage are placed before the fastest providers in the second stage.

This use case is chosen to demonstrate how SDON system assists in configuring an overlay that takes over these functionalities, hiding the complexity in performance distinction of providers and automatically reconfiguring the overlay, resulting in faster computation completion times.



(a) Randomly arranged providers.

(b) Optimally arranged providers.

Figure 7.14: Task completion time is reduced when fast providers are placed at the end of the processing pipeline.

```
network = create_network()
network:add_node('n1', {ip='10.2.1.1', run='consumer.sh'})
for i = 2, 8 do
  network:add_node('n'..i, {ip='10.2.1.'..i, run='provider.sh', record_packets={offload={magic_number='0x1'},
  ↪ result={magic_number='0x2'}}})
  network:add_link('n1', 'n'..i)
  for j = i + 1, 8 do
    network:add_link('n'..i, 'n'..j)
  end
end
network:set_callback(function(network, devices)
  providers = providers_n_sorted(network, sort_count)
  for i = 1, #providers do
    network:get_node(providers[i]).ip = '10.2.1.'..(i+1)
  end
end)
```

Snippet 7.2: SDON system configuration for computation offloading.

NETWORK MODEL Snippet 7.2 shows the network model of the SDON system, providing functions that optimize the pipeline arrangement.

- In line 1, a new network model is created.
- In line 2, the consumer `n1` is added, and in the subsequent for loop, seven providers `n2`–`n8` are added. The magic numbers of the application messages *offload task* and *return result* are assigned to each provider. This enables SDON system nodes to identify these messages and report the timings to the controller. The controller then uses these timings to calculate sub-task times, estimating the CPU capacity per provider.
- Additionally, lines 5 to 8 ensure that the consumer and all providers can reach each other.
- The code for clustering providers based on performance (line 11) is omitted, which estimates task completion times by using the time delta between offload and result messages, calculates average times per provider, compares these averages with other providers, and then creates a list of providers where the fastest `sort_count` providers are moved from their current position to the end.
- The for loop in lines 12 to 14 virtually flips providers' positions in the computing pipeline by changing their overlay addresses.

This experiment demonstrates how a SDON system helps an application better utilize host resources.

7.3.2.2 *Experimental setup*

In this experiment, Mininet is used again to emulate a network consisting of 10 nodes, with one serving as a rendezvous server, another as the SDON controller, and the remaining eight nodes acting as SDON devices. One SDON device functions as the application consumer, while the other seven act as providers. `cgroups` is used to limit the CPU resources per provider to create a network of devices with heterogeneous capacities [ker15]. For CPU capacities, a random sample is drawn from a normal (Gaussian) distribution, resulting in sub-task times with a mean (μ) of 1000 ms and a standard deviation (σ) of 400 ms. The controller is set up with the SDON network model specified in Snippet 7.2, where `sort_count` is set to $n \in \{0, 1, \ldots, 7\}$, resulting in eight individual settings. Thus, with each setting, n providers are optimally placed.

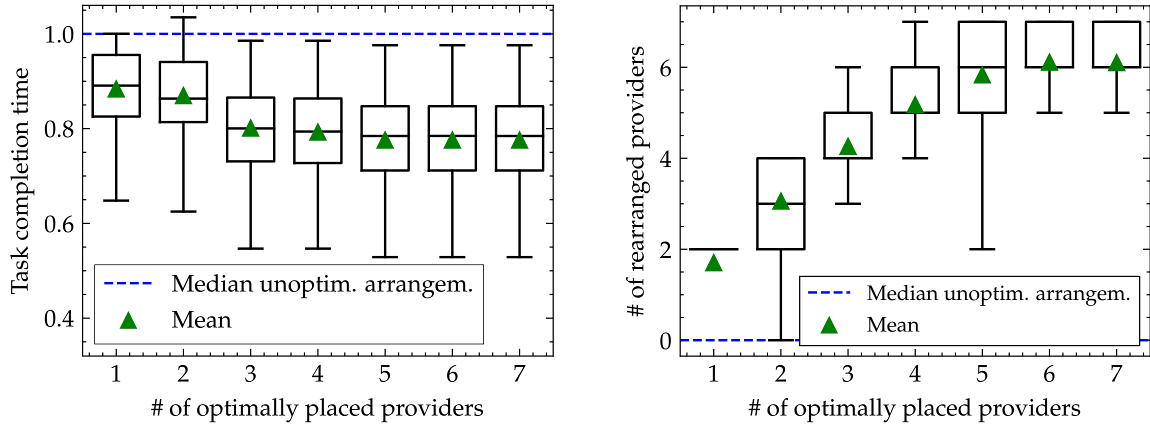
An experiment run begins with the rendezvous server and controller being started, followed by all nodes registering with both. Initially, the controller arranges providers randomly, as no information about the individual CPU capacities is known. The consumer submits ten tasks to the providers, resulting in seven sub-tasks computed on the providers, for a total of 700 sub-tasks being performed. As each provider reports the times of sub-tasks using the provided magic numbers, the controller learns about the performance of each provider. The SDON controller then rearranges the first n fastest providers. Once the overlay is reconfigured, the application submits another batch of ten tasks. The experiment is repeated 100 times, testing all eight possible settings in each iteration. For each iteration, a new random sample of CPU capacities is drawn, resulting in a total of 800 runs.

7.3.2.3 *Experimental metrics*

The time is measured from when each task is submitted by the consumer to the first group of providers until the final result is received from the last provider, and this is used as the overall computation completion time.

7.3.2.4 *Results*

The results are shown in Figure 7.15, illustrating the median task completion times based on the number of optimally placed providers. These times are plotted on the y-axis and normalized to the median time for an unoptimized



(a) Median task completion time in dependence on the number of optimally placed providers.

(b) Maximum per-relay load in dependence on the number of optionally placed providers.

Figure 7.15: Median task completion time and rearrangements in dependence on the number of optionally placed providers. Task completion times have been normalized to randomly placed providers within the same experiment iteration.

provider arrangement ($n = 0$) observed within the same iteration. This normalization is performed to compare all times with a random (unoptimized) arrangement, which is considered the baseline for its simplicity. The x-axis indicates the number of providers optimally placed within the computation pipeline. The dashed line represents the median of optimally placed providers, while the triangle denotes the mean completion time for each setting.

The results indicate an improvement in the median and mean completion times as more providers are placed optimally. Special attention should be given to the setting with two optimally placed providers, which is the only setting where a performance decrease is observed in some iterations. This decrease occurs when the initial provider arrangement includes similarly performing providers at the first processing pipeline stage (sub-tasks 1 to 4), but a provider from this balanced stage is swapped out for a slower provider in stage two (sub-tasks 5 to 6). Under these circumstances, the decline in performance in the first stage is greater than the gain in the second stage. The best performance gain is observed when three providers are optimally placed, resulting in optimal levels one and two. Optimizing the third level only slightly further improves performance, as the likelihood of additional improvements decreases when only providers within the same level are moved. Placing more than five providers does not result in further improvements, as this only causes the last two providers in the third level to swap, which does not impact performance. Regarding the first experiment's research question, the best performance gain is achieved when all but the last two providers are optimally placed, resulting in a performance gain of 22 %. The greatest performance gain is observed when

additional stages are optimized. Optimizing the first stage increases performance by 11 %, while an optimized second stage increases performance by 6 %.

7.3.3 *Conclusion*

In the evaluation, the novel concept of SDON system was introduced and applied in middleware, combining central control and configuration, intent-based overlay programming, support for IP applications, and operation in rigid environments with no control over intervening network devices. Through two use cases, it was demonstrated how overlay networks can be seamlessly configured and automatically optimized according to specified KPIs. The results indicate that, in the evaluation scenarios, SDON system and centrally enforced optimization at the SDON controller can improve mean network and host resource usage by up to 20 % and 22 %, respectively. This demonstrates that with SDON system, applications can delegate functionalities to the underlying overlay layer, making SDON system a suitable tool for accelerating the prototyping, development, and deployment of distributed applications or algorithms in real-world edge environments.

7.4 CHAPTER SUMMARY

This section presented the evaluation results of several key components of the SDON middleware.

First, a real-world experiment was conducted to demonstrate that the proposed protocol used by the communication layer reduces 1 to 2 RTTs in secure connection establishment in restricted networks compared to existing approaches.

Second, seven experiments were conducted, each in a separate real-world environment, including residential, mobile, cloud, grid, corporate, and mixed environments, to demonstrate how the proposed service layer successfully enforces required overlay network behaviors. Further, the experiment shows that a pre-existing application can use the middleware without modifications.

Third, two real-world data experiments indicate how the SDON layer improves edge computing application performance through centrally enforced optimizations by 20 % and 22 %, respectively.

These experiments were built upon and conducted on different early snapshots of the middleware implementation. They indicate that each middleware

layer fulfills the requirements, and the interaction between the layers works, ensuring that the middleware as a whole meets the requirements.

8 Conclusion

Distributed applications, such as those used in emerging application scenarios like autonomous vehicles, smart cities, healthcare, and industrial automation, require low-latency, high-bandwidth, and computational resources [Var+16; Cao+20]. Due to the distant locations of data centers from data production and consumption sites, cloud environments fail to meet these requirements. However, edge computing involves distributed applications that run at the cloud, edge, and end-systems simultaneously. Getting edge applications to work as unified systems spanning across these disparate edge environments makes application development complex [Shi+16].

Accordingly, this thesis presents contributions incorporated into a middleware, which simplifies the development and operation of applications running at the edge. This middleware provides functionalities that are essential for a seamless application operation at the edge. Functionalities that would otherwise have to be implemented by the applications themselves. Therefore, many of the "nasty" things that make edge application development and operation complex are hidden from the application.

Chapter 1 introduces research questions that we answer in the following:

> **RQ 1.1** How to achieve connectivity between edge devices efficiently?

In edge computing scenarios, devices are typically distributed across different networks. Communication between these networks is often restricted due to incompatible configurations or the presence of middleboxes (e.g., NATs, firewalls) by which communication is routed between these networks. Connectivity is usually unrestricted within networks, although some networks prevent this through client isolation.

Thus, mutual route discovery on the underlying network by both devices is required to connect edge devices. Many devices are not directly reachable from other networks by middleboxes, requiring the appliance of NAT traversal techniques. Furthermore, the heterogeneity of NATs makes it necessary to combine multiple techniques, each usually requiring separate protocols, to traverse middleboxes in the most optimal way. Therefore, to minimize the time

before two edge devices are linked, a new protocol is proposed that efficiently combines NAT traversal techniques [BRF23a] (Contribution 1). Compared to previous work using individual NAT traversal protocols, our protocol minimizes the number of messages required to determine the most local path between two edge devices, allowing for more efficient and faster connection establishment.

> **RQ 1.2** How to ensure that edge devices are linked securely?

Edge devices that host edge computing applications are often operated in untrusted networks. Therefore, there is an inherent risk that any information obtained might not be reliable. Data transmitted could be intercepted or manipulated.

Unique identification among edge devices is necessary to enable secure linking. Therefore, an approach is presented that equips all devices with a public-private key pair. The public part is also used for identification, and the keys together can be used to secure the communication. Thus, before two devices are linked, they mutually authenticate first. Further end-to-end encrypted communication between edge devices with the very first message is possible. This is achieved by piggybacking a Diffie-Hellman key agreement into the protocol described in previously [BRF23a] (Contribution 1). Compared with previous work, our protocol saves between 1 to 2 RTTs and secures the connection establishment process against multiple vulnerabilities.

> **RQ 2.1** How to integrate overlay networks into applications without imposing restrictions on application design and functionality?

Integrating overlay networks into applications often imposes restrictions on application design or functionality. Further, many integrations depend on a concrete type of overlay network, making it a complex task for the developer to migrate to a different overlay, which might be necessary to adapt to evolving application requirements.

Accordingly, an application programming interface (API) is presented, allowing the integration dynamic overlay networks without imposing restrictions. The approach establishes an IP overlay that can be adapted by any IP-based application without modification. While the application receives or sends communication from remote components as before, the overlay network transparently processes the packet delivery. Based on Contribution 1, the application

creates the impression of seamless, secure and location-transparent connectivity between all edge devices. The presented API has been successfully used with pre-existing applications and evaluated in real-world scenarios [Bor+23; Röb+23]. The evaluation demonstrated that using our approach, dynamic overlay networks can be transparently integrated without imposing restrictions. In addition, our approach enables applications to operate in restricted edge environments, which helps conduct real-world experiments (Contribution 2).

> **RQ 2.2** How to improve the use of heterogeneous edge resources through dynamic overlay networks?

Many distributed application systems [Kub14; Lig09; AZ17; Nig21; Ros+23] make strong assumptions about the underlying network infrastructure in terms of control and behavior. However, edge applications are often operated in networks where the edge computing application does not control the network configuration. Furthermore, edge devices are more heterogeneous, restricted, and unreliable than devices in cloud computing environments. Therefore, edge computing environments must be considered as best-effort systems, and additional mechanisms are required to provide applications with additional services, quality of service (QoS), security, and reliability for correct application functioning.

Therefore, a service layer is presented that dynamically integrates additional services into the overlay network through an extensible set of mechanisms. Individual services can be flexibly applied to certain overlay elements and operated in coexistence. The service layer is transparently integrated into the previously presented API. Thus, overlay services can be added or removed as needed, without affecting the application's transparency. As a result, applications got an idealized view of the edge resources that provide more service guarantees in an otherwise best-effort system (Contribution 2).

> **RQ 3.1** How to program overlay networks that assist application developers to use edge resources efficiently?

Overlay networks are as versatile as the edge applications that utilize them. They can vary in size, topology, node, and link properties. Further, overlay networks apply different mechanisms to maintain and restore the desired state.

Accordingly, an approach is presented that allows application developers to program *software-defined overlay networks* using high-level functional require-

ments. Desired overlay nodes, overlay links, their respective resource capabilities, and desired behavior can be described. This goal-oriented overlay programming relieves developers from identifying and describing routines to reach the desired overlay state, as our middleware automatically identifies these routines. Further, policies can be specified, describing how an overlay network should respond to unwanted network changes or customizing how overlay elements are mapped to edge resources [Bor+25] (Contribution 3).

> **RQ 3.2** How to simplify overlay network deployment and management at the network edge?

Deploying and operating an overlay network at the network edge is challenging due to the heterogeneous, restricted, and unknown environments and the absence of central control.

Therefore, an software-defined overlay networking middleware is presented that brings capabilities known from software-defined networking, such as centralized control, a common interface for edge device configuration, intent-based overlay network configuration, and a closed-loop mechanism identifying and reverting unintended network changes to the edge [Bor+25] (Contribution 3). The presented approach builds upon the protocol used for flexible and secure edge device linking Contribution 1 and API for unrestricted integration of dynamic overlay networks Contribution 2. Our evaluation results, assuming two contrary edge computation scenarios, show that deployments via our middleware, with centrally enforced optimizations, improve applications' performance by 20 % and 22 %, respectively.

All contributions have been combined into a fully functional open-source middleware, most evaluated in real-world experiments, to demonstrate their seamless integration and function. Additionally, the middleware was published as open-source to encourage further research in these areas: `https://github.com/drasyl/drasyl`.

FUTURE WORK

Beyond our contributions, we discovered several other open research gaps that remain as future work. These gaps include decentralizing network control, refining intent-based networking support, enhancing multi-application support, integrating with cloud deployment systems, and leveraging API for improved resource management.

REDUCTION OF SYSTEM'S DEPENDENCE ON THE CENTRAL NETWORK CONTROLLER The approaches presented in this thesis rely on a central network controller to maintain global view and control over the overlay network. So far, the controller poses a crucial role for the operation and health of the overlay network: New edge devices need to register with the controller, the current state of each device is aggregated at the controller and the controller disseminates new overlay network configurations. Therefore, the controller poses a potential bottleneck and single point of failure (SPOF), limiting the whole system's scalability and availability.

However, some of the controller's functions can be offloaded to the edge devices, reducing the overlay network's dependency on the controller. Selected edge devices can take over managing parts of the overall network. Further, moving closer to edge devices could improve the system's performance.

ENHANCING THE INTENT-BASED PROGRAMMING OF OVERLAY NETWORKING FUNCTIONALITIES The presented system has already been built upon certain concepts known from the intent-based networking (IBN) domain, such as goal-oriented descriptions of network functionalities and the closed-loop mechanism.

However, further improvements to overlay network management may be possible. Currently, intent profiling (Section 2.2) is missing, which enables intents to be described through natural language expressions by application developers. An investigation can be conducted into how existing intent profiler approaches can be utilized with overlay networks at the edge.

IMPROVE MULTI-APPLICATION SUPPORT Although the controller presented in the current approach can manage multiple applications simultaneously, further work is necessary in this regard. Currently, a single application can request all available resources and will also receive them.

To prevent resource monopolization, further mechanisms that improve the co-existence of applications need to be implemented. A mechanism that provides fairness among applications might be included. Various approaches are promising for this: For example, market-based approaches where resources are bid on by applications [NLB19]. This way, resources are allocated based on bids, which can reflect the urgency or importance of the application. Alternatively, fair scheduling can be applied, where resource allocation is based on historical usage. Here, higher priority is given in future allocations to applications that have had fewer resources in the past to balance overall usage [Mad+20]. In this context, it can also be monitored that an application requests only as many

resources as it actually consumes. Applications that intentionally request too many resources are accordingly penalized by being given lower priority.

INCORPORATION WITH APPLICATION DEPLOYMENT SYSTEMS The approaches presented in this work help simplify application deployment challenges in edge environments. This was achieved by creating overlay networks that provide an idealized view of the underlying edge resources. In the cloud computing field, many approaches with sophisticated application management capabilities exist, such as Kubernetes. However, these approaches perform poorly and inefficiently in edge environments, as they make strong assumptions about the behavior and control of the underlying network infrastructure, which is only available in cloud-like environments.

Future research can explore the integration of the proposed approach with existing cloud-based application deployment systems. Specifically, it warrants investigation into how the network control mechanisms presented in this work can enhance the operation of cloud-native application management solutions in edge environments. One avenue to explore is the implementation of the Container Network Interface [CNI24]. This integration would enable Kubernetes pods and other containerized applications to communicate efficiently at the edge, thereby overcoming the limitations posed by conventional network control in edge scenarios. The approach described here promises to be more effective than existing work such as [SAF23], which does not provide means to overcome edge network restrictions efficiently.

IMPROVED RESOURCE MANAGEMENT THROUGH EDGE AI Finding the most suitable edge resources to host distributed applications poses significant challenges due to the unpredictable behavior of these resources. Edge devices often have limited and variable capacities, making it difficult to guarantee sufficient computational and networking resources. This could potentially lead to suboptimal overlay network performance or even complete failure.

AI-based approaches are increasingly used in edge computing. Edge AI is defined as deploying AI technologies within an edge computing environment. Adaptive inference is a promising approach to enhancing resource matchmaking in edge environments by improving the predictability of resource behavior. Increased predictability allows the overlay network management system, for example, to gracefully degrade network performance in response to insufficient resources. This approach helps applications maintain functionality even under adverse conditions.

α

Appendix

A.1 Additional Publications

[Röb+23] Kevin Röbert, Heiko Bornholdt, Mathias Fischer, and Janick Edinger. "Latency-Aware Scheduling for Real-Time Application Support in Edge Computing." In: *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*. 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23). Rome, Italy: ACM, 2023, pages 13–18. DOI: 10.1145/3578354.3592866 (cited on pages 5, 8, 159).

[KBE23] Philipp Kisters, Heiko Bornholdt, and Janick Edinger. "SkABNet: A Data Structure for Efficient Discovery of Streaming Data for IoT." In: *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*. 32nd International Conference on Computer Communications and Networks (ICCCN 2023). Honolulu, HI, USA: IEEE, 2023. DOI: 10.1109/ICCCN58024.2023.10230169.

[Bor+21] Heiko Bornholdt, David Jost, Philipp Kisters, Michel Rottleuthner, Sehrish Shafeeq, Winfried Lamersdorf, Thomas C. Schmidt, and Mathias Fischer. "Smart Urban Data Space for Citizen Science." In: *Volume 80: Conference on Networked Systems 2021 (NetSys 2021)*. Conference on Networked Systems (NetSys 2021). Lübeck, Germany: ECEASST, 2021. DOI: 10.14279/tuj.eceasst.80.1158.

[PKB21] Wolf Posdorfer, Julian Kalinowski, and Heiko Bornholdt. "Towards EU-GDPR Compliant Blockchains with Intentional Forking." In: *Advances in Computer, Communication and Computational Sciences, Proceedings of IC4S 2019*. International Conference on Computer, Communication and Computational Sciences (IC4S 2019). Bangkok, Thailand: Springer Singapore, 2021, pages 649–658. DOI: 10.1007/978-981-15-4409-5_58.

[PBL20] Wolf Posdorfer, Heiko Bornholdt, and Winfried Lamersdorf. "Transaction Dependency Model for Block Minimization in Arbitrary Blockchains." In: *Proceedings of the 2nd International Electronics Communication Conference*. 2nd International Electronics Communication Conference (IECC 2020). Singapore, Singapore: ACM, 2020, pages 59–66. DOI: 10.1145/3409934.3409935.

[Hau+18] Christopher Haubeck, Heiko Bornholdt, Winfried Lamersdorf, Abhishek Chakraborty, and Alexander Fay. "Step-based Evolution Support among Networked Production Automation Systems." In: *at - Automatisierungstechnik* (2018), pages 849–858. DOI: 10.1515/auto-2018-0047.

[Bor+19] Heiko Bornholdt, David Jost, Philipp Kisters, Michel Rottleuthner, Dirk Bade, Winfried Lamersdorf, Thomas C. Schmidt, and Mathias Fischer. "SANE: Smart Networks for Urban Citizen Participation." In: *2019 26th International Conference on Telecommunications (ICT)*. 26th International Conference on Telecommunications (ICT 2019). Hanoi, Vietnman: IEEE, 2019, pages 496–500. DOI: 10.1109/ICT.2019.8798771.

[Pos+20] Wolf Posdorfer, Julian Kalinowski, Heiko Bornholdt, and Winfried Lamersdorf. "Decentralized Billing and Subcontracting of Application Services for Cloud Environment Providers." In: *Advances in Service-Oriented and Cloud Computing, International Workshops of ESOCC 2018*. 8th European Conference On Service-Oriented And Cloud Computing (ESOCC 2018). Como, Italy: Springer Cham, 2020, pages 91–101. DOI: 10.1007/978-3-030-63161-1_7.

A.2 SUPERVISED THESES

[Röb22] Kevin Röbert. "A Secure Context-Aware Middleware for Computation Offloading in Untrustworthy, Open, and Dynamic Edge Environments." Master's thesis. University of Hamburg, Feb. 3, 2022 (cited on page 135).

[Kis24] Liliana Kistenmacher. "QUIC Load Balancing: Enumeration Attacks on QUIC-aware Load Balancers and Countermeasures." Master's thesis. University of Hamburg, Feb. 22, 2024.

[Sem23] Anton Semjonov. “Opportunistic Distributed Computation Offloading using WebAssembly.” Master’s thesis. University of Hamburg, Oct. 16, 2023.

[Brü22] Timon Brüning. “Methoden zur Qualitätsanalyse von Daten in Sensornetzwerken im Kontext einer Smart City.” Master’s thesis. University of Hamburg, Feb. 18, 2022.

[Mai21] Tom Maier. “Dezentrale Access-Control-Konzepte für einen kontrollierten Austausch von Daten zwischen Bürger:innen auf Smart-City-Marktplätzen.” Master’s thesis. University of Hamburg, Apr. 14, 2021.

[Sza21] Phil Szalay. “Methoden für eine Event-basierte Verarbeitung von Sensordaten aus heterogenen Datenquellen im Smart-Home-Bereich am Beispiel eines Incorum-Ansatzes.” Master’s thesis. University of Hamburg, June 29, 2021.

[Abt19] Florian Abt. “Monitoring-Dienste im Web of Things.” Master’s thesis. University of Hamburg, Dec. 1, 2019.

[Bag19] Samuel Bagdassarian. “Predicting Suitable Update Times for IoT Devices.” Master’s thesis. University of Hamburg, Dec. 27, 2019.

[Rei19] Yanneck Reiß. “Caching-Strategien für Daten in Wireless-Sensornetzwerken im Kontext einer Smart City.” Master’s thesis. University of Hamburg, Mar. 5, 2019.

[Vie18] Sebastiano Vierk. “Software-Wartung im Kontext von Firmware-Updates bei IoT-Geräten.” Master’s thesis. University of Hamburg, Dec. 14, 2018.

[Moh22] Patrick Mohr. “Evaluating Impacts on the Randomness Properties of Gossip-Based Peer Sampling when using the drasyl Communication Framework.” Bachelor’s thesis. University of Hamburg, May 23, 2022.

[Bau21] Simon Bauer. “Ursachen, Auswirkungen und Behandlung von Free Riding in Smart-City-Datenräumen.” Bachelor’s thesis. University of Hamburg, June 7, 2021.

[Hir19] Michael Hirsch. “Evaluierung von Question-Answering Systemen für Sensornetze.” Bachelor’s thesis. University of Hamburg, Nov. 28, 2019.

[Ngu19] Minh Hieu Nguyen. "Flexible und automatisierte Provisionierung von IoT-Systemen am Beispiel von Raspberry-Pi-basierten Robotern." Bachelor's thesis. University of Hamburg, May 1, 2019.

[Wut19] Maximilian Wutz. "Entwurf einer Middleware zur Standardisierung von heterogenen Sensordaten am Beispiel von openHAB." Bachelor's thesis. University of Hamburg, Mar. 27, 2019.

[Sch23] Tom Schmolzi. "A Proposal-Based, Decentralized Demand-Side Management Peak-Reduction Algorithm." base.camp project report. University of Hamburg, Mar. 2023.

[Röb21] Kevin Röbert. "Overlay Network Framework for Rapid Development of Distributed P2P Applications." base.camp project report. University of Hamburg, Mar. 2021.

[Töt21] Fin Töter. "Towards a Decentralized Trust-Based Reputation System for Citizen-Driven Marketplaces." base.camp project report. University of Hamburg, Dec. 2021.

[Alt20] Claas Altschaffel. "Virtuelle Assistenten als Benutzungsschnittstelle für Smart-City-Plattformen." Diploma thesis. University of Hamburg, Mar. 23, 2020.

List of figures

1.1 Software-defined overlay networking. . . . 3
1.2 Layers common to all overlay networks. . . . 4
1.3 Contributions of this thesis. . . . 6
2.1 Architecture of SDN. . . . 12
2.2 Interaction of main components of an IBNS. . . . 14
2.3 Example overlay network built on top of an Internet-style underlay. 15
2.4 Example for an unstructured overlay network. . . . 16
2.5 Comparison of $O(N)$ and $O(\log N)$ complexity. . . . 17
2.6 Example for an structured overlay network. . . . 17
2.7 Example for a hybrid overlay network. . . . 18
2.8 Flow of source network address/port translation. . . . 19
2.9 Behavior of different NAT types. . . . 21
2.10 UDP hole punching process. . . . 27
2.11 TLS handshake modes. . . . 29
2.12 The Tasklet system. . . . 31
2.13 The Tasklet life cycle [Bor+23]. . . . 32
3.1 Stakeholders and their interactions in a software-defined overlay networking system. . . . 36
4.1 Overlay network development life cycle. . . . 43
4.2 JXTA architecture. . . . 51
4.3 Parallel Theatre distributed actor-based system. . . . 53
4.4 IronFleet divides a distributed system's behavior into layers of steps for verification. . . . 55
4.5 Mapping of a VPN topology to a SOLID overlay. . . . 59
4.6 Publish/Subscribe application requirements expressed as a set of Rhizoma constraints. . . . 63
5.1 Model of a software-defined overlay networking system. . . . 71
5.2 Software-defined overlay networking is conducted in three steps. 74
5.3 Overview of SDON middleware architecture as UML component diagram. . . . 76
5.4 Communication layer architecture. . . . 78
5.5 Sequential hole punching and key agreement. . . . 83
5.6 NAT traversal process applied by the peer discoverer component. 83

5.7 Communication overlay network provides secure connectivity between all devices. . . . 85
5.8 Service layer architecture. . . . 86
5.9 Interceptor design pattern used to implement various service mechanisms. . . . 88
5.10 Interceptors are chained to create peer-level processing pipelines. 89
5.11 Service overlay providing control of network topology and service guarantees. . . . 91
5.12 Software-defined overlay network programming. . . . 92
5.13 SDON layer. . . . 93
5.14 Software-defined network model. . . . 94
5.15 Device matchmaking applied by intent activator. . . . 97
5.16 Software-defined overlay network. . . . 99
6.1 Yggdrasil (1895) by Lorenz Frølich. . . . 101
6.2 Communication layer protocol messages. . . . 104
6.3 Registration process. . . . 106
6.4 Discovery process. . . . 107
6.5 Communication overlay established by the communication layer. 108
6.6 Endpoints are checked. . . . 109
6.7 Network model describing intent-based overlay networks. . . . 118
6.8 SDON overlay network. . . . 122
6.9 SDON layer protocol. . . . 122
7.1 Application for communication layer evaluation. . . . 128
7.2 Location of devices and server of communication layer evaluation. 129
7.3 Distribution of the time to first bytes sent. . . . 130
7.4 Observed median ping times between server and devices during experiment. . . . 130
7.5 Application for service layer evaluation. . . . 133
7.6 Location of all environments and nodes with their respective roles. . . . 134
7.7 Source and filtered output images of image convolution task. . . 139
7.8 Average transmission times in computation offloading. . . . 140
7.9 Histograms showing the density of the completion times. . . . 141
7.10 Real-time application used for first evaluation of SDON layer. . 145
7.11 Routes used to contact all servers. . . . 146
7.12 Path latency and per-relay load in dependence on the number of considered relays per path. . . . 148
7.13 Computation-offloading application used for first evaluation of SDON layer. . . . 150

7.14 Task completion time is reduced when fast providers are placed at the end of the processing pipeline. 151
7.15 Rearrangements in in dependence on the number of optionally placed providers. 154

List of tables

1.1 The outline of the thesis. 8

2.1 Mapping of traditional NAT types to behavior policies. 23

4.1 Common API for structured overlay networks. 57

4.2 Related work in overlay network construction systems. 66

5.1 List of intents describing network model shown Figure 5.14. . . 96

6.1 API for implementing mechanisms to enforce Service goals. . . 110

6.2 Mapping between service goals and mechanisms. 115

6.3 Lua-API describing desired overlay networks. 120

7.1 Devices, networks/ISPs, operating systems and NAT types used in the evaluation [Röb22]. 135

List of snippets

4.1 Example VR application specification in FogBrain. 46
4.2 Service descriptions are directly embedded to the source code in Jadex. 47
4.3 Implementation of the Chord graph in the SQL-like overlay network specification language. 49
4.4 Imperative verification example. 55
4.5 Pod manifest for an Nginx and MariaDB pod with resource management and monitoring mechanisms. 61
6.1 Implementation of a service mechanism that enforces confidentiality using encryption. 111
6.2 Creation of a processing pipeline that enforces service goals reliability and confidentiality. 116
6.3 Example usage of network modeling API. 121
7.1 SDON network model for NAT-aware routing. 146
7.2 SDON network model for computation offloading. 152

Bibliography

[ST23] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. 4th edition. 2023. ISBN: 978-90-815406-4-3 (cited on pages 1, 15, 40, 41, 88).

[Ren+19] Jinke Ren, Guanding Yu, Yinghui He, and Geoffrey Ye Li. "Collaborative Cloud and Edge Computing for Latency Minimization." In: *IEEE Transactions on Vehicular Technology* 68.5 (May 2019), pages 5031–5044. ISSN: 1939-9359. DOI: 10.1109/TVT.2019.2904244 (cited on page 1).

[Mao+17] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective." In: *IEEE Communications Surveys & Tutorials* 19.4 (2017), pages 2322–2358. ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2745201 (cited on page 1).

[STS08] Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. "Overlay Weaver: An overlay construction toolkit." In: *Computer Communications* 31.2 (2008). Special Issue: Foundation of Peer-to-Peer Computing, pages 402–412. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2007.08.002. URL: https://www.sciencedirect.com/science/article/pii/S0140366407002939 (cited on pages 2, 5, 65, 66).

[Rod+04] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. "MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks." In: *First Symposium on Networked Systems Design and Implementation (NSDI 04)*. San Francisco, CA: USENIX Association, Mar. 2004. URL: https://www.usenix.org/conference/nsdi-04/macedon-methodology-automatically-creating-evaluating-and-designing-overlay (cited on pages 2, 5, 48, 65, 66).

[BP12] Lars Braubach and Alexander Pokahr. "Developing distributed systems with active components and Jadex." In: *Scalable Computing: Practice and Experience* 13.2 (2012), pages 100–120 (cited on pages 2, 5, 46, 47, 53, 66, 67).

[BL18] Filippo Battaglia and Lucia Lo Bello. “A novel JXTA-based architecture for implementing heterogenous Networks of Things.” In: *Computer Communications* 116 (2018), pages 35–62. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2017.11.002. URL: https://www.sciencedirect.com/science/article/pii/S0140366417303584 (cited on pages 2, 5, 66, 67).

[Lig09] Lightbend, Inc. *Akka: Build, Operate, and Secure Low Latency Systems*. July 2009. URL: https://akka.io/ (visited on 06/04/2024) (cited on pages 2, 5, 53, 66, 159).

[Kre+15] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. “Software-Defined Networking: A Comprehensive Survey.” In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pages 14–76. ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2371999 (cited on page 3).

[Ope24] Open Networking Foundation. *Open Network Operating System*. July 16, 2024. URL: https://opennetworking.org/onos/ (visited on 09/25/2024) (cited on pages 3, 70).

[Cle+22] Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. *Intent-Based Networking - Concepts and Definitions*. RFC 9315. Oct. 2022. DOI: 10.17487/RFC9315. URL: https://www.rfc-editor.org/info/rfc9315 (cited on pages 3, 13).

[LF23] Aris Leivadeas and Matthias Falkner. “A Survey on Intent-Based Networking.” In: *IEEE Communications Surveys & Tutorials* 25.1 (2023), pages 625–655. ISSN: 1553-877X. DOI: 10.1109/COMST.2022.3215919 (cited on pages 3, 13, 14, 77, 92).

[Tar10] Sasu Tarkoma. *Overlay Networks: Toward Information Networking*. Auerbach Publications, 2010. ISBN: 978-1-4398-1373-7. DOI: 10.1201/9781439813737 (cited on pages 4, 15).

[BR23] Heiko Bornholdt and Kevin Röbert. *drasyl - a middleware for rapid development of distributed applications*. Software. Version 0.10.0. Jan. 2023. DOI: 10.25592/uhhfdm.14206. URL: https://github.com/drasyl/drasyl (cited on pages 4, 101).

[BRK21] Heiko Bornholdt, Kevin Röbert, and Philipp Kisters. “Accessing Smart City Services in Untrustworthy Environments via Decentralized Privacy-Preserving Overlay Networks.” In: *2021 IEEE International Conference on Service-Oriented System Engineering (IEEE*

SOSE 2021). 15th IEEE International Conference On Service-Oriented System Engineering (IEEE SOSE 2021). Oxford, United Kingdom: IEEE, 2021, pages 144–149. DOI: 10.1109/SOSE52839.2021.00021 (cited on pages 5, 8).

[BRF23a] Heiko Bornholdt, Kevin Röbert, and Mathias Fischer. "Low-Latency TLS 1.3-Aware Hole Punching." In: *ICC 2023 - IEEE International Conference on Communications*. IEEE International Conference on Communications (ICC) 2023. Rome, Italy: IEEE, 2023, pages 1481–1486. DOI: 10.1109/ICC45041.2023.10279326 (cited on pages 5, 8, 27, 29, 127, 158).

[Bor+23] Heiko Bornholdt, Kevin Röbert, Martin Breitbach, Mathias Fischer, and Janick Edinger. "Measuring the Edge: A Performance Evaluation of Edge Offloading." In: *2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. 2nd Workshop on Serverless computing for pervasive cloud-edge-device systems and services (*LESS'24). Atlanta, GA, USA: IEEE, 2023, pages 212–218. DOI: 10.1109/PerComWorkshops56833.2023.10150261 (cited on pages 5, 8, 32, 132, 159).

[Röb+23] Kevin Röbert, Heiko Bornholdt, Mathias Fischer, and Janick Edinger. "Latency-Aware Scheduling for Real-Time Application Support in Edge Computing." In: *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking*. 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '23). Rome, Italy: ACM, 2023, pages 13–18. DOI: 10.1145/3578354.3592866 (cited on pages 5, 8, 159).

[Bor+25] Heiko Bornholdt, Kevin Röbert, Stefan Schulte, Janick Edinger, and Mathias Fischer. "A Software-Defined Overlay Networking Middleware for a Simplified Deployment of Distributed Application at the Edge." In: *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*. SAC '25. Catania, Italy: Association for Computing Machinery, 2025 (cited on pages 7, 8, 160).

[BBP21] Heiko Bornholdt, Dirk Bade, and Wolf Posdorfer. "Incorum: A Citizen-Centric Sensor Data Marketplace for Urban Participation." In: *Advances in Computer, Communication and Computational Sciences, Proceedings of IC4S 2019*. International Conference on Computer, Communication and Computational Sciences (IC4S

2019). Bangkok, Thailand: Springer Singapore, 2021, pages 659–669. DOI: 10.1007/978-981-15-4409-5_59.

[Bor21] Heiko Bornholdt. "Towards Citizen-Centric Marketplaces for Urban Sensed Data." In: *Advances in Service-Oriented and Cloud Computing, International Workshops of ESOCC 2020*. 8th European Conference On Service-Oriented And Cloud Computing (ESOCC 2020). Heraklion, Crete, Greece: Springer Cham, 2021, pages 140–150. DOI: 10.1007/978-3-030-71906-7_12.

[Bor23a] Heiko Bornholdt. *NatPy - python-based network address translator with configurable mapping, allocation, and filtering behavior for Netfilter NFQUEUE*. Software. Dec. 2023. DOI: 10.25592/uhhfdm.14208. URL: https://github.com/HeikoBornholdt/NatPy (cited on page 147).

[Bor23b] Heiko Bornholdt. *netty-tun - netty channel communicating via TUN devices*. Software. Version 1.2.2. Apr. 2023. DOI: 10.25592/uhhfdm.14210. URL: https://github.com/drasyl/netty-tun (cited on page 116).

[BRF23b] Heiko Bornholdt, Kevin Röbert, and Mathias Fischer. *Low-Latency TLS 1.3-Aware Hole Punching Proof of Concept*. Software. Jan. 2023. DOI: 10.25592/uhhfdm.16613. URL: https://github.com/HeikoBornholdt/tls-hole-punching (cited on page 129).

[TFW21] Andrew S. Tanenbaum, Nick Feamster, and David Wetherall. *Computer Networks*. 6th edition. Pearson Education, 2021. ISBN: 978-1-292-37406-2 (cited on pages 12, 71).

[Fis22] Mathias Fischer. *Lecture notes in Computer Networks*. 2022 (cited on page 12).

[Doy22] Jeff Doyle. *Intent-Based Networking*. John Wiley & Sons, Inc., 2022. ISBN: 978-1-119-87905-3 (cited on page 13).

[Toy21] Mehmet Toy. *Future Networks, Services and Management: Underlay and Overlay, Edge, Applications, Slicing, Cloud, Space, AI/ML, and Quantum Computing*. Springer Nature Switzerland AG, 2021. ISBN: 978-3-030-81960-6. DOI: 10.1007/978-3-030-81961-3 (cited on page 15).

[Sch10] Ingo Scholtes. "Harnessing Complex Structures and Collective Dynamics in Large Networked Computing Systems." PhD thesis. Universität Trier, 2010 (cited on pages 15–17).

[Gnu03] Gnutella Developer Forum. *The Annotated Gnutella Protocol Specification v0.4*. 2003. URL: https://rfc-gnutella.sourceforge.net/developer/stable/index.html (visited on 08/04/2024) (cited on page 16).

[Cha+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. "Making gnutella-like P2P systems scalable." In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pages 407–418. ISBN: 1581137354. DOI: 10.1145/863955.864000. URL: https://doi.org/10.1145/863955.864000 (cited on page 16).

[RF02] Matei Ripeanu and Ian Foster. "Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems." In: *Peer-to-Peer Systems*. Edited by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pages 85–93. ISBN: 978-3-540-45748-0 (cited on page 16).

[MIF02] R. Matei, A. Iamnitchi, and P. Foster. "Mapping the Gnutella network." In: *IEEE Internet Computing* 6.1 (2002), pages 50–57. DOI: 10.1109/4236.978369 (cited on page 16).

[Sto+03] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for Internet applications." In: *IEEE/ACM Transactions on Networking* 11.1 (Feb. 2003), pages 17–32. ISSN: 1558-2566. DOI: 10.1109/TNET.2002.808407 (cited on pages 17, 44, 49, 58, 88, 112).

[GG09] Wojciech Galuba and Sarunas Girdzijauskas. "Peer to Peer Overlay Networks: Structure, Routing and Maintenance." In: *Encyclopedia of Database Systems*. Edited by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pages 2056–2061. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1215. URL: https://doi.org/10.1007/978-0-387-39940-9_1215 (cited on page 18).

[Mos+96] Robert Moskowitz, Daniel Karrenberg, Yakov Rekhter, Eliot Lear, and Geert Jan de Groot. *Address Allocation for Private Internets*. RFC

1918. Feb. 1996. DOI: 10.17487/RFC1918. URL: https://www.rfc-editor.org/info/rfc1918 (cited on page 18).

[HS99] Matt Holdrege and Pyda Srisuresh. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663. Aug. 1999. DOI: 10.17487/RFC2663. URL: https://www.rfc-editor.org/info/rfc2663 (cited on page 18).

[BC02] Scott W. Brim and Brian E. Carpenter. *Middleboxes: Taxonomy and Issues*. RFC 3234. Feb. 2002. DOI: 10.17487/RFC3234. URL: https://www.rfc-editor.org/info/rfc3234 (cited on pages 19, 72).

[ES01] Kjeld Borch Egevang and Pyda Srisuresh. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. Jan. 2001. DOI: 10.17487/RFC3022. URL: https://www.rfc-editor.org/info/rfc3022 (cited on page 19).

[Qia24] Liu Qiaoqiao. *What Is Network Address Translation (NAT)?* June 17, 2024. URL: https://info.support.huawei.com/info-finder/encyclopedia/en/NAT.html (visited on 07/29/2024) (cited on page 19).

[Per+13] Simon Perreault, Ikuhei Yamagata, Shin Miyakawa, Akira Nakagawa, and Hiroyuki Ashida. *Common Requirements for Carrier-Grade NATs (CGNs)*. RFC 6888. Apr. 2013. DOI: 10.17487/RFC6888. URL: https://www.rfc-editor.org/info/rfc6888 (cited on page 19).

[ICA11] ICANN. *Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied*. Feb. 3, 2011. URL: https://itp.cdn.icann.org/en/files/announcements/release-03feb11-en.pdf (visited on 08/02/2024) (cited on page 19).

[Ros+03] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Joel Weinberger. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489. Mar. 2003. DOI: 10.17487/RFC3489. URL: https://www.rfc-editor.org/info/rfc3489 (cited on page 20).

[Pen+16] Reinaldo Penno, Simon Perreault, Mohamed Boucadair, Senthil Sivakumar, and Kengo Naito. *Updates to Network Address Translation (NAT) Behavioral Requirements*. RFC 7857. Apr. 2016. DOI: 10.17487/RFC7857. URL: https://www.rfc-editor.org/info/rfc7857 (cited on page 22).

[REH09] Roberto Roverso, Sameh El-Ansary, and Seif Haridi. “NATCracker: NAT Combinations Matter.” In: *2009 Proceedings of 18th International Conference on Computer Communications and Networks*. Aug. 2009, pages 1–7. DOI: 10.1109/ICCCN.2009.5235278 (cited on pages 22, 24).

[Hät+10] Seppo Hätönen, Aki Nyrhinen, Lars Eggert, Stephen Strowes, Pasi Sarolahti, and Markku Kojo. “An experimental study of home gateway characteristics.” In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC ’10. Melbourne, Australia: Association for Computing Machinery, 2010, pages 260–266. ISBN: 9781450304832. DOI: 10.1145/1879141.1879174. URL: https://doi.org/10.1145/1879141.1879174 (cited on pages 23, 24, 109).

[JA07] Cullen Fluffy Jennings and Francois Audet. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. RFC 4787. Jan. 2007. DOI: 10.17487/RFC4787. URL: https://www.rfc-editor.org/info/rfc4787 (cited on pages 23, 24).

[Sch+03] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550. July 2003. DOI: 10.17487/RFC3550. URL: https://www.rfc-editor.org/info/rfc3550 (cited on page 23).

[For+08] Bryan Ford, Saikat Guha, Kaushik Biswas, Senthil Sivakumar, and Pyda Srisuresh. *NAT Behavioral Requirements for TCP*. RFC 5382. Oct. 2008. DOI: 10.17487/RFC5382. URL: https://www.rfc-editor.org/info/rfc5382 (cited on page 24).

[Guh+09] Saikat Guha, Bryan Ford, Senthil Sivakumar, and Pyda Srisuresh. *NAT Behavioral Requirements for ICMP*. RFC 5508. Apr. 2009. DOI: 10.17487/RFC5508. URL: https://www.rfc-editor.org/info/rfc5508 (cited on page 24).

[Haa+16] Steffen Haas, Shankar Karuppayah, Selvakumar Manickam, Max Mühlhäuser, and Mathias Fischer. “On the resilience of P2P-based botnet graphs.” In: *2016 IEEE Conference on Communications and Network Security (CNS)*. Oct. 2016, pages 225–233. DOI: 10.1109/CNS.2016.7860489 (cited on pages 24, 33, 72, 147).

[WP17] Liang Wang and Ivan Pustogarov. *Towards Better Understanding of Bitcoin Unreachable Peers*. 2017. arXiv: 1709.06837 [cs.NI] (cited on pages 24, 33, 72).

[CK13] Stuart Cheshire and Marc Krochmal. *NAT Port Mapping Protocol (NAT-PMP)*. RFC 6886. Apr. 2013. DOI: 10.17487/RFC6886. URL: https://www.rfc-editor.org/info/rfc6886 (cited on pages 24, 26, 81).

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003. ISBN: 9780321200686 (cited on page 25).

[Win+13] Dan Wing, Stuart Cheshire, Mohamed Boucadair, Reinaldo Penno, and Paul Selkirk. *Port Control Protocol (PCP)*. RFC 6887. Apr. 2013. DOI: 10.17487/RFC6887. URL: https://www.rfc-editor.org/info/rfc6887 (cited on pages 25, 81).

[BPW13] Mohamed Boucadair, Reinaldo Penno, and Dan Wing. *Universal Plug and Play (UPnP) Internet Gateway Device - Port Control Protocol Interworking Function (IGD-PCP IWF)*. RFC 6970. July 2013. DOI: 10.17487/RFC6970. URL: https://www.rfc-editor.org/info/rfc6970 (cited on pages 25, 81).

[Lee96] Marcus D. Leech. *SOCKS Protocol Version 5*. RFC 1928. Mar. 1996. DOI: 10.17487/RFC1928. URL: https://www.rfc-editor.org/info/rfc1928 (cited on page 26).

[Red+20] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 8656. Feb. 2020. DOI: 10.17487/RFC8656. URL: https://www.rfc-editor.org/info/rfc8656 (cited on pages 26, 81).

[GTF04] Saikat Guha, Yutaka Takeda, and Paul Francis. “NUTSS: a SIP-based approach to UDP and TCP network connectivity.” In: *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*. FDNA ’04. Portland, Oregon, USA: Association for Computing Machinery, 2004, pages 43–48. ISBN: 158113942X. DOI: 10.1145/1016707.1016715. URL: https://doi.org/10.1145/1016707.1016715 (cited on pages 26, 29).

[DA08] Luca Deri and Richard Andrews. “N2N: A Layer Two Peer-to-Peer VPN.” In: *Resilient Networks and Services*. Edited by David Hausheer and Jürgen Schönwälder. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pages 53–64. ISBN: 978-3-540-70587-1 (cited on pages 26, 72).

[FSK05] Bryan Ford, Pyda Srisuresh, and Dan Kegel. "Peer-to-Peer Communication Across Network Address Translators." In: *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. Anaheim, CA: USENIX Association, Apr. 2005. URL: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/peer-peer-communication-across-network-address (cited on page 26).

[Big+05] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, and Adrian Perrig. "NATBLASTER: Establishing TCP Connections Between Hosts Behind NATs." In: *Proceedings of the ACM SIGCOMM ASIA Workshop*. Apr. 2005 (cited on page 26).

[GF05] Saikat Guha and Paul Francis. "Characterization and measurement of TCP traversal through NATs and firewalls." In: *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement*. IMC '05. Berkeley, CA: USENIX Association, 2005, page 18 (cited on page 28).

[Edd22] Wesley Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293. URL: https://www.rfc-editor.org/info/rfc9293 (cited on page 28).

[Res18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446 (cited on pages 29, 30, 82).

[Sch+16] Dominik Schäfer, Janick Edinger, Sebastian VanSyckel, Justin Mazzola Paluska, and Christian Becker. "Tasklets: Overcoming Heterogeneity in Distributed Computing Systems." In: *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. June 2016, pages 156–161. DOI: 10.1109/ICDCSW.2016.22 (cited on page 31).

[LK22] Phillip A. Laplante and Mohamad H. Kassab. *Requirements Engineering for Software and Systems*. 4th edition. Auerbach Publications, 2022. ISBN: 978-0-367-65452-8. DOI: 10.1201/9781003129509 (cited on pages 35, 37, 41).

[Mac07] Leszek A. Maciaszek. *Requirements Analysis and Systems Design*. 3rd edition. Pearson Education, 2007. ISBN: 978-0-321-44036-5 (cited on pages 37, 40).

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. "A network in a laptop: rapid prototyping for software-defined networks." In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: Association for Computing Machinery, 2010. ISBN: 9781450304092. DOI: 10.1145/1868447.1868466. URL: https://doi.org/10.1145/1868447.1868466 (cited on pages 38, 147).

[Fis12] Mathias Fischer. *Construction of Attack-Resilient and Efficient Overlay-Topologies for Large-Scale P2P-based IPTV Infrastructures*. Cuvillier Verlag, 2012. ISBN: 978-3954042081 (cited on pages 39, 40).

[IEE90] IEEE. "IEEE Standard Glossary of Software Engineering Terminology." In: *IEEE Std 610.12-1990* (Dec. 1990), pages 1–84. DOI: 10.1109/IEEESTD.1990.101064 (cited on page 40).

[FMP05] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. "A Model-Based Approach for Robustness Testing." In: *Testing of Communicating Systems*. Edited by Ferhat Khendek and Rachida Dssouli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 333–348. ISBN: 978-3-540-32076-0 (cited on page 40).

[Hil90] Mark D. Hill. "What is scalability?" In: *SIGARCH Comput. Archit. News* 18.4 (Dec. 1990), pages 18–21. ISSN: 0163-5964. DOI: 10.1145/121973.121975. URL: https://doi.org/10.1145/121973.121975 (cited on page 40).

[Roß11] Michael Roßberg. *Skalierbare Autokonfiguration sabotageresistenter virtueller privater Netze*. Cuvillier Verlag, 2011. ISBN: 978-3869557755 (cited on page 41).

[RF20] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. 1st edition. O'Reilly Media, 2020. ISBN: 978-1-492-04345-4 (cited on pages 43, 44, 50, 54, 56, 61, 88).

[Lua+05] Eng Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. "A survey and comparison of peer-to-peer overlay network schemes." In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), pages 72–93. ISSN: 1553-877X. DOI: 10.1109/COMST.2005.1610546 (cited on pages 44, 45).

[AS04] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. "A survey of peer-to-peer content distribution technologies." In: *ACM Comput. Surv.* 36.4 (Dec. 2004), pages 335–371. ISSN: 0360-0300.

doi: 10.1145/1041680.1041681. url: https://doi.org/10.1145/1041680.1041681 (cited on page 44).

[KS10] Jinu Kurian and Kamil Sarac. "A survey on the design, applications, and enhancements of application-layer overlay networks." In: *ACM Comput. Surv.* 43.1 (Dec. 2010). issn: 0360-0300. doi: 10.1145/1824795.1824800. url: https://doi.org/10.1145/1824795.1824800 (cited on pages 44, 58, 60).

[Pas12] Andrea Passarella. "A survey on content-centric technologies for the current Internet: CDN and P2P solutions." In: *Computer Communications* 35.1 (2012), pages 1–32. issn: 0140-3664. doi: https://doi.org/10.1016/j.comcom.2011.10.005. url: https://www.sciencedirect.com/science/article/pii/S0140366411003173 (cited on page 44).

[RM06] John Risson and Tim Moors. "Survey of research towards robust peer-to-peer networks: Search methods." In: *Computer Networks* 50.17 (2006), pages 3485–3521. issn: 1389-1286. doi: https://doi.org/10.1016/j.comnet.2006.02.001. url: https://www.sciencedirect.com/science/article/pii/S1389128606000223 (cited on page 44).

[Rat+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. "A scalable content-addressable network." In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. San Diego, California, USA: Association for Computing Machinery, 2001, pages 161–172. isbn: 1581134118. doi: 10.1145/383059.383072. url: https://doi.org/10.1145/383059.383072 (cited on page 44).

[RD01] Antony Rowstron and Peter Druschel. "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems." In: *Middleware 2001*. Edited by Rachid Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pages 329–350. isbn: 978-3-540-45518-9 (cited on page 44).

[Cla+01] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System." In: *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability Berkeley, CA, USA, July 25–26, 2000 Proceedings*. Edited by

Hannes Federrath. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pages 46–66. ISBN: 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: https://doi.org/10.1007/3-540-44702-4_4 (cited on page 44).

[KR02] T. Klingberg and Manfredi R. *Gnutella 0.6*. June 2002. URL: https://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html (visited on 06/04/2024) (cited on page 44).

[KR04] T. Klingberg and Manfredi R. *The FastTrack Protocol*. Apr. 14, 2004. URL: http://cvs.berlios.de/cgi-bin/viewcvs.cgi/gift-fasttrack/giFT-FastTrack/PROTOCOL?rev=HEAD%5C&content-type=text/vnd.viewcvs-markup (visited on 12/17/2012) (cited on page 44).

[FB20] Stefano Forti and Antonio Brogi. "Continuous Reasoning for Managing Next-Gen Distributed Applications." In: *Electronic Proceedings in Theoretical Computer Science* 325 (Sept. 2020), pages 164–177. ISSN: 2075-2180. DOI: 10.4204/eptcs.325.22. URL: http://dx.doi.org/10.4204/EPTCS.325.22 (cited on pages 45, 46, 66).

[Bro+20] Antonio Brogi, Stefano Forti, Carlos Guerrero, and Isaac Lera. "Towards Declarative Decentralised Application Management in the Fog." In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Oct. 2020, pages 223–230. DOI: 10.1109/ISSREW51248.2020.00077 (cited on pages 45, 66).

[CM12] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. 5th edition. Springer Berlin, Heidelberg, 2012. ISBN: 978-3-642-55481-0. DOI: 10.1007/978-3-642-55481-0 (cited on page 46).

[Any17] Anyscale, Inc. *Ray: Productionizing and scaling Python ML workloads simply*. May 21, 2017. URL: https://www.ray.io/ (visited on 06/04/2024) (cited on page 46).

[Lop97] Cristina Videira Lopes. "D: A Language Framework for Distributed Programming." PhD thesis. Northeastern University, 1997 (cited on page 48).

[Kil+07] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. "Mace: language support for building distributed systems." In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: Association for Computing Machinery, 2007, pages 179–188. ISBN: 9781595936332. DOI: 10.1145/1250734.1250755. URL: https://doi.org/10.1145/1250734.1250755 (cited on page 48).

[WCB01] Matt Welsh, David Culler, and Eric Brewer. "SEDA: an architecture for well-conditioned, scalable internet services." In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: Association for Computing Machinery, 2001, pages 230–243. ISBN: 1581133898. DOI: 10.1145/502034.502057. URL: https://doi.org/10.1145/502034.502057 (cited on page 48).

[BB05] Stefan Behnel and Alejandro Buchmann. "Overlay Networks – Implementation by Specification." In: *Middleware 2005*. Edited by Gustavo Alonso. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 401–410. ISBN: 978-3-540-32269-6 (cited on pages 48, 49, 66).

[Loo+05] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. "Implementing declarative overlays." In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, pages 75–90. ISBN: 1595930795. DOI: 10.1145/1095810.1095818. URL: https://doi.org/10.1145/1095810.1095818 (cited on page 48).

[Beh07] Stefan Behnel. "A Model Driven Architecture for Adaptable Overlay Networks." PhD thesis. Technische Universität Darmstadt, 2007 (cited on page 49).

[BB07] Stefan Behnel and Alejandro Buchmann. "Models and Languages for Overlay Networks." In: *Databases, Information Systems, and Peer-to-Peer Computing*. Edited by Gianluca Moro, Sonia Bergamaschi, Sam Joseph, Jean-Henry Morin, and Aris M. Ouksel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 211–218. ISBN: 978-3-540-71661-7 (cited on page 49).

[Gon01] Li Gong. "JXTA: a network programming environment." In: *IEEE Internet Computing* 5.3 (May 2001), pages 88–95. ISSN: 1941-0131. DOI: 10.1109/4236.935182 (cited on pages 50, 66, 67).

[Pro20] Protocol Labs. *libp2p - Modular Peer-to-Peer Networking Stack*. 2020. URL: https://libp2p.io/ (visited on 06/04/2024) (cited on pages 50, 66, 67).

[Tra+02] Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. "Project JXTA Virtual Network." In: (2002) (cited on page 50).

[Ben14] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System*. 2014. arXiv: 1407.3561 [cs.NI] (cited on page 51).

[Agh86] Gul A. Agha. *Actors: A Model of Concurrent Computation In Distributed Systems*. Technical Report. Massachusetts Institute of Technology, 1986 (cited on page 52).

[CN22] Franco Cicirelli and Libero Nigro. "Performance Prediction of Scalable Multi-agent Systems Using Parallel Theatre." In: *Intelligent Sustainable Systems*. Edited by Atulya K. Nagar, Dharm Singh Jat, Gabriela Marín-Raventós, and Durgesh Kumar Mishra. Singapore: Springer Nature Singapore, 2022, pages 45–64. ISBN: 978-981-16-6369-7 (cited on page 52).

[Fer99] Jacques Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201360489 (cited on page 52).

[Lee06] E.A. Lee. "The problem with threads." In: *Computer* 39.5 (May 2006), pages 33–42. ISSN: 1558-0814. DOI: 10.1109/MC.2006.180 (cited on page 52).

[Nig21] Libero Nigro. "Parallel Theatre: An actor framework in Java for high performance computing." In: *Simulation Modelling Practice and Theory* 106 (2021), page 102189. ISSN: 1569-190X. DOI: https://doi.org/10.1016/j.simpat.2020.102189. URL: https://www.sciencedirect.com/science/article/pii/S1569190X20301283 (cited on pages 53, 66, 159).

[AZ17] Austin Aske and Xinghui Zhao. "An Actor-Based Framework for Edge Computing." In: *Proceedings of The10th International Conference on Utility and Cloud Computing*. UCC '17. Austin, Texas, USA: Association for Computing Machinery, 2017, pages 199–

200. ISBN: 9781450351492. DOI: 10.1145/3147213.3149214. URL: https://doi.org/10.1145/3147213.3149214 (cited on pages 53, 66, 159).

[BPR01] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. "Developing Multi-agent Systems with JADE." In: *Intelligent Agents VII Agent Theories Architectures and Languages*. Edited by Cristiano Castelfranchi and Yves Lespérance. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pages 89–103. ISBN: 978-3-540-44631-6 (cited on page 53).

[Boi+13] Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, Alessandro Ricci, and Andrea Santi. "Multi-agent oriented programming with JaCaMo." In: *Science of Computer Programming* 78.6 (2013). Special section: The Programming Languages track at the 26th ACM Symposium on Applied Computing (SAC 2011) & Special section on Agent-oriented Design Methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments, pages 747–761. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2011.10.004. URL: https://www.sciencedirect.com/science/article/pii/S016764231100181X (cited on page 53).

[CNS20] Franco Cicirelli, Libero Nigro, and Paolo F. Sciammarella. "Seamless development in Java of distributed real-time systems using actors." In: *International Journal of Simulation and Process Modelling* 15 (2020), pages 13–29. DOI: 10.1504/IJSPM.2020.106965 (cited on page 53).

[SWT17] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and proving with distributed protocols." In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017). DOI: 10.1145/3158116. URL: https://doi.org/10.1145/3158116 (cited on page 54).

[Haw+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. "IronFleet: proving practical distributed systems correct." In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pages 1–17. ISBN: 9781450338349. DOI: 10.1145/2815400.2815428. URL: https://doi.org/10.1145/2815400.2815428 (cited on pages 54, 55).

[Wil+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. "Verdi: a framework for implementing and formally verifying distributed systems." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pages 357–368. ISBN: 9781450334686. DOI: 10.1145/2737924.2737958. URL: https://doi.org/10.1145/2737924.2737958 (cited on page 54).

[Lam94] Leslie Lamport. "The temporal logic of actions." In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pages 872–923. ISSN: 0164-0925. DOI: 10.1145/177492.177726. URL: https://doi.org/10.1145/177492.177726 (cited on page 54).

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002. URL: https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/ (cited on page 54).

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming." In: *Commun. ACM* 12.10 (Oct. 1969), pages 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259 (cited on page 54).

[Lam77] L. Lamport. "Proving the Correctness of Multiprocess Programs." In: *IEEE Transactions on Software Engineering* SE-3.2 (Mar. 1977), pages 125–143. ISSN: 1939-3520. DOI: 10.1109/TSE.1977.229904 (cited on page 54).

[Yua+14] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems." In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pages 249–265. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan (cited on page 55).

[Lip75] Richard J. Lipton. "Reduction: a method of proving properties of parallel programs." In: *Commun. ACM* 18.12 (Dec. 1975), pages 717–

721. ISSN: 0001-0782. DOI: 10.1145/361227.361234. URL: https://doi.org/10.1145/361227.361234 (cited on page 55).

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Edited by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pages 348–370. ISBN: 978-3-642-17511-4 (cited on page 55).

[Gar+05] Pedro García, Carles Pairot, Rubén Mondéjar, Jordi Pujol, Helio Tejedor, and Robert Rallo. "PlanetSim: A New Overlay Network Simulation Framework." In: *Software Engineering and Middleware*. Edited by Thomas Gschwind and Cecilia Mascolo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pages 123–136. ISBN: 978-3-540-31975-7 (cited on page 55).

[BHK07] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. "OverSim: A Flexible Overlay Network Simulation Framework." In: *2007 IEEE Global Internet Symposium*. May 2007, pages 79–84. DOI: 10.1109/GI.2007.4301435 (cited on page 55).

[Ope01] OpenSim Ltd. *OMNeT++ Discrete Event Simulator*. Feb. 15, 2001. URL: https://omnetpp.org/ (visited on 06/04/2024) (cited on page 55).

[Rob14] Stewart Robinson. *Simulation: The Practice of Model Development and Use*. 2nd edition. Palgrave MacMillan, 2014. ISBN: 978-1137328021 (cited on page 55).

[Nai+06] Stephen Naicken, Anirban Basu, Barnaby Livingston, and Sethalat Rodhetbhai. "A survey of peer-to-peer network simulators." In: *Proceedings of The Seventh Annual Postgraduate Symposium, Liverpool, UK*. Volume 2. 2006, page 13 (cited on page 56).

[Dab+03] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. "Towards a Common API for Structured Peer-to-Peer Overlays." In: *Peer-to-Peer Systems II*. Edited by M. Frans Kaashoek and Ion Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pages 33–44. ISBN: 978-3-540-45172-3 (cited on pages 56, 57, 65).

[Law24] Averill M. Law. *Simulation Modeling and Analysis*. 6th edition. McGraw Hill, 2024. ISBN: 9781264268245 (cited on page 56).

[Sto+02] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. "Internet indirection infrastructure." In: *SIGCOMM Comput. Commun. Rev.* 32.4 (Aug. 2002), pages 73–86. ISSN: 0146-4833. DOI: 10.1145/964725.633033. URL: https://doi.org/10.1145/964725.633033 (cited on pages 58, 65–67).

[For04] Bryan Ford. "Unmanaged Internet Protocol: taming the edge network management crisis." In: *SIGCOMM Comput. Commun. Rev.* 34.1 (Jan. 2004), pages 93–98. ISSN: 0146-4833. DOI: 10.1145/972374.972391. URL: https://doi.org/10.1145/972374.972391 (cited on pages 58, 65–67, 72).

[RSS10] Michael Rossberg, Guenter Schaefer, and Thorsten Strufe. "Distributed Automatic Configuration of Complex IPsec-Infrastructures." In: *Journal of Network and Systems Management* 18.3 (Sept. 2010), pages 300–326. ISSN: 1573-7705. DOI: 10.1007/s10922-010-9168-7. URL: https://doi.org/10.1007/s10922-010-9168-7 (cited on pages 58, 59, 65–67).

[SK05] Karen Seo and Stephen Kent. *Security Architecture for the Internet Protocol*. RFC 4301. Dec. 2005. DOI: 10.17487/RFC4301. URL: https://www.rfc-editor.org/info/rfc4301 (cited on page 58).

[Zer14] ZeroTier, Inc. *ZeroTier - Global Area Networking*. 2014. URL: https://github.com/zerotier/ZeroTierOne (visited on 06/04/2024) (cited on pages 59, 66, 67).

[Tai19] Tailscale Inc. *Best VPN Service for Secure Networks*. 2019. URL: https://tailscale.com/ (visited on 06/04/2024) (cited on pages 59, 66, 67).

[Zer24a] ZeroTier, Inc. *The Protocol | ZeroTier Documentation*. Apr. 12, 2024. URL: https://docs.zerotier.com/protocol/ (visited on 06/04/2024) (cited on page 59).

[Zer24b] ZeroTier, Inc. *libzt: ZeroTier's Lightweight End-to-End Encrypted Network Virtualization Library*. Jan. 16, 2024. URL: https://github.com/zerotier/libzt (visited on 01/16/2024) (cited on page 59).

[Don17] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *Network and Distributed System Security Symposium*. 2017. URL: https://api.semanticscholar.org/CorpusID:2590070 (cited on page 59).

[Tai20] Tailscale Inc. *How Tailscale works*. Mar. 20, 2020. URL: https://tailscale.com/blog/how-tailscale-works (visited on 06/04/2024) (cited on page 59).

[DZH03] Zhenhai Duan, Zhi-Li Zhang, and Y.T. Hou. "Service overlay networks: SLAs, QoS, and bandwidth provisioning." In: *IEEE/ACM Transactions on Networking* 11.6 (Dec. 2003), pages 870–883. ISSN: 1558-2566. DOI: 10.1109/TNET.2003.820436 (cited on page 60).

[Kub14] Kubernetes Authors. *Production-Grade Container Orchestration*. 2014. URL: https://kubernetes.io/ (visited on 06/04/2024) (cited on pages 61, 66, 67, 159).

[Bur+22] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: Up & Running*. 3rd edition. O'Reilly Media, 2022. ISBN: 978-1-098-11020-8 (cited on page 61).

[SAF23] Kensworth Subratie, Saumitra Aditya, and Renato J. Figueiredo. "EdgeVPN: Self-organizing layer-2 virtual edge networks." In: *Future Generation Computer Systems* 140 (2023), pages 104–116. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0167739X22003235 (cited on pages 61, 66, 67, 162).

[Bar+23] Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pages 215–231. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/bartolomeo (cited on pages 61, 66).

[Doc13] Docker Inc. *Docker: Accelerated Container Application Development*. Mar. 20, 2013. URL: https://www.docker.com/ (visited on 06/04/2024) (cited on page 61).

[Igo04] Igor Sysoev. *nginx*. Oct. 4, 2004. URL: https://nginx.org/ (visited on 06/04/2024) (cited on page 62).

[Mar09] MariaDB Corporation. *MariaDB Enterprise Open Source Database | MariaDB*. Oct. 29, 2009. URL: https://mariadb.org/ (visited on 06/04/2024) (cited on page 62).

[Yin+09] Qin Yin, Adrian Schüpbach, Justin Cappos, Andrew Baumann, and Timothy Roscoe. "Rhizoma: A Runtime for Self-deploying, Self-managing Overlays." In: *Middleware 2009*. Edited by Jean M. Bacon and Brian F. Cooper. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pages 184–204. ISBN: 978-3-642-10445-9 (cited on pages 63–67).

[Cap+16] Mauro Caporuscio, Vincenzo Grassi, Moreno Marzolla, and Raffaela Mirandola. "GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly." In: *IEEE Transactions on Software Engineering* 42.2 (Feb. 2016), pages 136–152. ISSN: 1939-3520. DOI: 10.1109/TSE.2015.2476797 (cited on pages 64, 66).

[AK09] I. Al-Oqily and A. Karmouch. "Towards automating overlay network management." In: *Journal of Network and Computer Applications* 32.2 (2009), pages 461–473. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2008.02.013. URL: https://www.sciencedirect.com/science/article/pii/S1084804508000593 (cited on pages 64–67).

[Yin+08] Qin Yin, Justin Cappos, Andrew Baumann, and Timothy Roscoe. "Dependable self-hosting distributed systems using constraints." In: *Proceedings of the Fourth Conference on Hot Topics in System Dependability*. HotDep'08. San Diego, California: USENIX Association, 2008, page 11 (cited on page 64).

[Syc+96] K. Sycara, A. Pannu, M. Willamson, Dajun Zeng, and K. Decker. "Distributed intelligent agents." In: *IEEE Expert* 11.6 (Dec. 1996), pages 36–46. ISSN: 2374-9407. DOI: 10.1109/64.546581 (cited on page 64).

[KB06] Madhu Kumar S.D and Umesh Bellur. "An underlay aware, adaptive overlay for event broker networks." In: *Proceedings of the 5th Workshop on Adaptive and Reflective Middleware (ARM '06)*. ARM '06. Melbourne, Australia: Association for Computing Machinery, 2006, page 4. ISBN: 1595934197. DOI: 10.1145/1175855.1175859. URL: https://doi.org/10.1145/1175855.1175859 (cited on page 66).

[Buf08] John F. Buford. "Management of peer-to-peer overlays." In: *Int. J. Internet Protoc. Technol.* 3.1 (July 2008), pages 2–12. ISSN: 1743-8209. DOI: 10.1504/IJIPT.2008.019291. URL: https://doi.org/10.1504/IJIPT.2008.019291 (cited on pages 65–67).

[Net20] NetFoundry Inc. *OpenZiti - Open Source Zero Trust Networking*. 2020. URL: https://openziti.io/ (visited on 06/04/2024) (cited on pages 65–67).

[Hog22] Luc Hogie. "Idawi: a decentralised middleware for achieving the full potential of the IoT, the fog, and other difficult computing environments." In: *Proceedings of the 1st Workshop on Middleware for the Edge*. MIDDLEWEDGE '22. Quebec, Quebec City, Canada: Association for Computing Machinery, 2022, pages 1–5. ISBN: 9781450399302. DOI: 10.1145/3565385.3565876. URL: https://doi.org/10.1145/3565385.3565876 (cited on pages 66, 67).

[Ros+23] Lorenzo Rosa, Andrea Garbugli, Antonio Corradi, and Paolo Bellavista. "INSANE: A Unified Middleware for QoS-aware Network Acceleration in Edge Cloud Computing." In: *Proceedings of the 24th International Middleware Conference*. Middleware '23. Bologna, Italy: Association for Computing Machinery, 2023, pages 57–70. DOI: 10.1145/3590140.3629105. URL: https://doi.org/10.1145/3590140.3629105 (cited on pages 66, 67, 159).

[DAS23] Javier Jose Diaz Rivera, Muhammad Afaq, and Wang-Cheol Song. "Blockchain and Intent-Based Networking: A Novel Approach to Secure and Accurate Network Policy Implementation." In: *2023 24st Asia-Pacific Network Operations and Management Symposium (APNOMS)*. Sept. 2023, pages 77–82 (cited on pages 65–67).

[Ryu17] Ryu SDN Framework Community. *Ryu SDN Framework*. Jan. 1, 2017. URL: https://ryu-sdn.org/ (visited on 09/25/2024) (cited on page 70).

[LL14] Lin Lin and Ping Lin. "Software-Defined Networking (SDN) for Cloud Applications." In: *Cloud Computing: Challenges, Limitations and R&D Solutions*. Edited by Zaigham Mahmood. Cham: Springer International Publishing, 2014, pages 209–233. ISBN: 978-3-319-10530-7. DOI: 10.1007/978-3-319-10530-7_9. URL: https://doi.org/10.1007/978-3-319-10530-7_9 (cited on page 70).

[MBB11] Philip Matthews, Iljitsch van Beijnum, and Marcelo Bagnulo. *Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers*. RFC 6146. Apr. 2011. DOI: 10.17487/RFC6146. URL: https://www.rfc-editor.org/info/rfc6146 (cited on page 72).

[DC98] Dr. Steve E. Deering and Alex Conta. *Generic Packet Tunneling in IPv6 Specification*. RFC 2473. Dec. 1998. DOI: 10.17487/RFC2473. URL: https://www.rfc-editor.org/info/rfc2473 (cited on page 72).

[Mah04] Qusay H. Mahmoud. *Middleware for Communications*. 1st edition. John Wiley & Sons, Inc., 2004. ISBN: 0-470-86206-8 (cited on page 73).

[OMG11] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1 (cited on pages 76, 105).

[Cas+03] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. "Secure routing for structured peer-to-peer overlay networks." In: *SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pages 299–314. ISSN: 0163-5980. DOI: 10.1145/844128.844156. URL: https://doi.org/10.1145/844128.844156 (cited on page 80).

[KHR18] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. July 2018. DOI: 10.17487/RFC8445. URL: https://www.rfc-editor.org/info/rfc8445 (cited on page 81).

[Pet+20] Marc Petit-Huguenin, Gonzalo Salgueiro, Jonathan Rosenberg, Dan Wing, Rohan Mahy, and Philip Matthews. *Session Traversal Utilities for NAT (STUN)*. RFC 8489. Feb. 2020. DOI: 10.17487/RFC8489. URL: https://www.rfc-editor.org/info/rfc8489 (cited on page 81).

[Sch+02] Eve Schooler, Jonathan Rosenberg, Henning Schulzrinne, Alan Johnston, Gonzalo Camarillo, Jon Peterson, Robert Sparks, and Mark J. Handley. *SIP: Session Initiation Protocol*. RFC 3261. July 2002. DOI: 10.17487/RFC3261. URL: https://www.rfc-editor.org/info/rfc3261 (cited on page 81).

[IT21] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000 (cited on page 82).

[RTM22] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. Apr. 2022. DOI: `10.17487/RFC9147`. URL: `https://www.rfc-editor.org/info/rfc9147` (cited on page 82).

[DH22] Whitfield Diffie and Martin E. Hellman. "New Directions in Cryptography." In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 1st edition. New York, NY, USA: Association for Computing Machinery, 2022, pages 365–390. ISBN: 9781450398275. URL: `https://doi.org/10.1145/3549993.3550007` (cited on page 88).

[Wel82] E. Weldon. "An Improved Selective-Repeat ARQ Strategy." In: *IEEE Transactions on Communications* 30.3 (Mar. 1982), pages 480–486. ISSN: 1558-0857. DOI: `10.1109/TCOM.1982.1095497` (cited on page 88).

[VGS05] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays." In: *Journal of Network and Systems Management* 13.2 (June 2005), pages 197–217. ISSN: 1573-7705. DOI: `10.1007/s10922-005-4441-x`. URL: `https://doi.org/10.1007/s10922-005-4441-x` (cited on pages 88, 113).

[Sch+00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. 2nd edition. John Wiley & Sons, Inc., 2000. ISBN: 0-471-60695-2 (cited on page 88).

[Lin01] John Lindow. "Norse Mythology: A Guide to the Gods, Heroes, Rituals, and Beliefs." In: *Oxford University Press* (2001) (cited on page 101).

[Ber+12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "High-speed high-security signatures." In: *Journal of Cryptographic Engineering* 2.2 (Sept. 2012), pages 77–89. ISSN: 2190-8516. DOI: `10.1007/s13389-012-0027-1`. URL: `https://doi.org/10.1007/s13389-012-0027-1` (cited on page 102).

[Rog02] Phillip Rogaway. "Authenticated-encryption with associated-data." In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. Washington, DC, USA: Association for Computing Machinery, 2002, pages 98–107. ISBN: 1581136129.

DOI: 10.1145/586110.586125. URL: https://doi.org/10.1145/586110.586125 (cited on page 103).

[Tho21] Erik Thormarker. *On using the same key pair for Ed25519 and an X25519 based KEM*. Cryptology ePrint Archive, Paper 2021/509. https://eprint.iacr.org/2021/509. 2021. URL: https://eprint.iacr.org/2021/509 (cited on page 103).

[Arc20] Scott Arciszewski. *XChaCha: eXtended-nonce ChaCha and AEAD_-XChaCha20_Poly1305*. Internet-Draft draft-irtf-cfrg-xchacha-03. Work in Progress. Internet Engineering Task Force, Jan. 2020. 18 pages. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/ (cited on page 104).

[CK21] Yann Collet and Murray Kucherawy. *Zstandard Compression and the 'application/zstd' Media Type*. RFC 8878. Feb. 2021. DOI: 10.17487/RFC8878. URL: https://www.rfc-editor.org/info/rfc8878 (cited on page 112).

[IFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. "Lua–an extensible extension language." In: *Softw. Pract. Exper.* 26.6 (June 1996), pages 635–652. ISSN: 0038-0644 (cited on page 117).

[Che76] Peter Pin-Shan Chen. "The entity-relationship model–toward a unified view of data." In: *ACM Trans. Database Syst.* 1.1 (Mar. 1976), pages 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440. URL: https://doi.org/10.1145/320434.320440 (cited on page 117).

[Röb22] Kevin Röbert. "A Secure Context-Aware Middleware for Computation Offloading in Untrustworthy, Open, and Dynamic Edge Environments." Master's thesis. University of Hamburg, Feb. 3, 2022 (cited on page 135).

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. "King: estimating latency between arbitrary internet end hosts." In: *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*. IMW '02. Marseille, France: Association for Computing Machinery, 2002, pages 5–18. ISBN: 158113603X. DOI: 10.1145/637201.637203. URL: https://doi.org/10.1145/637201.637203 (cited on page 147).

[ker15] kernel development community. *Control Group v2*. 2015. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#controllers (visited on 05/24/2024) (cited on page 153).

[Var+16] Blesson Varghese, Nan Wang, Sakil Barbhuiya, Peter Kilpatrick, and Dimitrios S. Nikolopoulos. "Challenges and Opportunities in Edge Computing." In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. Nov. 2016, pages 20–26. DOI: 10.1109/SmartCloud.2016.18 (cited on page 157).

[Cao+20] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. "An Overview on Edge Computing Research." In: *IEEE Access* 8 (2020), pages 85714–85728. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2991734 (cited on page 157).

[Shi+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pages 637–646. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198 (cited on page 157).

[NLB19] Duong Tung Nguyen, Long Bao Le, and Vijay K. Bhargava. "A Market-Based Framework for Multi-Resource Allocation in Fog Computing." In: *IEEE/ACM Transactions on Networking* 27.3 (June 2019), pages 1151–1164. ISSN: 1558-2566. DOI: 10.1109/TNET.2019.2912077 (cited on page 161).

[Mad+20] Arkadiusz Madej, Nan Wang, Nikolaos Athanasopoulos, Rajiv Ranjan, and Blesson Varghese. "Priority-based Fair Scheduling in Edge Computing." In: *2020 IEEE 4th International Conference on Fog and Edge Computing (ICFEC)*. May 2020, pages 39–48. DOI: 10.1109/ICFEC50348.2020.00012 (cited on page 161).

[CNI24] CNI Authors. *Container Network Interface*. July 22, 2024. URL: https://github.com/containernetworking/cni (visited on 09/25/2024) (cited on page 162).

[KBE23] Philipp Kisters, Heiko Bornholdt, and Janick Edinger. "SkABNet: A Data Structure for Efficient Discovery of Streaming Data for IoT." In: *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*. 32nd International Conference on Computer Communications and Networks (ICCCN 2023). Honolulu, HI, USA: IEEE, 2023. DOI: 10.1109/ICCCN58024.2023.10230169.

[Bor+21] Heiko Bornholdt, David Jost, Philipp Kisters, Michel Rottleuthner, Sehrish Shafeeq, Winfried Lamersdorf, Thomas C. Schmidt, and Mathias Fischer. "Smart Urban Data Space for Citizen Science." In: *Volume 80: Conference on Networked Systems 2021 (NetSys*

2021). Conference on Networked Systems (NetSys 2021). Lübeck, Germany: ECEASST, 2021. DOI: 10.14279/tuj.eceasst.80.1158.

[PKB21] Wolf Posdorfer, Julian Kalinowski, and Heiko Bornholdt. "Towards EU-GDPR Compliant Blockchains with Intentional Forking." In: *Advances in Computer, Communication and Computational Sciences, Proceedings of IC4S 2019*. International Conference on Computer, Communication and Computational Sciences (IC4S 2019). Bangkok, Thailand: Springer Singapore, 2021, pages 649–658. DOI: 10.1007/978-981-15-4409-5_58.

[PBL20] Wolf Posdorfer, Heiko Bornholdt, and Winfried Lamersdorf. "Transaction Dependency Model for Block Minimization in Arbitrary Blockchains." In: *Proceedings of the 2nd International Electronics Communication Conference*. 2nd International Electronics Communication Conference (IECC 2020). Singapore, Singapore: ACM, 2020, pages 59–66. DOI: 10.1145/3409934.3409935.

[Hau+18] Christopher Haubeck, Heiko Bornholdt, Winfried Lamersdorf, Abhishek Chakraborty, and Alexander Fay. "Step-based Evolution Support among Networked Production Automation Systems." In: *at - Automatisierungstechnik* (2018), pages 849–858. DOI: 10.1515/auto-2018-0047.

[Bor+19] Heiko Bornholdt, David Jost, Philipp Kisters, Michel Rottleuthner, Dirk Bade, Winfried Lamersdorf, Thomas C. Schmidt, and Mathias Fischer. "SANE: Smart Networks for Urban Citizen Participation." In: *2019 26th International Conference on Telecommunications (ICT)*. 26th International Conference on Telecommunications (ICT 2019). Hanoi, Vietnman: IEEE, 2019, pages 496–500. DOI: 10.1109/ICT.2019.8798771.

[Pos+20] Wolf Posdorfer, Julian Kalinowski, Heiko Bornholdt, and Winfried Lamersdorf. "Decentralized Billing and Subcontracting of Application Services for Cloud Environment Providers." In: *Advances in Service-Oriented and Cloud Computing, International Workshops of ESOCC 2018*. 8th European Conference On Service-Oriented And Cloud Computing (ESOCC 2018). Como, Italy: Springer Cham, 2020, pages 91–101. DOI: 10.1007/978-3-030-63161-1_7.

[Kis24] Liliana Kistenmacher. "QUIC Load Balancing: Enumeration Attacks on QUIC-aware Load Balancers and Countermeasures." Master's thesis. University of Hamburg, Feb. 22, 2024.

[Sem23] Anton Semjonov. "Opportunistic Distributed Computation Offloading using WebAssembly." Master's thesis. University of Hamburg, Oct. 16, 2023.

[Brü22] Timon Brüning. "Methoden zur Qualitätsanalyse von Daten in Sensornetzwerken im Kontext einer Smart City." Master's thesis. University of Hamburg, Feb. 18, 2022.

[Mai21] Tom Maier. "Dezentrale Access-Control-Konzepte für einen kontrollierten Austausch von Daten zwischen Bürger:innen auf Smart-City-Marktplätzen." Master's thesis. University of Hamburg, Apr. 14, 2021.

[Sza21] Phil Szalay. "Methoden für eine Event-basierte Verarbeitung von Sensordaten aus heterogenen Datenquellen im Smart-Home-Bereich am Beispiel eines Incorum-Ansatzes." Master's thesis. University of Hamburg, June 29, 2021.

[Abt19] Florian Abt. "Monitoring-Dienste im Web of Things." Master's thesis. University of Hamburg, Dec. 1, 2019.

[Bag19] Samuel Bagdassarian. "Predicting Suitable Update Times for IoT Devices." Master's thesis. University of Hamburg, Dec. 27, 2019.

[Rei19] Yanneck Reiß. "Caching-Strategien für Daten in Wireless-Sensornetzwerken im Kontext einer Smart City." Master's thesis. University of Hamburg, Mar. 5, 2019.

[Vie18] Sebastiano Vierk. "Software-Wartung im Kontext von Firmware-Updates bei IoT-Geräten." Master's thesis. University of Hamburg, Dec. 14, 2018.

[Moh22] Patrick Mohr. "Evaluating Impacts on the Randomness Properties of Gossip-Based Peer Sampling when using the drasyl Communication Framework." Bachelor's thesis. University of Hamburg, May 23, 2022.

[Bau21] Simon Bauer. "Ursachen, Auswirkungen und Behandlung von Free Riding in Smart-City-Datenräumen." Bachelor's thesis. University of Hamburg, June 7, 2021.

[Hir19] Michael Hirsch. "Evaluierung von Question-Answering Systemen für Sensornetze." Bachelor's thesis. University of Hamburg, Nov. 28, 2019.

[Ngu19] Minh Hieu Nguyen. "Flexible und automatisierte Provisionierung von IoT-Systemen am Beispiel von Raspberry-Pi-basierten Robotern." Bachelor's thesis. University of Hamburg, May 1, 2019.

[Wut19] Maximilian Wutz. "Entwurf einer Middleware zur Standardisierung von heterogenen Sensordaten am Beispiel von openHAB." Bachelor's thesis. University of Hamburg, Mar. 27, 2019.

[Sch23] Tom Schmolzi. "A Proposal-Based, Decentralized Demand-Side Management Peak-Reduction Algorithm." base.camp project report. University of Hamburg, Mar. 2023.

[Röb21] Kevin Röbert. "Overlay Network Framework for Rapid Development of Distributed P2P Applications." base.camp project report. University of Hamburg, Mar. 2021.

[Töt21] Fin Töter. "Towards a Decentralized Trust-Based Reputation System for Citizen-Driven Marketplaces." base.camp project report. University of Hamburg, Dec. 2021.

[Alt20] Claas Altschaffel. "Virtuelle Assistenten als Benutzungsschnittstelle für Smart-City-Plattformen." Diploma thesis. University of Hamburg, Mar. 23, 2020.

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Sofern im Zuge der Erstellung der vorliegenden Dissertationsschrift generative Künstliche Intelligenz (gKI) basierte elektronische Hilfsmittel verwendet wurden, versichere ich, dass meine eigene Leistung im Vordergrund stand und dass eine vollständige Dokumentation aller verwendeten Hilfsmittel gemäß der Guten wissenschaftlichen Praxis vorliegt. Ich trage die Verantwortung für eventuell durch die gKI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.

Pinneberg, September 30, 2024

Heiko Tobias Bornholdt

COLOPHON

This document was typeset using LaTeX and the typographical look-and-feel `classicthesis`[1] developed by André Miede and Ivo Pletikosi. Furthermore, the style customization in this document was inspired by Aaron Turon's *Understanding and expressing scalable concurrency*[2].

1 https://bitbucket.org/amiede/classicthesis/

2 https://www.ccs.neu.edu/home/turon/thesis.pdf