Computer Communications 242 (2025) 108294

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Simplifying distributed application deployment at the edge through software-defined overlay networks

Heiko Bornholdt [a,*], Kevin Röbert [a], Stefan Schulte [b], Janick Edinger [a], Mathias Fischer [a]

[a] University of Hamburg, Department of Informatics, Vogt-Kölln-Straße 30, 22527, Hamburg, Germany
[b] Hamburg University of Technology, Institute for Data Engineering, Blohmstraße 15, 21079, Hamburg, Germany

ARTICLE INFO

Keywords:
Edge computing
Software-defined networking
Overlay networking

ABSTRACT

The need for low latency, bandwidth efficiency, and privacy has driven the deployment of distributed applications to the network edge. However, edge environments introduce concrete challenges such as limited infrastructure control, constrained connectivity due to NAT or firewalls, and the heterogeneity of devices and network conditions. This paper introduces a software-defined overlay networking (SDON) middleware that addresses these issues by simplifying the development and deployment of edge applications through centralized control and dynamic overlay management. SDON allows applications to define high-level requirements, such as node and link characteristics and the network topology. These requirements are translated into device-specific configurations and enforced across suitable edge devices. We implemented our SDON middleware as a fully functional software and evaluated it in two edge computing use cases: i) routing for video streaming across middleboxed edge devices and ii) computation offloading on heterogeneous edge devices. Our results show that deployments via SDON, with centrally enforced optimizations, improve application performance by reducing mean streaming latency by 20 % and computation times by 22 %.

1. Introduction

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 such as CPU cycles, memory, and GPU acceleration. Although cloud infrastructures offer virtually unlimited and standardized resources within a common environment, their centralized nature makes them unsuitable for many deployment scenarios—necessitating a shift towards edge computing. Edge applications include widely distributed services in heterogeneous, uncontrolled, and untrusted environments managed by separate administrative authorities. Such environments include home, business, mobile, or public networks and smartphone, tablet, laptop, desktop, or server-class devices. Deploying applications in these settings introduces distinct challenges: limited connectivity across network boundaries, lack of trust in the infrastructure, and the absence of centralized control make it difficult to configure and manage edge deployments efficiently. These networks and devices must collaborate seamlessly to ensure the proper functioning of applications [1]. Therefore, edge applications require mechanisms that enforce quality of service (QoS) goals regarding network and device resources to operate efficiently in an otherwise best-effort system. However, implementing such mechanisms is complex, time-consuming, and error-prone. As a result, developers often employ simpler mechanisms, resulting in less efficient edge resource utilization of their applications [2].

The absence of central control, edge heterogeneity, and limited network configuration capabilities pose significant challenges to edge application deployment [3]. These challenges are not adequately addressed by traditional management solutions, such as Kubernetes and software-defined networking (SDN), which are designed with strong assumptions about control, reliability, and reachability over the underlying infrastructure. However, other approaches that aim to support edge deployments impose restrictions on the application design by requiring the application to be developed using a specific paradigm or framework [4–8].

One promising approach to address these challenges is the use of overlay networks. An overlay network introduces a logical layer between the network and application layers, creating virtual links and topologies independent of the physical infrastructure. This enables the application to interact with the network in a controlled and idealized manner, without requiring changes to the underlay. Overlay networks can help enforce QoS and improve resource usage by decoupling application-level communication from physical constraints. However,

* Corresponding author.
E-mail address: heiko.bornholdt@uni-hamburg.de (H. Bornholdt).

https://doi.org/10.1016/j.comcom.2025.108294
Received 24 February 2025; Received in revised form 25 June 2025; Accepted 1 August 2025
Available online 28 August 2025

building and maintaining overlay networks in highly dynamic and heterogeneous edge environments remain complex tasks, often requiring substantial effort and customization from the developer [1].

To address this complexity, we propose a communication middleware that is deployed on edge devices running distributed applications. This middleware automatically constructs and maintains an IP overlay network that realizes the applications' communication and placement requirements. It combines advanced network management principles, such as centralized control, intent-based configuration, and reactive adaptation—with the flexibility and infrastructure, independence of overlay networks. As a result, distributed applications can operate over a virtualized, idealized network view, without requiring changes to the underlying infrastructure or to the applications themselves. In our previous poster paper, we introduced the concept of such a middleware [9], but that work remained conceptual and did not include a concrete implementation or evaluation.

The main contribution of this paper is a fully implemented and evaluated SDON middleware that simplifies the development, deployment, and operation of distributed applications in edge environments. Beyond combining SDN principles with overlay flexibility, our system introduces a novel runtime model that enables dynamic, intent-driven adaptations and supports seamless operation across heterogeneous and unmanaged edge devices. The goal is to relieve developers from low-level configuration and coordination tasks, enabling faster prototyping and robust execution of edge applications without requiring modifications to the applications themselves.

The SDON middleware achieves this by:

- providing a high-level API to specify communication and placement requirements, such as network topology, link properties, and service assignments,
- automatically constructing and maintaining an IP overlay network that fulfills these requirements on heterogeneous edge devices,
- continuously monitoring runtime conditions and adapting the overlay through a closed-loop control mechanism, and
- supporting unmodified IP-based applications by transparently embedding overlay functionality on the device level.

We evaluated our SDON concept in two complementary use cases. The first use case involves a video streaming application with SDON-managed routing that reduces streaming latency under middlebox-imposed communication restrictions. The second use case involves computation offloading, where SDON-managed service placement minimizes task completion times through optimized node arrangements. Our SDON middleware is open-source available at https://github.com/drasyl/drasyl/.

The paper is structured as follows: Section 2 identifies the requirements for a SDON system and evaluates related work. Section 3 presents our SDON middleware architecture. Section 4 provides insights into our SDON middleware implementation. Section 5 outlines the two use cases used in the evaluation, and Section 6 presents the evaluation results. Section 7 discusses the findings, limitations, and potential extensions of our work. Section 8 concludes the paper.

2. Requirements analysis and related work

This section derives the requirements for our proposed SDON system and classifies related work accordingly.

2.1. Requirements analysis

Our scenario-based requirements analysis (Fig. 1) outlines the stakeholders involved in a SDON-based edge deployment and describes their roles, interactions, and resulting system requirements [10]. A *distributed application* is standalone software composed of communicating components deployed across edge devices. While the application is executed like any other local program, it transparently uses a virtual overlay network managed by the SDON system to realize its communication and placement requirements. The *SDON system* runs alongside the application on each participating edge device and provides overlay functionality. A centralized controller component interprets a declarative overlay specification and orchestrates device behavior, including service placement and virtual link configuration. Although execution is distributed, the control logic is centralized. An *application developer* integrates the SDON middleware into the application and expresses overlay-related requirements using a high-level API. The developer expects the SDON system to handle tasks such as routing, resource selection, and service placement transparently. An *application user* interacts with the application as with any conventional software, expecting seamless functionality. *Edge devices* provide local resources—such as CPU, memory, and network connectivity—to the overlay. Devices can either be statically known and prepared for use or dynamically discovered and integrated, especially in peer-to-peer contexts.

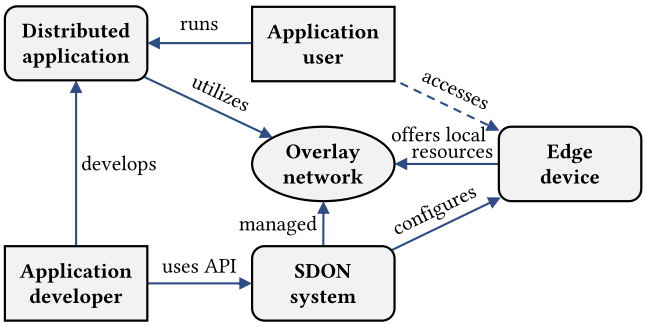


Fig. 1. Stakeholders and interactions in a SDON system.

The identified roles and their interactions expose various technical and non-technical requirements that the SDON system must fulfill. These are discussed in the following section.

2.1.1. Functional requirements

Functional requirements describe *what* the system must do. They define the externally visible behavior and services that the SDON system shall provide to fulfill its purpose [10].

Dynamic overlay programming (R1): Provides the fundamental capability to program overlay networks that reflect the application's communication and placement requirements. Since both application goals and available resources may change at runtime, the system must support dynamic reconfiguration of the overlay.

Self-configuration (R2): Instead, the system infers and applies the necessary configuration from high-level goals defined in a declarative model. These goals include topology preferences (e.g., mesh, star), link constraints (e.g., latency, bandwidth), and node constraints running application services (e.g., CPU, memory). The SDON system interprets this model and ensures consistent behavior across all devices by autonomously deriving and distributing the overlay configuration.

QoS support (R3): Supports service guarantees by enforcing QoS constraints such as bandwidth, delay, or path redundancy. These constraints must be expressible in a high-level and technology-agnostic form, allowing developers to specify quality expectations for individual links or overlay paths. To ensure consistent performance, the system must continuously monitor relevant metrics and adapt the overlay when constraints are no longer met.

2.1.2. Non-functional requirements

Non-functional requirements specify *how* the system should perform its functions. They characterize quality attributes such as performance, robustness, or scalability, and describe constraints and expectations regarding the system's behavior under varying conditions [10].

Efficiency (R4): Ensures that overlay management does not cause excessive resource overhead. The system should optimize resource usage across CPU, memory, bandwidth, and energy, both locally per device and globally.

Robustness (R5): Maintains reliable operation even in the presence of fluctuating resource availability, partial failures, or invalid inputs. Mechanisms for fault tolerance, self-healing, and fallback must be provided.

Scalability (R6): Allows operation across small to large deployments, supporting horizontal scaling by incorporating additional edge resources. Central bottlenecks and single points of failure should be avoided.

Extensibility (R7): Enables the integration of new components or overlay services without disrupting existing deployments. The architecture must support pluggable and replaceable subsystems.

Security (R8): Ensures confidentiality, integrity, and availability of all overlay communications. This includes support for end-to-end encryption, authentication, and resilience to denial-of-service and eavesdropping attacks.

2.2. Related work

This section reviews existing approaches to overlay network construction and management, especially in the context of distributed applications and edge environments. The evaluation is structured along key requirements identified in Section 2.1, and the classification is summarized in Table 1. The entries in the table are sorted chronologically by publication year.

Table 1
Related work in overlay networking systems [11–17].

System / *Author*	Year	R1	R2	R3	R4	R5	R6	R7	R8
i3 [11]	2002		●		●	●	●	●	●
MACEDON [5]	2004	◐		◐	◐	?	●	●	?
UIP [18]	2004		●		●	●	●		●
Behnel et al. [12]	2005	◐	●	●	●	◐	◐		
Kumar et al. [19]	2006		●	●		●	●		?
Buford [20]	2008	●	●			●	◐		
Overlay Weaver [4]	2008	◐		◐	◐	?	●	●	?
Akka [8]	2009			◐		●	●	●	●
Al-Oqily et al. [21]	2009	●	●	●	◐	◐			◐
Rhizoma [22]	2009	●	●	◐		●	●		?
SOLID [23]	2010		●			●	●		●
JXTA [13]	2011	●			◐	●	●	●	●
Jadex [6]	2012	◐	●		●		◐	●	◐
Kubernetes [24]	2014	◐	◐	●	◐	●	●	●	◐
ZeroTier [25]	2014		●		●		◐		●
GoPrime [26]	2016		●	●		●	●		?
ActorEdge [14]	2017		◐				●		
EmbJXTAChord [7]	2018	●	◐		●	●	●	●	●
Tailscale [27]	2019		●		●		◐		●
libp2p [28]	2020				●	●	●	●	●
OpenZiti [29]	2020	◐	◐		●	●	●	●	●
Parallel Theater [15]	2021		◐				●	●	?
Idawi [30]	2022				●	●	●	●	?
EdgeVPN [31]	2023		●	●	●	●	●	●	●
INSANE [32]	2023		●	●		?	?	●	?
`Oakestra` [16]	2023	◐	●	●	◐	●	●	●	?
Diaz Rivera et al. [33]	2023	◐	●		●	●	●	●	●
PolyNet [17]	2024	◐	◐	●	●	●	●	◐	?
Our system		●	●	●	●	◐	◐	●	●

● met ◐ partially met (blank) not met ? unknown.

Previous approaches aim to separate the application from the overlay network implementation by providing unified interfaces for interacting with the overlay network [4,5,20,29,33]. These approaches allow easy switching between overlay network implementations, but only at build time. This means the overlay network cannot be changed dynamically during runtime, preventing it from adapting to changing demands while the application runs. Only a few conceptual works, such as [20–22], consider runtime reconfiguration of overlay schemes. However, these primarily define abstract policy frameworks rather than fully implemented systems and remain limited to specific overlay types.

Several approaches provide partial support for self-configuration, allowing policies to be defined and applied within the overlay system [6,19,22,25–27,32]. However, these methods impose strict constraints on the overlay structure, often making dynamic overlay programming and self-configuration mutually exclusive. Moreover, self-configuration is often tied to specific overlay schemes (e.g., DHTs), which limits adaptability. Systems like INSANE [32] or Jadex [6] offer automation within narrowly scoped scenarios, such as peer-to-peer links or service discovery, but cannot be generalized to broader overlay configurations.

Similarly, QoS support is often either absent or only partially implemented to single aspects [20,22,29,33]. Most approaches focus on single performance aspects and lack holistic support for multi-dimensional QoS optimization. Existing approaches frequently make strong assumptions about underlay control and availability conditions that are generally only met in cloud environments. Consequently, operation in restricted edge environments is often considered out of scope. While some solutions [6,7,18,25,27,30,31] effectively address edge environment challenges and bypass middlebox barriers, many advanced systems, such as Kubernetes [24], are designed with cloud-based infrastructures in mind. Approaches like EdgeVPN [31] or ZeroTier [25] provide mechanisms such as NAT traversal or failover support, yet lack control interfaces for programmable QoS behavior.

Robust operation in heterogeneous and dynamic edge environments remains limited. Many systems rely on stable underlay conditions and high resource availability, assumptions not met in practice outside data centers [24]. Some frameworks like [7,28] offer low-level primitives for efficient overlay construction but leave performance guarantees to the user. Only a few, like [25,27], address robustness explicitly through encryption or adaptive routing.

While most systems claim scalability through decentralized designs, they often sacrifice adaptability or QoS enforcement. Central coordination mechanisms necessary for dynamic reconfiguration are missing or rudimentary, limiting practical extensibility.

Security aspects such as confidentiality, integrity, and access control are frequently neglected or limited to encryption. Only few systems such as ZeroTier [25] or Tailscale [27] implement systematic data-plane encryption and peer authentication, but even these do not support customizable security policies or fine-grained trust zones across overlays.

Additionally, many approaches provide OSI layer 7 API overlays, requiring specific application adaptation. Only [23–25,27,31] create overlays usable by IP-based applications.

Summarizing prior work, a recurring observation is the frequent absence of *dynamic* overlay network reconfiguration in response to changing application requirements or resource constraints. Further, there is often insufficient QoS support, which is crucial for effective resource management in edge environments. Additionally, existing applications frequently require application modifications to be compatible with a given approach, or limitations in the application design process may arise. To the best of our knowledge, no existing system addresses all outlined requirements jointly. In particular, there is a lack of middleware that provides dynamic reconfiguration, self-configuration, QoS-awareness, and extensibility across heterogeneous edge networks

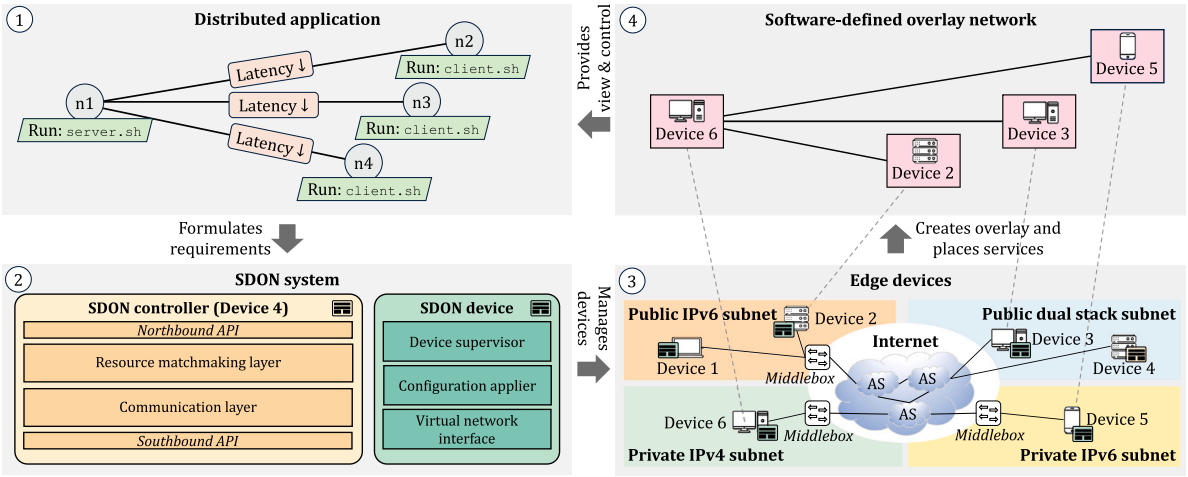


Fig. 2. SDON system overview. ① A distributed application *formulates requirements* that express its communication and service goals. ② The SDON system, consisting of a centralized controller running on device 4, interprets these requirements and selects suitable *managed devices*. ③ These devices are configured to *create the overlay and place application services*. ④ The resulting overlay network *provides an idealized view and control* of edge device resources to the application.

without requiring modifications to the application layer. This motivates our SDON-based approach.

3. SDON System Architecture

This section describes the architecture of our SDON system, tailored to support the deployment of distributed applications in edge environments as discussed in this paper. First, a system overview is given, followed by details on the SDON controller that orchestrates SDON-enabled devices to construct overlay networks for distributed applications running at the network edge.

The implementation of the SDON system builds upon the open-source *drasyl* framework, which provides reusable networking primitives, such as secure messaging, NAT traversal, and peer discovery, but does not prescribe specific application-layer behavior. Within this project, we introduced a novel middleware layer atop drasyl by implementing a software-defined overlay mechanism. Specifically, we developed a centralized SDON controller, a declarative network model for specifying overlay requirements, and runtime reconfiguration logic to support dynamic overlay construction and adaptation. These components go significantly beyond the core capabilities of drasyl and were designed specifically for this work to enable programmable overlay networking in heterogeneous edge environments.

Fig. 2 depicts the SDON system. At the top left, a distributed application formulates high-level requirements for an overlay network essential for its operation. These include node-level requirements (e.g., which services to host) and link-level constraints (e.g., maximum latency or minimum bandwidth), expressed as declarative specifications within the application's configuration. At the bottom left, a SDON controller on device 4 manages all other SDON-enabled devices (bottom right). The controller receives the overlay network requirements and then automatically determines the necessary low-level configuration steps and suitable edge devices to build the required overlay network (top right).

The SDON middleware operates at the application layer, which allows it to function without requiring any modifications to the underlying system stack or infrastructure. However, it exposes Layer 3 functionality to edge applications by constructing a virtual IP-based overlay network atop the existing underlay. This enables transparent communication between distributed components, as IP packets are intercepted, processed, and forwarded according to the overlay configuration. Physical and virtual network interfaces of the underlay remain unchanged.

3.1. SDON Controller

The role of the SDON controller is similar to the one of a SDN controller by providing central control and a common API for implementing the desired network behavior. However, the restrictions of edge networks require a different approach to achieve central control. A SDN controller directly configures network devices, such as SDN-enabled switches to enforce control mechanisms. In contrast, a SDON controller only controls the SDON-enabled end devices and not the networks in between. Therefore, it must implement network control mechanisms directly on the end devices. This is achieved by creating an overlay network that connects end devices without requiring reconfiguring the existing edge network infrastructure. Unlike traditional SDN controllers such as ONOS [34] and OpenDaylight [35], which control infrastructure devices (e.g., switches) in managed networks, our SDON controller operates at the network edge. It manages end devices directly and must therefore implement network control functionality without relying on configurable intermediate infrastructure. While the end device must be SDON-enabled, the application placed on it does not need to implement any network control functionality. Application services can communicate with remote services using IP. In the SDON context, we distinguish between nodes and devices: A device represents a physical execution platform (e.g., a workstation, edge node, or embedded host) that participates in the overlay and runs the SDON middleware. A node, in contrast, is a logical element defined in the network model. It specifies the roles and service capabilities required from a hosting device (e.g., run a video relay or perform task execution), which guide the placement of application services. The SDON controller's task is to map logical nodes to suitable physical devices. In addition to configuring overlay links, the SDON controller also orchestrates the deployment of application services. This includes instructing selected devices to launch specific services associated with their assigned overlay roles. Hence, the controller not only manages network-level behavior but also steers application-level execution, making it responsible for the overall composition and operation of distributed edge applications. To support these roles, the controller is structured into four sub-components, each responsible for a specific aspect of overlay coordination and execution (see Section 4.1 for details).

The overall architecture and implementation of the SDON controller, including its APIs, protocols, and internal modules, were developed specifically as part of this work. The design follows SDN principles, adapted to the constraints and requirements of edge environments as defined in Section 2.1.1. In analogy to SDN controllers such as ONOS and OpenDaylight, our controller follows a layered

architecture consisting of a northbound APIs, core logic components, and a southbound APIs. This separation of concerns enables modularity and extensibility.

Moreover, communication between controller and edge devices builds on a secure P2P protocol introduced in our prior work [36]. This protocol uses TLS 1.3 and hole punching techniques to ensure that SDON devices, even when behind NATs or firewalls, can establish secure and direct paths to each other. This is crucial for enabling flexible overlay topologies and keeping protocol overhead minimal, thus supporting efficient runtime adaptation.

3.1.1. Communication layer

Flexible overlay network creation requires reachability between the SDON controller and all SDON devices, as well as among the SDON devices. However, edge devices are often distributed across multiple subnets, each applying different configurations and managed by separate administrative authorities. Here, the only common denominator is that all devices are *somehow* connected via the Internet. But many of these devices are operated behind middleboxes, such as NATs and firewalls, rendering them unreachable from outside their subnet. Analysis of popular P2P applications has shown that up to 92% of devices in these applications are affected by middleboxes [37]. Therefore, the communication layer applies techniques for peer discovery and routing known from the P2P domain, namely NAT traversal and public-key based routing. NAT traversal renders middleboxed devices reachable, using the most efficient network path available in many cases. Public-key-based routing equips every device with a public–private key pair, where the public part is used for addressing and the private part for authentication.

3.1.2. Resource matchmaking layer

While the communication layer achieves secure any-to-any reachability between SDON devices, the resource matchmaking layer maps overlay network elements to the most suitable SDON devices. The matchmaker operates opportunistically based on the available knowledge at the time of matchmaking. As edge resources are dynamic, this would cause the overlay network state deviate increasingly from the desired state over time. To address this, a closed-loop mechanism monitors the overlay network's desired state and current overlay network deployed state and periodically reruns the matchmaking. Each SDON device continuously collects runtime metrics such as link latency, processing delay, and compliance with the assigned overlay role, and reports them to the controller. The controller compares this current state with the desired state defined in the application-specific network model. Developers may extend the insight mechanism with custom, application-level metrics to enable domain-specific overlay adaptations.

The matchmaking process is outlined in Algorithm 1. Matchmaking requires traversing all overlay nodes N (line 3). Each node's requirements, including those of its connected links, 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 s_d of n is compared with the current state s_c of each candidate device d to determine the best match (lines 7 to 11). Each state is represented as a resource vector $s = (p_1, p_2, \ldots, p_k)$, where each component p_i denotes a resource property (e.g., CPU usage, latency). The distance between s_d and s_c is computed by subtracting the vectors component-wise and applying a priority vector $w = (w_1, w_2, \ldots, w_k)$:

$$\begin{aligned}\Delta s_{\text{weighted}} &= w \circ |s_d - s_c| \\ &= \left(w_1 \cdot |s_{d1} - s_{c1}|,\ \ldots,\ w_k \cdot |s_{dk} - s_{ck}|\right)\end{aligned}$$

The overall distance is the sum of all weighted differences:

$$\text{distance}(s_d, s_c) = \sum_{i=1}^{k} w_i \cdot |s_{di} - s_{ci}|$$

This allows certain properties (e.g., low latency) to influence the selection more strongly than others. For example, if $s_d = (0.5, 50)$, $s_c = (0.4, 60)$, and $w = (2, 1)$, then the weighted difference is $(0.2, 10)$ and the total distance is 10.2. The device with the smallest such distance ($min\_distance$) is selected for the mapping. 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). Following this matchmaking process, a configuration is generated and sent to each edge device to which an overlay node has been mapped. This configuration specifies a set of predefined routines for the device to apply (e.g., establishing a connection with other devices to establish an overlay link, custom routes, or running a script that starts an application service).

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
```

3.1.3. Northbound API

This component provides the interface that allows applications to model the desired overlay network. This model includes information such as the number of desired nodes, the application services the nodes should run, nodes' overlay IP addresses, and the required links between them.

3.1.4. Southbound API

Used for communication with SDON devices to disseminate instructions for overlay network construction and periodically collect insights into local device status, configuration compliance, and running application services. These insights are used in the closed-loop mechanism.

3.2. SDON Device

Regular edge devices are transformed into SDON-enabled devices, which then register with a SDON controller. They await configuration from the SDON controller to construct an overlay network with other SDON devices and run application services placed on them. The SDON device includes three sub-components:

3.2.1. Device supervisor

Supervises the SDON middleware running on the edge device. Maintaining the connection to the SDON controller, collecting local device information sent to the controller, and coordinating the other sub-components.

3.2.2. Configuration applier

Applies the configuration directives received from the controller. Monitors the implementation progress of directives and sends this information to the supervisor. The applier continues until a configuration is successfully applied.

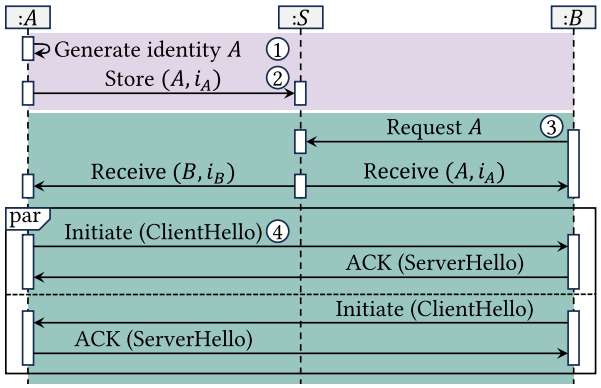


Fig. 3. Communication layer providing secure routability between SDON devices. The upper block shows device A's identity generation and reachability information storing at rendezvous server S. The lower block shows hole punching and key agreement between devices A and B.

3.2.3. Virtual network interface

Provides a network interface on the SDON device, allowing locally running application services to coordinate with remote services using IP communication.

4. SDON Middleware Implementation

We implemented the SDON system as a fully functional middleware. Our Java-based implementation is compatible with Linux-, macOS-, Android-, and Windows-based systems and can construct overlay networks across these platforms. We evaluated our approach with two complementary edge computing use cases to showcase the advantages of simplified edge application deployment. This section provides insights into our SDON middleware by detailing the implementation of the protocol used to achieve secure connectivity among all SDON devices, our API allows applications to describe the desired overlay network, and dissemination and activation of configurations to construct overlay networks.

4.1. SDON Controller

The SDON controller can be placed on any system supported by the middleware and does not impose requirements to be operated on a public Internet host. Once started, the controller runs a service that awaits registration from SDON devices offering their local resources and applications specifying their overlay network requirements.

4.1.1. Communication layer

The communication layer, providing secure reachability among SDON devices and the controller, is implemented as follows (Fig. 3): At initial startup, each SDON device generates an Ed25519 key pair locally, where the 32-byte public part is used as the device identifier A ①. Device A then registers to a rendezvous server S, where reachability information i_A (like the device's IP address) is stored ②. When device B then initiates communication with A (e.g., to establish an overlay link), it requests reachability information for A from S ③. S then sends both devices each other's reachability information. A and B then attempt to connect with each other in parallel using the received information ④. These reachability tests will modify the state of any intervening middleboxes to pass through each other's inbound communication. Further, these tests are used to authenticate both devices and agree on a key used for secure communication using a Diffie–Hellman key exchange involving a `ClientHello` and `ServerHello` message.

Our communication layer is based on an existing protocol that uses hole punching communication to piggyback TLS handshakes [36]. This protocol minimizes the delay before SDON devices can securely communicate. Fast connectivity helps efficiently establish and maintain overlay network topology in dynamic environments.

4.1.2. Northbound API

A Lua-based API allows applications to "programm" the desired overlay network through Lua's scripting capabilities describing the desired network, nodes, and links, as well as their desired properties. Further, the model can be defined to adapt dynamically to changes in edge resources. For this, a callback function is specified, which is called by the SDON controller every time resources change. This function can execute regular Lua scripts to adjust the SDON device mapping or network model. The ability to script overlay networks enables the detailed description of various overlay configurations and centrally enforced optimizations.

Listing 1 SDON network model example.

```
1  net = create_network()
2  net:add_node('n1', {ip = '10.2.1.1', run =
     ↪ 'server.sh'})
3  for i = 2, 4 do
4   net:add_node('n'..i, {ip = '10.2.1.'..i, run =
      ↪ 'client.sh'})
5   net:add_link('n1', 'n'..i)
6  end
7  net:set_callback(function(net, devices)
8   best_combination, min_total_latency = nil, math.huge
9   for i = 1, #devices - 3 do
10   for j = i + 1, #devices - 2 do
11    for k = j + 1, #devices - 1 do
12     for l = k + 1, #devices do
13      d1, d2, d3, d4 = devices[i], devices[j],
          ↪ devices[k], devices[l]
14      total_latency = insights(d1, d2, 'latency') +
          ↪ insights(d1, d3, 'latency') + insights(d1,
          ↪ d4, 'latency')
15      if total_latency < min_total_latency then
16       min_total_latency = total_latency
17       best_combination = {d1, d2, d3, d4}
18      end
19     end
20    end
21   end
22  end
23  for i = 1, 4 do
24   net:node('n'..i).device = best_combination[i]
25  end
26 end)
```

Listing 1 includes an example, modeling the overlay network shown in Fig. 2. Line 1 creates the network model, line 2 creates node `n1` with desired overlay IP address `10.2.1.1` and application service `server.sh`. Lines 3 to 6 add clients `n2`–`n4` all linked to `n1` and running service `client.sh`. Line 7 defines a callback method that searches four devices with the lowest latency (Lines 8 to 22). Finally, lines 23 to 25 maps the overlay nodes to the best-suited devices. This model includes all the information our SDON middleware needs to keep the application running on the best devices at all times. Further, the constructed overlay network transparently hides underlying network restrictions.

4.1.3. Resource matchmaking layer and southbound API

The resource matchmaker traverses the network model and checks which nodes are mapped to which SDON device. For each SDON device, configuration directives are created based on the properties of the corresponding overlay node. For each device, the actual configuration instructions are compared with the previously applied instructions, if

available. Only the changed configurations are distributed to the SDON devices using the Southbound API. The matchmaker also collects the insights received from all SDON devices and calls the network model callback function any time new insights have become available. After executing the callback, the model is checked for changes, resulting in new configurations distributed to the SDON devices.

4.2. SDON Device

The SDON device can be placed on any platform the middleware supports. It either operates as a dedicated system, which awaits the execution of any application managed by the SDON controller, or it is embedded into a specific application. In the latter case, upon starting the application, the SDON controller is contacted in the background, allowing the application to join the existing overlay.

4.2.1. Device supervisor and configuration applier

Upon being selected to run an application service, the SDON device receives configuration instructions from the SDON controller, compares them with existing configurations and applies the necessary updates locally. The applier monitors the implementation progress of each directive and reports this information periodically to the SDON controller. The applier will never cease until successfully (re-)applying this configuration.

4.2.2. Virtual network interface

The network interface enabling the IP application to communicate through the overlay network transparently is provided by creating a TUN device on the device. The kernel of Linux/macOS provides built-in support for this, while Wintun[1] is required for Windows devices. The TUN device is configured with the IP address and netmask specified in the network model. The SDON middleware processes all IP packets sent to this interface, according to the overlay network configuration, which then transmits them to the designated destination SDON device. Similarly, the local TUN interface delivers IP packets received through the overlay network to the locally running application.

5. Edge computing use cases for evaluation

To showcase how our SDON middleware simplifies developing and deploying distributed applications running at the network edge, we employ two edge computing use cases. In the first case, the SDON middleware optimizes the overlay network to improve network resource usage, and in the second case, device resource usage.

5.1. Use Case 1: Video streaming

This use case assumes a video streaming application in which one application service running a server wants to stream data to a set of client services, each placed in a different edge network. The server and most of the clients are placed behind separate symmetric NATs, while some clients are not NATed and, therefore, publicly reachable. Since hole punching is not practically feasible over two symmetric NATs, the only option for the server to stream to NATed clients is relayed communication via one of the public clients [38].

Achieving the best performance requires determining which clients are accessible and which are only accessible via a relay. In the latter case, the application must find which device can act as the relay with the lowest latency and configure the routing table accordingly. Application developers often use a simpler-to-implement approach, in which all nodes route through a centralized relay, as shown in Fig. 4(a).

We choose this use case to indicate how our SDON middleware helps in programming an overlay network that takes over these functions, hiding the indirection caused by NATs and automatically configuring

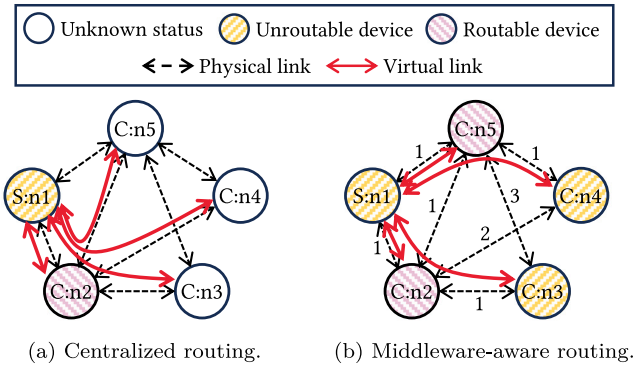


Fig. 4. In (a), server `n1` communicates with all clients `n2`–`n5` through `n2` acting as a relay. In (b), faster paths are used, thanks to additional knowledge about link latencies and NAT information.

the overlay network using the fastest physical path available, as shown in Fig. 4(b).

Listing 2 showcases the overlay network model, able to optimize routing between server and clients. 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 s. 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 server `n1` is added with the overlay IP address being set to 10.2.1.1. In lines 4 to 10, clients `n2`–`n100` are added. Initially, the server uses `n2` as a central relay to reach all clients (see Fig. 4(a)). Lines 11 to 22 define the callback function triggered upon receiving of new latency insights. This function allows identifying the fastest relay and updating the topology and routing tables accordingly (as seen in Fig. 4(b)). Although the exact Lua script for selecting the fastest relay in lines 14 to 16 is not included for brevity, the process involves creating a latency matrix based on the received latency information, identifying routable nodes, and selecting the path with the lowest latency.

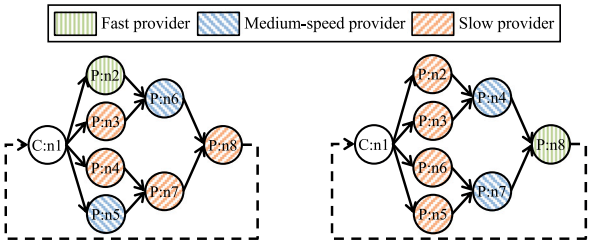
5.2. Use Case 2: Computation offloading

This use case assumes a computation offloading system (as shown in Fig. 5) where an application service (denoted as the consumer) wants to offload a batch of similar computations to a group of other services (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 Fig. 6. Here, we see how the timings of subsequent sub-tasks impact overall completion time. In Fig. 6(a), the

[1] https://www.wintun.net/.

Listing 2 SDON model for latency-aware video streaming.

```
1  net = create_network({measure_latency='0.2|60'})
2  initial_relay = 'n2'
3  net:add_node('n1', {ip='10.2.1.1',run='server.sh'})
4  for i = 2, 100 do
5   net:add_node('n'..i,
      ↪  {ip='10.2.1.'..i,run='client.sh'})
6   net:add_link('n1', initial_relay)
7   net:add_link(initial_relay, 'n'..i)
8   net:get('n1').routes['n'..i] = initial_relay
9   net:get('n'..i).routes['n1'] = initial_relay
10 end
11 net:set_callback(function(net, dev)
12  net:clear_links()
13  for i = 2, 100 do
14   latencies = latency_matrix(net:nodes())
15   possible_relays = routable_nodes(net:nodes())
16   fastest_relay = fastest_path(latencies,
       ↪  possible_relays, 'n1', 'n'..i)
17   net:add_link('n1', fastest_relay)
18   net:add_link(fastest_relay, 'n'..i)
19   net:get('n1').routes['n'..i] = fastest_relay
20   net:get('n'..i).routes['n1'] = fastest_relay
21  end
22 end)
```



(a) Randomly arranged providers. (b) Optimally arranged providers.

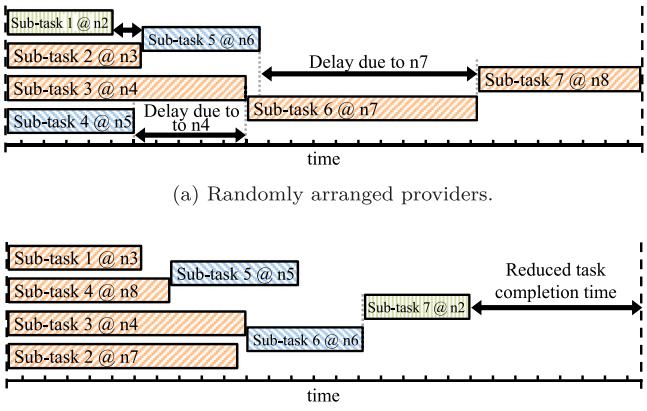
Fig. 5. In (a), sub-tasks are assigned to providers n2–n8 without considering performance or dependencies, causing delays as subsequent sub-tasks must wait for the slowest provider. In (b), providers are arranged optimally so that parallel tasks are performed by equally capable providers, resulting in a reduced overall task completion time.

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.

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 Fig. 6(b): 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.

We choose this use case to indicate how our SDON middleware helps in programming an overlay network that takes over these functionalities, hiding the complexity in performance distinction of providers and reconfiguring the overlay automatically, resulting in faster computation completion times.

Listing 3 shows the SDON model, providing functions that optimize the pipeline arrangement. In line 2, consumer n1 is added, and in the for loop after, seven providers n2–n8 are added. The magic numbers of



(a) Randomly arranged providers.

(b) Optimally arranged providers.

Fig. 6. Task completion time is reduced when fast providers are placed at the end of the processing pipeline.

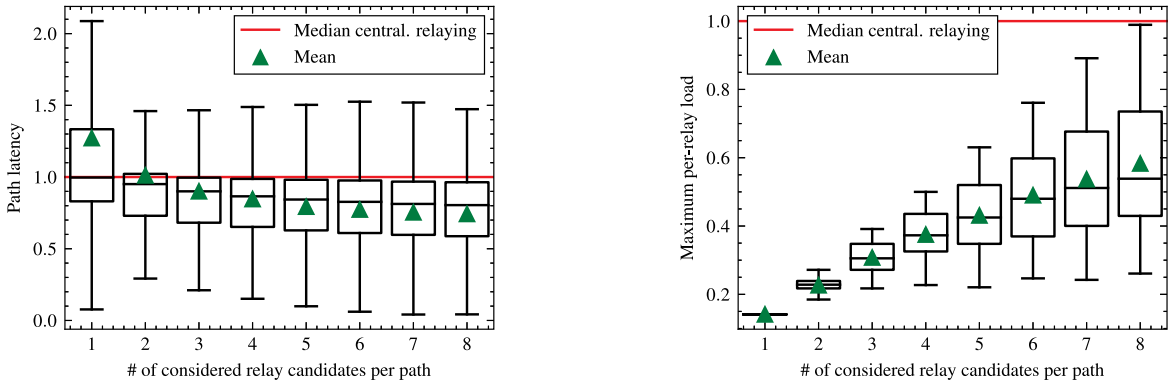
the messages the application uses to offload a task and return the result are specified for each provider. This results in SDON devices being able to identify the corresponding IP packets and report to the controller the message timings. Specifically, when a task offloading packet is received, the middleware records a timestamp marking the start of computation. When the result packet is sent back to the consumer, a second timestamp is recorded. The difference between the two timestamps reflects the processing duration of the task on that provider. These timing values are periodically reported to the controller, which stores them per node and aggregates them to compute average execution durations. Based on this performance profile, the controller ranks all providers and rearranges the overlay to place faster nodes at later pipeline stages, thereby minimizing delays caused by straggling providers. These values reflect the end-to-end processing performance (sub-task times) per provider, which is used as a basis for pipeline optimization. Further, lines 5 to 8 ensure that the consumer and all providers can reach each other. The code for clustering providers based on performance (lines 11 to 12) is omitted, which uses the time delta between offload and result messages to estimate task completion times, calculate average times per provider, compare these averages to other providers, and then create a list of providers where the fastest `sort_count` providers are flipped from their current position to the end. The for loop in lines 13 to 15 will virtually flip providers' positions in the computing pipeline by changing their overlay network IP addresses.

6. Evaluation

In this section, we present two experiments and their results that indicate the capabilities of our SDON middleware in taking over application functionalities and enhancing application performance. The first experiment corresponds to the video streaming application, and the second corresponds to the computation offloading application presented in Section 5. In the following, we describe the experimental setup, the metrics used, and the results for both experiments, respectively.

6.1. Evaluation of use Case 1: Video streaming

This experiment demonstrates how our SDON middleware optimizes an application's utilization of network resources. This is demonstrated using the use case outlined in Section 5.1. With this experiment, we address the research questions: "To what extent can SDON middleware-implemented routing improve an application's performance?" and "What is the impact on the physical network load as a result of using this routing strategy?".



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

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

Fig. 7. Median server–client-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.

Listing 3 SDON model for optimizing computation offloading.

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

6.1.1. Setup

In this experiment, we used Mininet[2] to emulate an edge environment consisting of 103 devices, with one serving as a rendezvous server, another as the SDON controller, and the remaining 101 devices acting as SDON devices. One SDON device acts as the server and the other 100 as clients. This deployment size is sufficient for our evaluation goals, as it allows us to analyze relay selection dynamics in a realistically sized and heterogeneous edge environment. Larger deployments would primarily increase system load and latency variance, but not qualitatively change the observed behavior relevant to our study. 92% of the SDON devices are placed behind separate symmetric NATs, reflecting real-world end-host statistics in edge networks [37]. Moreover, we incorporate latencies based on the King dataset [39], which provides real-world latencies between a set of 1740 Internet hosts where a random subset of 101 hosts is drawn to configure the latencies of the paths between server and clients. The controller is provided with the SDON model as shown in Listing 2, where `measure_latency` is set to `n|60` with $n \in \{\frac{0}{8}, \frac{1}{8}, \ldots, \frac{8}{8}\}$ resulting in nine individual settings. Therefore, for each setting, n relay candidates are considered for every server–client path. $n = \frac{0}{8}$ results in centralized relaying, whereby all paths use the same predefined relay. An experiment run begins with the rendezvous server and SDON controller being started, followed by all SDON devices registering to both. Subsequently, the SDON controller instructs the SDON devices to perform latency measurements with a random subset (defined by the above parameter) to choose afterwards the fastest relay, and to configure corresponding routing tables accordingly. After populating the routing tables, the server sequentially contacts all clients. The experiment was repeated 100 times, each testing all nine possible settings. For each iteration, a new random subset of latencies from the King dataset was drawn, resulting in a total of 900 runs.

[2] http://mininet.org.

6.1.2. Metrics

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

We focused on end-to-end latency as the primary performance metric, since the central QoS objective in this use case was to minimize delay under restrictive network conditions. While additional transport-layer metrics such as jitter or packet loss could have provided further detail, they were not critical to evaluating the scenario's main goal. The observed latency reductions were already sufficient to demonstrate that SDON-based routing improves performance by selecting more efficient overlay paths.

6.1.3. Results

Path latencies. Fig. 7(a) illustrates the median path latencies for server–client communications in dependence on the number of considered relays. These latencies are plotted on the y-axis and are normalized to the median latency for centralized relaying within the same experiment iteration. This normalization was conducted to compare all latencies with centralized relaying, a common (because it is simple to implement) approach to communicating across middleboxes. The x-axis represents the number of relays considered for each server–client-path to find the fastest relay through latency measurements. The solid line depicts the median latency of centralized relaying, while the triangle denotes the mean latency for each setting.

The findings demonstrate a logarithmic enhancement in both median and mean path latencies as the number of relay candidates increases. When only one relay is considered, the highest scattering is observed. This can be attributed to the random selection of a relay in this scenario, as there is no basis for selecting the optimal solution

without testing multiple relay candidates. As a result, this setting also has the same median as centralized relaying, while in all other cases, the median could be improved. Regarding the first experiment research question, including a second and third relay improves performance by 5% with each additional candidate. This improvement decreases to 2% and ultimately 1% for each subsequent candidate. Overall, performance can be increased by 20% when all eight available relays are considered.

Maximum per-relay load. Fig. 7(b) shows the median of the maximum per-relay load for each tested configuration, plotted against the number of relays used in the overlay. The y-axis values are normalized relative to the median load observed in the centralized baseline. For each iteration, the maximum number of UDP packets forwarded by any individual relay node was recorded, and the median of these maxima was computed across all runs. This metric captures how evenly traffic is distributed across relay nodes: lower values indicate a more balanced load, whereas higher values suggest the existence of overloaded relays. A high maximum per-relay load indicates that only a few relays are responsible for forwarding large portions of the traffic, pointing to an unbalanced distribution. The sub-figure is structured identically to Fig. 7(a).

The results indicate a logarithmic load growth as more and more relay candidates are considered. This trend can be attributed to the increasing probability of selecting the globally optimal relay. Also, the scattering of the load is increasing because of this. With all eight settings, the load per relay is within the load observed when a central relay is used. Regarding the second experiment research question, including a second relay increases the load by 67%. This improvement decreases to 40% and 21% for each subsequently added candidate. Overall, performance can be increased by 20% when all eight available relays are tested.

Considering the results of both figures in Fig. 7, a good trade-off of reduced latency and medium load is achieved when, for each path, half of the relays are tested.

6.2. Evaluation of use Case 2: Computation offloading

This experiment demonstrates how our SDON middleware optimizes edge resource utilization. This is demonstrated using the use case outlined in Section 5.2. Through this experiment, we explore the research questions: "To what extent can SDON middleware-implemented topology optimization improve an application's performance?".

6.2.1. Setup

In this experiment, Mininet is used again to emulate a network consisting of 10 devices, with one serving as a rendezvous server, another as the SDON controller, and the remaining eight devices acting as SDON devices. One SDON device acts as the consumer, and the other seven as providers. This deployment size is sufficient to illustrate the effects of performance-based provider selection and dynamic overlay reconfiguration. We used `cgroups` [40] to limit the CPU resources per provider to emulate a network of devices with heterogeneous capacities. For CPU capacities, we draw a random sample from a normal (Gaussian) distribution, which results in sub-task times with a mean (μ) of 1000 ms and a standard deviation (σ) of 400 ms. The controller was set up with the SDON model specified in Listing 3, where `sort_count` is set to $n \in \{0, 1, \ldots, 7\}$ resulting in eight individual settings. Therefore, with each setting, n providers are optimally placed. An experiment run begins with the rendezvous server and SDON controller being started, followed by all SDON devices registering to both. Initially, the SDON controller arranges providers randomly as no information about the individual CPU capacities is known. The consumer submits ten tasks to the providers, which results in seven sub-tasks computed on the providers for a total of 700 sub-tasks performed. As each provider reports times of sub-tasks thanks to the provided magic numbers, the controller learns about the performance of each provider. The SDON controller then rearranges the first fastest n providers. Once the overlay is reconfigured, the application will submit another batch of ten tasks. The experiment was repeated 100 times, each testing all eight possible settings. For each iteration, a new random sample of CPU capacities was drawn, resulting in a total of 800 runs.

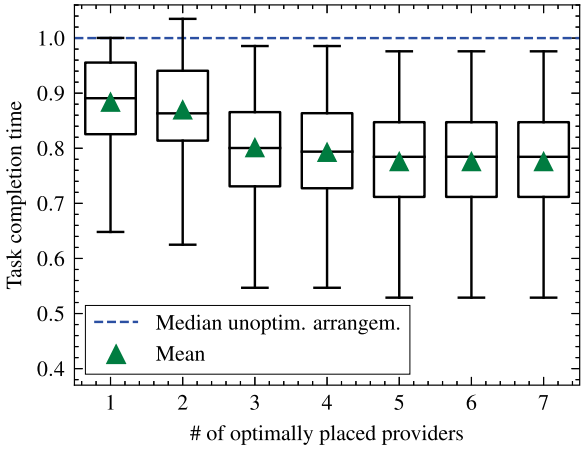


Fig. 8. Median task completion time in dependence on the number of optimally placed providers. Task completion times have been normalized to randomly placed providers within the same experiment iteration.

6.2.2. Metrics

We measure the time between each task submitted by the consumer to the first group of providers until receiving the final result from the last provider and use this as the overall computation completion time.

This metric directly reflects the objective of this scenario, which is to evaluate how effectively the SDON controller allocates computational tasks across heterogeneous providers. While alternative metrics such as CPU utilization or energy consumption could have provided a more comprehensive view of system behavior, they were not the focus of this evaluation. Notably, the observed reductions in task completion times alone sufficiently demonstrate that our middleware improves execution efficiency through optimized task placement, making additional metrics unnecessary for supporting this claim.

6.2.3. Results

The results are shown in Fig. 8, illustrating the median task completion times in dependence on the number of optimally placed providers. These times are plotted on the y-axis and normalized to the median time for an unoptimized provider arrangement ($n = 0$) observed within the same iteration. This normalization was conducted to compare all times with a random (unoptimized) arrangement, considered the baseline for its naivety. The x-axis presents 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 and more providers are placed optimally. Special attention must be paid to the setting with two optimally placed providers, which is the only setting where a performance decrease can be observed in some iterations. This decrease occurred when the initial provider arrangement had similar-performing providers at the first processing pipeline stage (sub-tasks 1–4), but then a provider from this balanced stage was swapped out for a slower provider on stage (sub-tasks 5–6). Under these circumstances, the decline in performance in the first stage three is higher 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 additionally only slightly further improves the performance, as the

likelihood of further improvements decreases as only providers within the 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 at all. Regarding the experiment research question, the best performance gain is achieved when all but not the last two providers are optimally placed, resulting in a performance gain of 22%. The biggest performance gain is observed when additional stages are optimized. Optimizing the first stage increases performance by 11%, whereas an optimized second stage increases performance by 6%. The optimization of more than three providers only leads to negligible performance gains, as only the nodes within the first stage can be repositioned.

7. Discussion

This section discusses the practical implications, limitations, and future directions of our SDON middleware, with a focus on applicability, comparison to related systems, scalability, security, and trust assumptions.

The *applicability of our middleware in real-world deployments* is supported by our evaluation setup, which uses realistic latency distributions (King dataset) and emulated NAT-constrained environments via Mininet. Moreover, earlier middleware versions have already been deployed in physical edge testbeds [41,42]. While these earlier deployments required static overlay definitions, the current system adds dynamic, runtime reconfiguration and forms a robust foundation for future operational use.

The *gap to existing systems and their limitations* becomes clear when comparing to traditional overlay tools and orchestrators. Systems like OverlayWeaver or MACEDON do not support runtime reconfiguration, while tools like Kubernetes assume full control over the infrastructure and lack built-in mechanisms for NAT traversal or overlay topology management. Our SDON middleware complements such systems by offering runtime-adaptive, intent-based overlay control tailored to constrained edge environments.

Security is a critical concern in SDN-based systems for edge environments. Our middleware includes essential protections such as device authentication, end-to-end encryption, and access control. However, the SDON controller forms a single point of trust and failure. For deployments in less trusted settings, future work must explore sandboxing of application services to better protect SDON devices from untrusted workloads, mechanisms for fair resource sharing to prevent applications from monopolizing device resources, and metric validation strategies, e.g., by cross-verifying reported values across multiple devices to detect inconsistencies.

The scalability of the SDON controller is currently limited by its centralized architecture. This design was deliberately chosen to provide a global view of the network and centralized control over overlay construction and adaptation, key advantages for coordination and consistency. As network size and complexity grow, however, maintaining global state and coordinating adaptations becomes more challenging. While our evaluation scenarios remain within a manageable scope, larger deployments may require decentralization. As a direction for future work, we envision extending the SDON middleware to not only place application services on edge devices, but also distribute selected control-plane tasks of the controller itself. This recursive placement of control logic could enable more scalable and fault-tolerant overlay coordination.

The *manual effort required to define network models and intents* remains a practical limitation. Developers must explicitly encode application goals, resource constraints, and placement preferences. While this ensures flexibility and transparency, it introduces complexity. Future extensions should explore intent abstractions, domain-specific languages, or automated model synthesis, potentially supported by machine learning.

8. Conclusion

We presented a middleware that provides application developers with central control and a common API for programming dynamic overlay networks that connect heterogeneous edge devices in restricted networks. Our lightweight and secure communication layer and the flexible overlay scripting API provide an idealized view and fine-grained control of underlying edge resources. This streamlines the development and deployment of distributed applications in real-world edge environments. Through two edge computing use cases, we demonstrated how our SDON middleware can efficiently offload application functionalities and enforce centrally driven optimizations. The evaluation results indicate an improved mean network resource utilization by up to 20% and device resource utilization 22%, respectively.

We plan to extend the SDON middleware with several features. For example, an intent-based networking API allows more complex high-level policies that the overlay network can automatically enforce. Additionally, we aim to decentralize the SDON controller, eliminating the single point of failure. Lastly, we will integrate the Container Network Interface, enabling SDON devices to be used within Kubernetes. This will facilitate the operation of Kubernetes in restricted edge environments.

CRediT authorship contribution statement

Heiko Bornholdt: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Conceptualization. **Kevin Röbert:** Validation, Software. **Stefan Schulte:** Validation, Supervision. **Janick Edinger:** Validation, Supervision. **Mathias Fischer:** Visualization, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet Things J. 3 (5) (2016) 637–646, http://dx.doi.org/10.1109/JIOT.2016.2579198.

[2] T. Qiu, J. Chi, X. Zhou, Z. Ning, M. Atiquzzaman, D.O. Wu, Edge computing in industrial internet of things: Architecture, advances and challenges, IEEE Commun. Surv. Tutor. 22 (4) (2020) 2462–2488, http://dx.doi.org/10.1109/COMST.2020.3009103.

[3] M. Asim, Y. Wang, K. Wang, P.-Q. Huang, A review on computational intelligence techniques in cloud and edge computing, IEEE Trans. Emerg. Top. Comput. Intell. 4 (6) (2020) 742–763, http://dx.doi.org/10.1109/TETCI.2020.3007905.

[4] K. Shudo, Y. Tanaka, S. Sekiguchi, Overlay weaver: An overlay construction toolkit, Comput. Commun. 31 (2) (2008) 402–412, http://dx.doi.org/10.1016/j.comcom.2007.08.002.

[5] A. Rodriguez, C. Killian, S. Bhat, D. Kostić, A. Vahdat, MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks, in: First Symposium on Networked Systems Design and Implementation, NSDI 04, USENIX Association, San Francisco, CA, 2004.

[6] L. Braubach, A. Pokahr, Developing distributed systems with active components and jadex, Scalable Comput.: Pr. Exp. 13 (2) (2012) 100–120.

[7] F. Battaglia, L. Lo Bello, A novel JXTA-based architecture for implementing heterogenous networks of things, Comput. Commun. 116 (2018) 35–62, http://dx.doi.org/10.1016/j.comcom.2017.11.002.

[8] Lightbend, Inc., Akka: Build, operate, and secure low latency systems, 2009, https://akka.io/. (Accessed 04 June 2024).

[9] H. Bornholdt, K. Röbert, S. Schulte, J. Edinger, M. 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, in: SAC '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 746–748, http://dx.doi.org/10.1145/3672608.3707729.

[10] P.A. Laplante, M.H. Kassab, Requirements Engineering for Software and Systems, fourth ed., Auerbach Publications, 2022, http://dx.doi.org/10.1201/9781003129509.

[11] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, S. Surana, Internet indirection infrastructure, 32, (4) Association for Computing Machinery, 2002, pp. 73–86, http://dx.doi.org/10.1145/964725.633033,

[12] S. Behnel, A. Buchmann, Overlay networks – implementation by specification, in: G. Alonso (Ed.), Middleware 2005, Springer Berlin Heidelberg, 2005, pp. 401–410.

[13] L. Barolli, F. Xhafa, JXTA-Overlay: A P2P platform for distributed, collaborative, and ubiquitous computing, IEEE Trans. Ind. Electron. 58 (6) (2011) 2163–2172, http://dx.doi.org/10.1109/TIE.2010.2050751.

[14] A. Aske, X. Zhao, An actor-based framework for edge computing, in: Proceedings of The10th International Conference on Utility and Cloud Computing, Association for Computing Machinery, 2017, pp. 199–200, http://dx.doi.org/10.1145/3147213.3149214.

[15] L. Nigro, Parallel theatre: An actor framework in Java for high performance computing, Simul. Model. Pr. Theory 106 (2021) 102189, http://dx.doi.org/10.1016/j.simpat.2020.102189.

[16] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, J. Ott, Oakestra: A lightweight hierarchical orchestration framework for edge computing, in: 2023 USENIX Annual Technical Conference, USENIX ATC 23, USENIX Association, 2023, pp. 215–231, URL https://www.usenix.org/conference/atc23/presentation/bartolomeo.

[17] V. Daneshmand, K.C. Subratie, R.J. Figueiredo, PolyNet: Cost- and performance-aware multi-criteria link selection in software-defined edge-to-cloud overlay networks, in: 2024 IEEE 10th International Conference on Network Softwarization, NetSoft, 2024, pp. 127–135, http://dx.doi.org/10.1109/NetSoft60951.2024.10588920.

[18] B. Ford, Unmanaged internet protocol: taming the edge network management crisis, SIGCOMM Comput. Commun. Rev. 34 (1) (2004) 93–98, http://dx.doi.org/10.1145/972374.972391.

[19] M. Kumar S.D, U. Bellur, An underlay aware, adaptive overlay for event broker networks, in: 5th Workshop on Adaptive and Reflective Middleware, ARM'06, Association for Computing Machinery, 2006, p. 4, http://dx.doi.org/10.1145/1175855.1175859.

[20] J.F. Buford, Management of peer-to-peer overlays, Int. J. Internet Protoc. Technol. 3 (1) (2008) 2–12, http://dx.doi.org/10.1504/IJIPT.2008.019291.

[21] I. Al-Oqily, A. Karmouch, Towards automating overlay network management, J. Netw. Comput. Appl. 32 (2) (2009) 461–473, http://dx.doi.org/10.1016/j.jnca.2008.02.013.

[22] Q. Yin, A. Schüpbach, J. Cappos, A. Baumann, T. Roscoe, Rhizoma: A runtime for self-deploying, self-managing overlays, in: J.M. Bacon, B.F. Cooper (Eds.), Middleware 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 184–204.

[23] M. Rossberg, G. Schaefer, T. Strufe, Distributed automatic configuration of complex ipsec-infrastructures, J. Netw. Syst. Manage. 18 (3) (2010) 300–326, http://dx.doi.org/10.1007/s10922-010-9168-7.

[24] Kubernetes Authors, Production-grade container orchestration, 2014, https://kubernetes.io/. (Accessed 04 June 2024).

[25] ZeroTier, Inc., ZeroTier - global area networking, 2014, https://github.com/zerotier/ZeroTierOne.

[26] M. Caporuscio, V. Grassi, M. Marzolla, R. Mirandola, GoPrime: A fully decentralized middleware for utility-aware service assembly, IEEE Trans. Softw. Eng. 42 (2) (2016) 136–152, http://dx.doi.org/10.1109/TSE.2015.2476797.

[27] Tailscale Inc., Best VPN service for secure networks, 2019, https://tailscale.com/. (Accessed 04 June 2024).

[28] Protocol Labs, Libp2p - modular peer-to-peer networking stack, 2020, https://libp2p.io/. (Accessed 04 June 2024).

[29] NetFoundry Inc., OpenZiti - open source zero trust networking, 2020, https://openziti.io/. (Accessed 04 June 2024).

[30] L. 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, Association for Computing Machinery, 2022, pp. 1–5, http://dx.doi.org/10.1145/3565385.3565876.

[31] K. Subratie, S. Aditya, R.J. Figueiredo, EdgeVPN: Self-organizing layer-2 virtual edge networks, Future Gener. Comput. Syst. 140 (2023) 104–116, http://dx.doi.org/10.1016/j.future.2022.10.007, URL https://www.sciencedirect.com/science/article/pii/S0167739X22003235.

[32] L. Rosa, A. Garbugli, A. Corradi, P. Bellavista, INSANE: A unified middleware for qos-aware network acceleration in edge cloud computing, in: Proceedings of the 24th International Middleware Conference, Middleware '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 57–70, http://dx.doi.org/10.1145/3590140.3629105.

[33] J.J. Diaz Rivera, M. Afaq, W.-C. 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, 2023, pp. 77–82.

[34] Open Networking Foundation, ONOS: Open network operating system, 2024, https://opennetworking.org/onos/. (Accessed 12 June 2025).

[35] The Linux Foundation, OpenDaylight: Open source SDN controller platform, 2024, https://www.opendaylight.org/. (Accessed 12 June 2025).

[36] H. Bornholdt, K. Röbert, M. Fischer, Low-latency TLS 1.3-aware hole punching, in: ICC 2023 - IEEE International Conference on Communications, IEEE, 2023, pp. 1481–1486, http://dx.doi.org/10.1109/ICC45041.2023.10279326.

[37] S. Haas, S. Karuppayah, S. Manickam, M. Mühlhäuser, M. Fischer, On the resilience of P2P-based botnet graphs, in: 2016 IEEE Conference on Communications and Network Security, CNS, 2016, pp. 225–233, http://dx.doi.org/10.1109/CNS.2016.7860489.

[38] A. Müller, G. Carle, A. Klenk, Behavior and classification of NAT devices and implications for NAT traversal, IEEE Netw. 22 (5) (2008) 14–19, http://dx.doi.org/10.1109/MNET.2008.4626227.

[39] K.P. Gummadi, S. Saroiu, S.D. Gribble, King: estimating latency between arbitrary internet end hosts, in: 2nd ACM SIGCOMM Workshop on Internet Measurment, Association for Computing Machinery, 2002, pp. 5–18, http://dx.doi.org/10.1145/637201.637203.

[40] The kernel development community, Control group v2, 2015, https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#controllers. (Accessed 24 May 2024).

[41] H. Bornholdt, K. Röbert, M. Breitbach, M. Fischer, J. 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, IEEE, 2023, pp. 212–218, http://dx.doi.org/10.1109/PerComWorkshops56833.2023.10150261.

[42] K. Röbert, H. Bornholdt, M. Fischer, J. Edinger, Latency-aware scheduling for real-time application support in edge computing, in: 6th International Workshop on Edge Systems, Analytics and Networking, ACM, 2023, pp. 13–18, http://dx.doi.org/10.1145/3578354.3592866.