DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enabling Microservice Interactions within Heterogeneous Edge Infrastructures

Giovanni Bartolomeo

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Enabling Microservice Interactions within Heterogeneous Edge Infrastructures

Aktivieren von Microservice-Interaktionen in heterogenen Edge-Infrastrukturen

Author:	Giovanni Bartolomeo
Supervisor:	Prof. Dr.-Ing. Ott Jörg
Advisor:	Dr. Mohan Nitinder
Submission Date:	15.09.2021

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2021 Giovanni Bartolomeo

Acknowledgments

A huge thanks to those who daily supported me during this journey. When everything seemed insurmountable, you were there. I can't thank you enough for believing in me and my dreams.

My thought also goes to who is not here with me anymore, despite the distance I'll always bring with me your memory.

Thanks to my friends and colleagues who helped and advised me during these years. I can count myself lucky to have had you on my side.

I also wish to thank the universities that hosed my Master's studies, TUM and UniPi, as well as all the professors and assistants that guided me through this path.

Finally, thanks to the Chair of Connected Mobility that offered me the possibility to undertake this thesis work, particularly thanks to my supervisors and advisors, Prof. Dr.-Ing. Jörg Ott, Dr. Nitinder Mohan, and Prof. Antonio Brogi, for the incredible support you gave me during the past months.

Grazie.

Munich, September 2021

Abstract

Distributed software laying across a heterogenous node network represents a challenge from many points of view. Microservices need to interact seamlessly, abstracting the infrastructure beneath and scaling at will while transparently balancing the traffic across multiple instances. In edge environments, nodes may move across network boundaries, have limited or no visibility, and turn suddenly unavailable. Developers and users cannot tolerate a decrease in the QoS but expect unconditional service availability. While such an environment provides benefits regarding the close positioning of the computation concerning the final user, there is still a lot of research in progress to find suitable strategies to enable a transition, or even better, a continuum, from the cloud to the edge. Bringing applications closer to the user comes with many efforts from the point of view of the communication. This work presents a set of networking components that overcomes the constraints of the edge environment, bringing further flexibility in the way the communication is carried out. The proposed approach exploits the positioning and the hardware capabilities of the nodes hosting the microservices instances to provide a powerful instrument for edge networking. Through semantic virtual addressing and distributed edge proxying, the services can interact and systematically specify the routing directives. Packets travel across a dynamic tunneled overlay that abstracts the networking discrepancies between the node's clusters. Service interactions happen transparently thanks to the L4 implementation and the fallback strategies that route the traffic to the most suitable available instances masquerading the network failures, even allowing distributed communication across nodes with no cloud support. The implementation has shown outstanding balancing capabilities even across multiple constrained nodes with limited visibility, performing the routing decisions even 20% faster than the best-performing competitor.

Kurzfassung

Verteilte Software, die über ein heterogenes Knotennetzwerk gelegt wird, stellt in vielerlei Hinsicht eine Herausforderung dar. Microservices müssen nahtlos interagieren, die darunter liegende Infrastruktur abstrahieren und nach Belieben skalieren, während der Datenverkehr transparent über mehrere Instanzen verteilt wird. In Edge-Umgebungen können sich Knoten über Netzwerkgrenzen hinweg bewegen, eine eingeschränkte oder keine Sichtbarkeit haben und plötzlich nicht mehr verfügbar sein. Entwickler und Benutzer können eine Verringerung der QoS nicht tolerieren, erwarten aber eine bedingungslose Dienstverfügbarkeit. Während eine solche Umgebung Vorteile hinsichtlich der engen Positionierung der Berechnung für den Endbenutzer bietet, wird noch viel geforscht, um geeignete Strategien zu finden, um einen Übergang oder besser ein Kontinuum von der Cloud zum Edge zu ermöglichen. Anwendungen näher an den Nutzer zu bringen ist aus kommunikativer Sicht mit viel Aufwand verbunden. Diese Arbeit stellt eine Reihe von Netzwerkkomponenten vor, die die Beschränkungen der Edge-Umgebung überwinden und eine weitere Flexibilität bei der Durchführung der Kommunikation ermöglichen. Der vorgeschlagene Ansatz nutzt die Positionierung und die Hardwarefähigkeiten der Knoten, die die Microservice-Instanzen hosten, um ein leistungsstarkes Instrument für Edge-Networking bereitzustellen. Durch semantische virtuelle Adressierung und verteiltes Edge-Proxying können die Dienste interagieren, indem sie die Routing-Entscheidung systematisch für jede Netzwerkanforderung festlegen. Pakete werden über ein dynamisches getunneltes Overlay übertragen, das die Netzwerkdiskrepanzen zwischen den Clustern des Knotens abstrahiert. Service-Interaktionen erfolgen transparent dank der L4-Implementierung und der Fallback-Strategien, die den Datenverkehr an die am besten geeigneten verfügbaren Instanzen weiterleiten, die Netzwerkausfälle maskieren und sogar eine verteilte Kommunikation über Knoten ohne Cloud-Unterstützung ermöglichen. Die Implementierung hat hervorragende Balancing-Fähigkeiten auch über mehrere eingeschränkte Knoten mit eingeschränkter Sichtbarkeit gezeigt und führt die Routing-Entscheidungen sogar 20% schneller aus als der leistungsstärkste Wettbewerber.

Contents

1 Introduction

1.1 Problem Statement

The centralized approach proposed by cloud computing is experiencing difficulties due to the consistently increasing number of devices and services [1]. Edge computing brings the data processing closer to its origin and the application deployment in proximity with the end-user. Distributing workloads to the edge of the network is the currently active trend pursued by developers and companies. Through latency reduction, energy optimization, context-aware computing, and an increased level of trust [2, 3, 4], the decentralization of the services represents the new frontier for modern software deployments. Instead of condensing all the traffic in a single faraway location, this paradigm uses multiple smaller computational units spread across geographical areas. The traffic is evenly distributed across the surface and can adapt according to the local loading factors. Moreover, the data produced by IoT devices do not cross the entire network before reaching a remote location. Still, it is computed locally, closer to the user, increasing privacy and reducing latencies.

Unfortunately, between the plentiful advantages of the edge computing paradigm, there are also many challenges still under active research. Thanks to the maturity of such an approach and the many benefits of centralization, cloud computing comes with symmetric, fully manageable, fault-tolerant architectures that enable scalability and reliability. At the edge of the network, there are no such guarantees regarding the underlying framework. Each computing node may be based on different architectures, like x86 or ARM, with varying networking capabilities, from bandwidth/latencies limitations to access restrictions through NATs and firewalls. This scenario increases the complexity of resources management, making it more challenging to enable developers to maintain applications seamlessly to the edge as if they are in the cloud. To create the abstraction of a uniform set of resources, it must be first possible to bring all the nodes on the same level, making them easier to monitor, provision, and connect. Thus, virtualization of hardware, applications, and network, is the key to ease the operational cost because it simplifies services deployment and reduces the configuration to enable instances interactions.

Many virtualization technologies and frameworks that enable application deployments and interconnection at the edge are usually adaptations of cloud-based technologies; they do not thoroughly address further constraints introduced in such environments. Moreover, enabling the communication between micro-services deployed at the edge turns out to be a challenge still under active research [5]. Services spread across multiple nodes, with different virtualization and network capabilities, find themselves among insurmountable boundaries. In many cases, developers must adopt complex solutions to enable services interactions. Cloud-based applications need adaptation before an edge deployment.

Thus, a technology that enables micro-service communication at the edge must envision as well transparent interactions from the point of view of the developer, lightweight footprint from the point of view of the host, and scalability from the point of view of the application.

Services must be able to interact, scale, and migrate. At the same time, the developer must manage a multi-cluster infrastructure abstracting the difficulties introduced by such a heterogeneous environment. An application deployed on a cluster must be able to communicate with any of the instances of another service, possibly spread across multiple clusters. The traffic load must be balanced, and the routes must adapt dynamically. The software developer only has to develop an application using the well-known networking stack, contacting a service with any of the well-known protocols, without worrying about the underlying framework. Furthermore, the heterogeneity of the edge environment must be exploited. According to the dynamic condition in which the nodes and services may reside, the logic behind the routing decisions changes. Thus, the developer must be able to specify a dynamic preference in the way that the platform handles traffic, for instance, deciding to exploit the node's close proximity to a service or the capacity of a remote cloud resource.

1.2 Contribution

This thesis proposes a set of networking components that enables all these features, extending and combining well-known technologies to create a new networking model that tackles the dynamic and variegated nature of edge infrastructures.

These networking components use semantic addressing [6] to give extra flexibility in the way the developers can enable communication. Services are not only represented by an IP address but by a set of addresses, each imposing a different routing technique. According to the developer's need, each network packet can be forwarded to the destination using a different routing algorithm. We can imagine, for example, sending a request to the closest instance of a specific service, or maybe to the one with higher capacity or throughput, all regardless of the virtualization technology chosen. In fact, containerized workloads can operate seamlessly with unikernels and the other way around. Another essential feature of the proposed components is the ability to support devices that reside behind NAT and firewalls out-of-the-box. All the nodes of a cluster and within clusters must be able to communicate and forward the service's packets while at the same time avoiding a single point of failures caused by the NAT traversing technique chosen. This is performed via a node self-maintained distributed and dynamic overlay network implemented via UDP tunneling. Sometimes it must also be taken into account that it is not always possible to reach the cloud. It may be far away, and the latencies can be significant; this can have an impact on the communication that can't be underestimated. In fact, the proposed communication model provides multiple fallback mechanisms, even enabling offline services interactions.

In order to achieve this result, almost the entire service-to-service communication management is delegated to the worker nodes. Each worker is responsible for the maintenance of its overlay network. While all the surrounding components on the cluster and the root managers are built to let the workers fetch the needed information for the routing and balancing decision.

The entire project's development had the constrained devices as the reference deployment target. Thus, all constituent parts, like the proxies and the balancers, have been designed to be as lightweight as possible. A worker node can dynamically and autonomously choose the destination according to the routing policy enforced by the semantic IP of the network request. These unique addresses are called ServiceIPs. Many ServiceIPs are bounded to the same service, but according to one chosen as a recipient, the logic by which the target instance is balanced changes. For instance, a network call performed to the ServiceIP associated with the "Closest" routing policy enforces the balancer to forward the packet to the closest worker node containing the service referred by the address. Along with the semantic addressing, we also propose a lazy resolution process assisted by an internal node cache. This procedure reduces the overhead at the worker level, maintaining only the essential information needed to enable communication across the services and minimizing the latencies between service interactions. Using a hierarchical DNS and a cluster-level balancer component, we provide support as well to incoming traffic originating outside the infrastructure.

As a reference architecture for the proposed networking components, we used EdgeIO, a lightweight framework for Edge workload deployments. The new networking components act as a plug-in for the framework to enable service interactions. Still, the architecture has been maintained enough general to make it usable by other orchestration platforms.

1.3 Thesis structure

Before giving a detailed description of the above-mentioned components, this thesis introduces on Chapter 2 the basic concepts behind the most common cloud and edge technologies that enable servers and services interactions. Many of the presented technologies represent the state-of-the-art used nowadays by the industries. With a bottom-up approach, the chapter starts describing the network topologies, continues with the virtualization technologies used, and concludes with the most common orchestration frameworks, platforms, and libraries. Among the presented tools, EdgeIO is detailed from a high-level perspective, showcasing the original architecture before introducing the components proposed in this thesis. The Chapter 3 begins with a list of the requirements that the networking components had to fulfill and continues with accurate characterization of each one of them through use cases, deployment procedures, and implementation status. Finally, Chapter 4 evaluates the proposed implementation in terms of overhead, latencies, bandwidth, and more. This chapter highlights the benefits and downsides of the network components, setting up the ground for the future developments of the system. Chapter 5 concludes this thesis with a description of the limitations that we found in this approach and the future work envisioned to overcome them.

2 Background

While Cloud computing offers high availability, virtualized, shared, and dynamic computing resources but with relatively high power consumption [7]. There is a paradigm shift from a centralized approach, where all the data, the computation, and the access to the resources are provisioned in one place, large datacenters [8], to a model where the data and the computation are distributed, closer to the user and on less capable devices, edge devices [3, 4]. Edge computing offers a horizontal system-level architecture that enables a cloud-to-thing-continuum thanks to compute, storage, and networking functions closer to the users [9]. Most of the current orchestration platforms rely on cloud assumptions, and most of the edge-based technologies are fueled by cloud technologies. The migration from cloud to edge is slow and controverted. It is essential to notice that the cloud is there to stay, and actually, the edge does not have to replace it but instead is more focused on creating a continuum to fulfill the blindspot that the cloud left behind.

On the one hand, among the reasons that currently drive the research to this decentralized approach is the increasing amount of data generated by the IoT devices, the latency constraints, privacy policy enforcement, and the lack of context-aware computation. Scaling the cloud and the network to the upcoming data magnitude may be unfeasible [2]. Even managing the users' data closer to where they are generated reduces the privacy concerns [10], but it opens the issues about securing constrained devices. On the other hand, the enterprise adoption of these technologies brings new challenges [11]. Production deployments mainly need guarantees in terms of scalability and fault tolerance. Instead of focusing on low latencies, enterprises want to masquerade any possible failure due to network unreliability and cloud outages. This is why the edge can open up to on-premise processing with redundant outbound links both to users and the cloud. In this latter scenario, the cloud is more of an escape valve to offload the traffic that the edge can't handle. This kind of infrastructure is usually called *edge-site deployment*. The term underlines that the computation mainly happens at the edge, and if the networks permit, the cloud is used alongside it.

Even if edge computing can bring many benefits, the open technological challenges are still many, and the research is still in progress. First of all, many different flavors of edge computing exist, each one with its own challenges. Secondly, the vast majority of the platforms and components running the cloud computing paradigm are not always suitable to handle the heterogeneity of the edge. As pointed out in [12] the different edge computing facets go from fog computing[13] to cloudlets[14], and continues up to the mobile edge computing [15]. The many terminologies currently in place to describe such environments may appear confusing and sometimes even overlapping in the context. Usually, even in this work, the terms fog and edge are used almost interchangeably, even though there is a slight difference.

Figure 2.1 shows a simplified landscape representing the main concepts behind the fog,

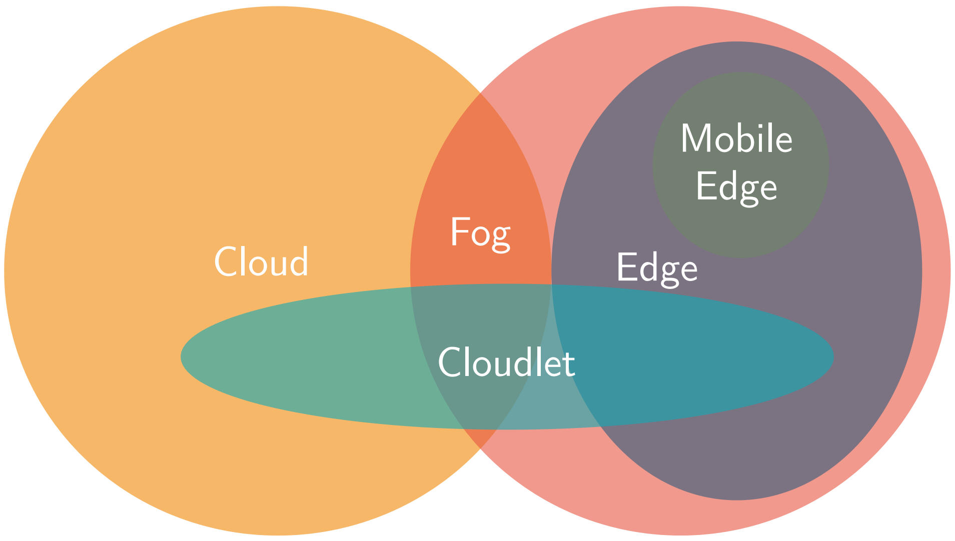


Figure 2.1: Fog/Edge landscape

edge, and mobile edge terminology. From the literature, [9] the main difference between *edge* and *fog* is given by the fact that while the fog is hierarchical and provides computing, networking, and storage from cloud to things, edge computing is limited to extend the computation at the network's edge. Then, technically, placing devices at the edge means putting them within the boundary of another subnetwork.

Cloudlets are instead resource-rich computers, with solid and redundant links placed at the edge. Cloudlets provide high reliability and low latencies along with users' close proximity. Logically cloudlets are placed exactly in the middle between mobile devices and the cloud.

Unlike pure mobile computing, mobile edge computing (MEC) represents the edge in mobile environments, with communication over radio or mobile connections and devices that can rapidly change their location. The main challenges of MEC are not only represented by the limited and discontinued communication or constrained devices but as well by the geographical and context awareness of these resources [16].

Modern applications are composed of multiple micro-services that cooperate together to respect non-functional requirements such as scalability, fault tolerance, independence, and more [17]. These highly decentralized applications bring challenges regarding the deployment and management of the work units. Each service may need different hardware capabilities, specific geographical requirements, or dynamic orchestration to adapt the workload according to the service level agreements (SLA) [18]. The use of virtualization is the key strategy that enables cloud computing to manage these complex applications deployments. Abstracting the underlying hardware makes the isolated services free to move and scale across the shared assets and enables as well the on-demand provisioning of new resources [19]. Unfortunately, application and hardware virtualization the way we know it does not work well at the edge due to the device's limitations and the heterogeneous environment. New lightweight approaches may be needed [20]. Moreover, the apps on edge are not really free to move since the communication across them may be compromised. In general, many platforms require the devices to be in the same network to interact. Even the single compute node with the underlying network fabric now needs to be abstracted, provisioned, and managed so that the application behaves like in the cloud. In this case, a refinement of the well-known cloud network virtualization techniques can masquerade the underlying fabric, easing the management of the routes that interconnect multiple services. Then, what is currently needed is a unique platform that can connect nodes, enable them to run multiple virtualization technologies, and transparently abstract the failures of these constrained devices and the surrounding environment. Many nodes can belong to private premises, behind NAT and firewalls, and with limited exposable ports. Links and workers can suddenly turn unavailable, and the traffic must be redirected somewhere else. Because of the distributed nature and the geographical position of the devices, even new approaches to the routing problem are required [21, 22].

In the following sections, the cloud and the edge networking state-of-the-art technologies are explored, pointing out what is currently missing to enable the above-mentioned abstractions and the contribution of the proposed work.

2.1 Communication in a Cloud environment

Most of the technologies that nowadays fuel the edge were originally adopted or invented because of the consistent spread of cloud technologies. Since the service requirements that undergo modern applications remain unchanged, these technologies rest important pillars in the development of edge infrastructures. Then, the complexities addressed for the DCNs (Data Center Networks) partially overlap the requirements of the edge as well. This section systematically presents the topologies, techniques, and challenges currently faced by the cloud with a bottom-up approach. With this big-picture in mind, the following sections compare this environment with the edge, highlighting the differences and describing the further steps needed to undertake the latest complexities introduced.

In cloud environments, due to the centralization of the computation, the amount of flowing data is enormous [1]. Thus the bandwidth handled by these infrastructures must befit the traffic generated by the services residing inside the datacenter and the clients outside. The network links must be redundant [23], a single failure can cause an entire datacenter to turn offline, thus thousands of services risks to be cut out from the internet [24]. The cloud must also provide simplified infrastructure maintenance while improving the flexibility for the multi-tenant demands of the enterprises. Moreover, the high-velocity data throughput must be supported to enhance the applications' QoS [25]. Lastly, while scaling up and down VMs, it must be possible to attach them to the network dynamically, provisioning the IP addresses and satisfying the bandwidth and latency requirements. From the point of view of the user that is requiring a new VM, the cloud has almost unlimited resources. Thus the entire underlying infrastructure must be invisible.

To fully understand the way the communication within a DCN is handled, it must be analyzed from different angles: the physical topology §2.1.1 and the virtualized infrastructure

(§2.1.2 and §2.2.4). Each networking architecture of a datacenter is built from two directions, designing first a topology that builds a cost-effective DCN to scale up the datacenter, and secondly developing networking and routing techniques that address the challenges of the existing topology [26].

On top of the cloud infrastructure then an orchestration platform can be used to handle containers and tasks programmed by the developers. Common orchestration platforms enable communication across different nodes, building overlay networks and assigning private addresses to the services. This further abstraction relies on top of the DCN networking, and it is easy to achieve when the underlying infrastructure is already virtualized and homogeneous.

2.1.1 Networking topologies

To handle distributed, high-demanding, and unpredictable application workloads, the underlying physical arrangement of the cloud network is the foundation on top of which it is possible to build abstraction. Large datacenters can even connect hundreds or thousands of servers [27]. For this reason, the research spent a lot of effort proposing several topologies models during the years. The goal is to handle such a massive amount of data with an efficient, cost-effective, and scalable design, and infer the properties and technologies that can then be used at the edge as well. To explore the problem space [26] fixes a set of features that can help to evaluate the different proposed topologies:

- *Parallel traffic handling* This refers to the capacity of handling the traffic of many applications in parallel. Several hundred applications can run in parallel on top of each rack and may need to access the medium altogether. The topology must be able to handle the traffic of the applications without disrupting the service.

- *Bandwidth oversubscription* The capacity of allocating several services such that the Service Level Agreement (SLA) requirements can still be satisfied. Each application deployed in a cloud environment comes with a set of conditions that are expected to be satisfied by the platform. The bandwidth available inside the datacenter is limited, and it is quite easy to incur into the oversubscription problem. This means that the bandwidth used by the applications surpasses the available physical bandwidth. Centralizing many applications in the same datacenter increases such risk dramatically.

- *Backwards capability* Reversing the direction of the traffic does not impact the performance. Each application expects a certain degree of symmetry in the way the nodes communicate. If the developer flips two services, client and server, across two nodes, there should be no difference in terms of network performance.

- *Scalability* This is the ability to handle the expansion in the number of servers. Datacenters' dimensions increase rapidly, and when extra capacity is needed, extending the number of servers attached to the network must be easy. A physical topology is expected to give flexibility in terms of hardware upgrades.

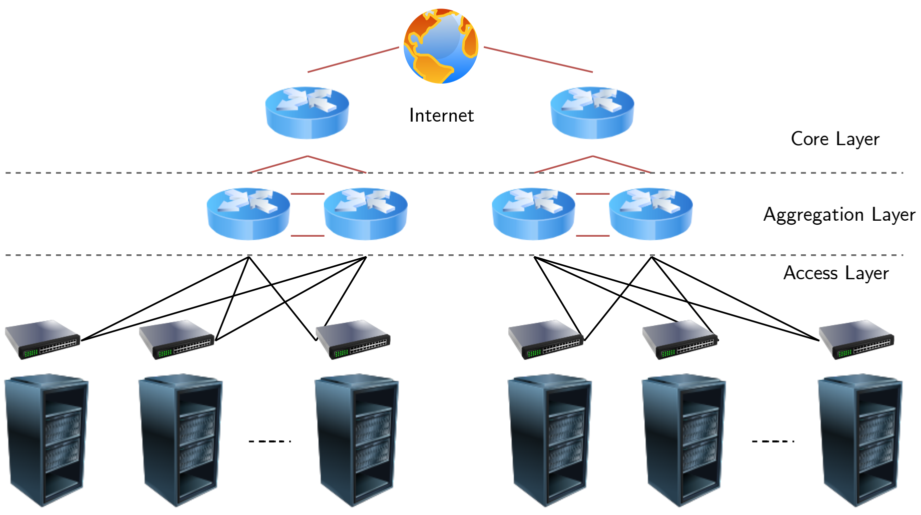


Figure 2.2: Cloud three tier topology

- *Automatic naming* This refers to the capacity of self-learning the addressing space in which the server resides without manual configuration. When attaching new racks to the networks, it must be possible for them to auto-configure themselves. Due to the dimension of such infrastructure, sometimes it is almost impossible to manage the configuration manually. An automatic design even reduces the risk of human errors. The operators should intervene in the network only when something fails.

- *Robustness* This metric considers the fault tolerance in case of a switch or link failure. In datacenters happens almost daily that some switches or routers fail. Even links can become unavailable, or components simply need to be replaced. It is not realistic to stop an entire datacenter in order to change a single switch. This is why a significant amount of replication in the components is needed. Moreover, an easily monitorable and maintainable infrastructure highly reduces the risk of failures and thus increases the robustness.

- *Load balance* This refers to the ability to balance the workloads avoiding bottlenecks. The internal north-south and east-west traffic are almost unpredictable; some links can saturate, and some routers and switches get congested. The infrastructure must redirect the traffic accordingly to avoid routes that are already saturated. One of the primary requisites for load balancing is the availability of alternative paths and the ability to choose among them accordingly.

One of the well-known topologies in the literature is the **Three-tier** DCN [28]. This topology is composed of a tree hierarchy of switches and routers divided into three layers. The top layer is called *core layer*, the middle tier forms the *aggregation layer* and the leaves are the *edge/access layer*. In this architecture, the links capacity increases from bottom to top. As shown in fig fig. 2.2 all the access layers Top of Rack (TOR) switches are connected to all the aggregation layer routers. The aggregation layer routers are replicated to improve fault tolerance and connected to the core layer routers that access the internet. This topology forces to have links with very high capacity on top of the tree (in red) compared to the ones on the bottom (in black) to limit oversubscription. This leads to higher costs.

Topologies like this one are called hierarchical, and many other variants have been proposed over time to overcome its shortcomings, like the **Fat-tree** [29] and **VL2** [30] topology. In general, the main problems connected to hierarchical topologies are related to scalability and parallel traffic handling. Scaling a hierarchical infrastructure when it comes to complex topologies results requires consistent investment. In real-world setups, like the one proposed in [31], it has been pointed out that even the load-balancing in these topologies may not be optimal.

Another approach to cloud topologies is the one proposed in [32] which targets scalability introducing DCells. It uses a recursively defined structure to interconnect servers. Each server connects to different levels of DCells via multiple links. High-level DCells are built recursively from many low-level ones. In infrastructures using this schema, it is possible to expand and scale the DCN recursively. For instance, to create a $DCell_1$ with $n = 4$ servers for each $DCell_0$ is possible to proceed as follows. Every 4 server are attached to a *mini-switch* form a $DCell_0$. The servers have 2 links, one to the switch of the same DCell and another to a server in another DCell. To compose a full $DCell_1$, $n + 1$ $DCell_0$ are needed. This infrastructure can scale up to k layers. Given a $DCell_{k-1}$ composed of t_{k-1} servers, a $DCell_k$ is composed of $t_{k-1} + 1$ $DCell_{k-1}$.

DCN topologies like DCell feature a very high level of scalability, parallel traffic handling, and algorithms for automatic scaling. Unfortunately, as pointed out in [33] the main drawback is the high oversubscription ratio. This can lead to congestion and can be a bottleneck when traffic increases. An alternative approach to DCell that mitigates these problems can be BCube [34]. In general, even if these recursive approaches seem promising, due to the lack of field testing, the oversubscription problems, the higher cabling costs, and the need for more capable network interfaces, they are not very used in real-world datacenters.

In both cases, for the hierarchical and the recursive topology, the bottlenecks happened mainly in the links. The oversubscription heavily limits the scalability and the bandwidth across the servers. More innovative approaches to the topology problem are defined by Jellyfish [31], and Scafida [35]. They propose respectively the use of a random graph and a scale-free topology instead of fixed symmetric topology. In these solutions, asymmetry is the key to overcoming the limitations mentioned above and achieving scaling in smaller and less homogenous environments. Unfortunately, due to the highly centralized nature of cloud computing, bringing randomization in the node positioning and thus enforcing a loose topology creates conflicts with the commonly used networking techniques. The cost of routing such a massive amount of traffic increases too much, and even granting fault tolerance becomes costly because of the significant amount of redundant paths that must be introduced.

One of the most common state-of-the-art DCN topology approaches is the Spine-and-Leaf architecture [36]. It technically belongs to the hierarchical topologies but has characteristics that enabled a high level of scalability and made the oversubscription problem easy to

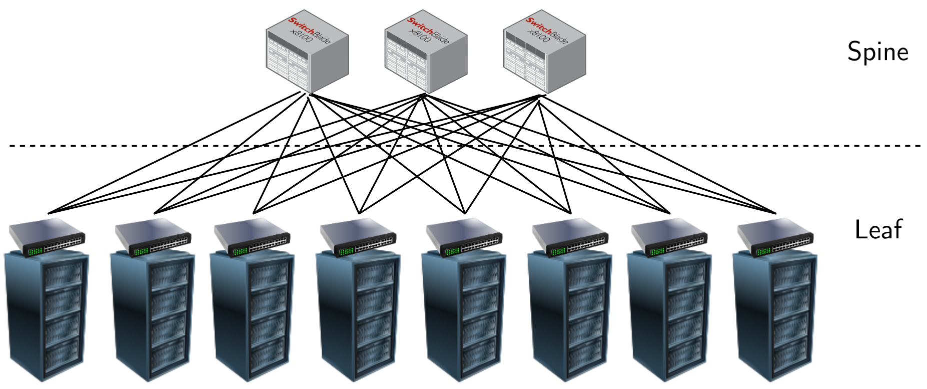


Figure 2.3: Two-tiered spine-and-leaf topology

overcome.

Figure 2.3 shows one of the most traditional implementations of this topology, the two-tiered spine-and-leaf. Every lower-tier switch (leaf layer) is connected to each top-tier switch (spine layer) in a full-mesh topology. The leaf layer is composed of switches that connect each rack. The spine layer interconnects all the leaves and is the backbone of the network. In the fabric, every leaf has a link that connects it to every spine switch. The multiple spine switches are there to enable load balancing, fault tolerance and increase scalability. In fact, the path to the spine switch is randomly chosen from each leaf to split the traffic evenly. In case of oversubscription in the topology, simply adding an additional spine switch will solve the problem. If device port capacity becomes a concern, a new leaf switch can be added by connecting it to every spine switch. The symmetric and deterministic nature of this topology also makes the routing task easier. The main drawback of this approach is the cost. The larger the number of leaf switches needed to uplink all of the physical hosts, the wider the spine needs to be, and the heavier is going to be the burden of cabling all together [37].

The capacity of handling hundreds or thousands of servers is required not only on large datacenter but can be transferred as well to small regional clouds, like in cloudlets [14] scenarios. Unfortunately, as opposed to these centralized approaches, edge deployments may even span across multiple locations, contemplating heterogeneous resources. The symmetry and level of replication shown in topologies like spine-and-leaf, DCells, and the three-tier are not always achievable on the private edge, or even worse, on mobile-edge scenarios. Moreover, as just discussed, any attempt of bringing asymmetry to these topologies, like in the case of Scafida and Jellyfish, creates difficulties in the actuation of the networking techniques developed. The edge is asymmetric by definition, and as explained in §2.3.1 the heterogeneity of this environment does not give any guarantee regarding the underlying fabric. On the bright side, the decentralization of the computation relaxes some of the networking requirements. The degree of parallelism handled by an edge node is many orders of magnitude lower than the cloud. Thus, even a lower degree of scalability is acceptable as well. While the cloud networking techniques reflect the topologies presented in this section, the challenge resides in finding the correct trade-off between flexibility and performance that enables networking between topologies composed of thousands of nodes but in distributed environments as well.

2.1.2 Traditional networking techniques

When in a datacenter thousands of servers are interconnected with the topologies mentioned in §2.1.1, the responsibility of forwarding the traffic correctly falls to the networking techniques in use. A good topology makes it easier to route and scale the traffic. However, efficient routing techniques are is still needed to achieve all the requirements of QoS, flexibility, and bandwidth that such a centralized architecture must handle. If, on the one hand, the cloud has the benefits of having complete control over the communication medium, the other aspect to consider is that such infrastructure is costly and complex to maintain. The physical layer alone can't help handle the scale; techniques that enable routing management and reduce networking complexity are needed.

Among the most traditional approaches resides the well-known Optical Networking [38]. Datacenters usually connect servers and switches using optical fiber. The information then travels in the form of light signals. Each packet, to be correctly dispatched, can be handled directly in the form of light. In this case, the switches are called *all-optical*. Otherwise, the packed can be first converted into electronic form and then can be switched electronically (*electro-optical switching*). These switches usually pre-configure the static routing paths employing optical or electronic circuits, respectively.

In order to compensate for the cost of using optical fiber cables, there are also some proposals contemplating wireless DCN architectures. For instance, the 60GHz W-DCN [39] approach places 60 GHz radios on the top of each rack to connect pairs of ToR switches.

Each one of the topologies mentioned in §2.1.1 uses routing schemes that highly depend on the underlying fabric to be able to interconnect racks and servers. For instance, server-centric structures like Dcell [32] and BCube [34] uses nodes for the multihop communication. Each switch connects a constant number of servers with a recursive structure. The servers use an address array such that they are neighbors if and only if their address arrays differ in one digit. Two neighboring servers that connect to the same level *i* switch are different at the *i-th* digit. From source to destination, the routing consists in just correcting one digit for each hop.

Topologies like Spine-and-leaf [36] are based on a layer 3 routing principle to optimize throughput and reduce latency. Routing protocols like eBGP are used for the IP fabric, while leaf switches can also have iBGP configured between the pairs for resiliency. Topologies like this are also typically configured to implement Equal Cost Multipathing (ECMP) to allow access to any spine switch in the layer 3 routing fabric. Each leaf node maintains multiple paths to each spine switch. If a spine switch fails, the is no impact as long as there are other active paths to adjacent spine switches.

These traditional networking techniques solve the problem of routing the traffic across the racks and servers. Still, they are relatively static, lack flexibility, and require a manual configuration investment. There is still the need to virtualize the fabric, the connectivity, and the routing even further to reach the level of generality needed to deploy multiple applications distributed across the infrastructure. Moreover, the manual configurations, the need for symmetry, and full control over the fabric make these networking techniques hard to use for the edge as well. There is still the need for another abstraction layer that increases the independence of the networking from the underlying fabric, enabling the applications to transparently move and interact across the infrastructure, whether they reside in a datacenter or a remote device located into a private network.

2.2 Virtualized networking

Network virtualization is the key to the current, and future success of cloud computing [40]. What is still missing from the picture above is the ease of management and the adaptation to the dynamicity of the resource requirements. Datacenter requirements change rapidly; the infrastructure must expand and restrain elastically to adapt the resource usage to this shared and unpredictable environment. Moreover, centralizing the traffic and the requests enforces the need to isolate and manage the resources efficiently. Virtual devices are easier to control, access, provision, and share. Since the end of the 90s with the VLANs §2.2.1 (virtual LAN) it was clear that virtualization was the key to achieve a future where applications are spread across multiple bridged devices. Segmenting virtually the network into subnetworks boosted the performance and improved the security. Later on, with the VPN technology §2.2.2, it was possible to extend local area networks across multiple sites seamlessly. Finally, in 2005, the revolution of software-defined networking approaches began §2.2.4.

2.2.1 Virtual LAN

A set of computers connected within the same LAN can be grouped in a partitioned and isolated subnetwork at network layer 2. Thus, Virtual Local Area Networks (VLAN - IEEE 802.1Q) separate an existing physical network into multiple logical networks. A client's membership inside a VLAN is configured via software, thus making the management and maintenance way more effortless.

A standard L2 frame, fig. 2.4, has a 14 bytes long header composed of the source and destination MAC addresses accompanied with the *Ether Type* which can contain the length of the packet or the protocol encapsulated in the payload. After the header, the frame continues with the actual payload (an upper layer packet) and a CRC.

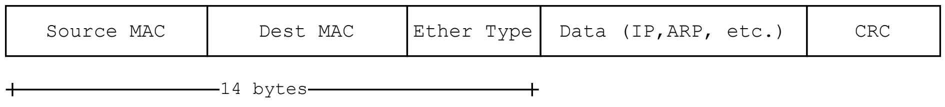


Figure 2.4: L2 frame

An L2 frame that uses the IEEE 802.1Q standard has additionally a 4-byte long TAG fig. 2.5 which is used to identify the VLAN.

There are two types of VLAN, tagged and untagged. The former operates with the hosts unaware of being in a virtual LAN and the switches that expose "untagged" ports. The client

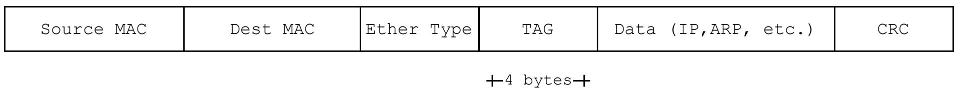


Figure 2.5: L2 tagged frame

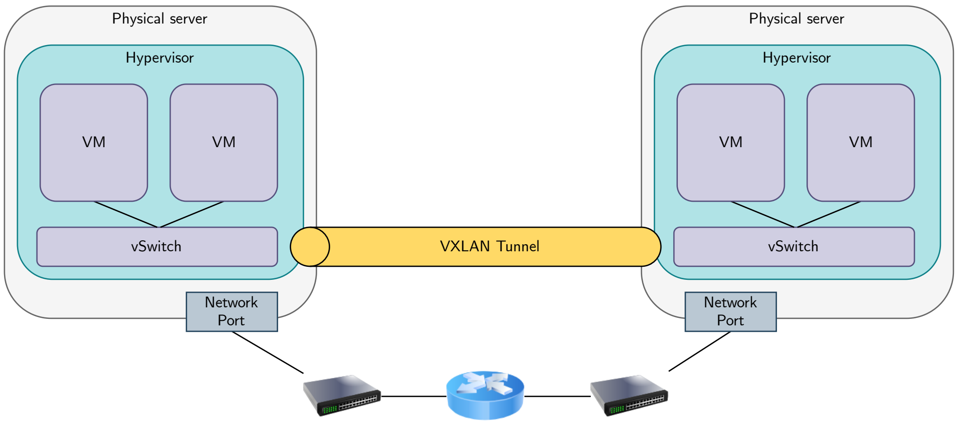


Figure 2.6: VXLAN network

sends the packets without the tag, and the switches are configured to add the tag to all the packets coming from the untagged port. When a frame leaves an untagged port, the switch strips the VLAN tag from the frame. The latter type is the opposite. The hosts are connected to "tagged" ports, which expect the tag value in the frame. In this case, the switch just forwards the packet accordingly.

A further step to the virtualization level needed is represented by the VXLAN [41] approach. Due to the rapid increase of servers and deployed virtualized services inside the datacenters, the VLAN approach may expose few limitations. VLANs only enable 4096 different domains because of the 4-byte long tag and only limits the infrastructure to layer 2 boundaries. These limitations highly impact the scalability and collide with the benefits of a virtualization technique. In a datacenter, multiple virtual machines can be instantiated inside a single node, and multiple nodes are connected together. The main problem remains how to make numerous virtual machines able to address services spanning across multiple locations interconnected over an L3 network. VXLAN solves the problem by encapsulating Layer 2 Ethernet frames in Layer 3 UDP packets. The original L2 frame, together with a new VXLAN header, is wrapped around a UDP packet before being forwarded. The new header comprises an 8-byte VXLAN header containing two reserved fields, a flag area, and a 24-bit VXLAN Network Identifier (VNI). Note that as opposed to the VLAN 4-byte long tag, the VNI allows up to 16 million different VXLANs. As shown in fig. 2.6, to enable the rack to rack communication, a tunnel is established between two top of rack (TOR) switches to encapsulate the original data frames sent by the source server. The destination TOR switch decapsulates these packets into the original data frames and forwards them to the destination server.

2.2.2 Virtual Private Networks

With containers and VMs enabling distributed computation, increased the need for a further abstraction that allows the creation of a unique and secure network spanning multiple locations. Software may need to move across a federated environment and securely communicate with remote devices. Exposing servers directly to the public internet may be risky, and a new solution was required to virtually extend a private network using the public internet [42]. The Virtual Private Network (VPN) technology enables such abstraction, allowing the users to communicate across shared or public networks. VMs may even be located on different datacenters but still, be able to communicate as if they are in the same network. The VPN technology builds a virtual topology on top of the existing, shared physical network infrastructure [43].

This technology encloses two main subcategories, **Remote Access** and **Site-to-Site** VPNs. The former uses an *Access Server* placed inside a network to extend the network boundaries to many other external clients. The client authenticates to the VPN server, encrypts and tunnels all the traffic to that server. The server decrypts the traffic and forwards it within the boundaries of the internal network. The client residing to the external network forwards all the traffic using the public internet, but thanks to the encryption, the traffic remains private. The latter subcategory is used to create a bridge between two networks, letting them act as they're one. Site-to-site VPNs are used when companies with dislocated sites need to have a unique network. This subcategory can be split even further in Intranet (multiple sites of the same organization) and Extranet (multiple sites of different organizations) Site-to-site VPNs.

To maintain the confidentiality in the communication across the locations, VPN technologies uses protocols like IPsec/L2TP [44] and SSL/TLS [45] for encryption respectively at layer 2 and 4.

With this technology, whether in the case of Site-to-site or Remote Access implementation, the client's traffic is tunneled from source to destination. There are dedicated VPN routers or VPN Servers that perform the packet encapsulation, encryption, decryption, and decapsulation process. The exact procedure, protocols, and primitive used highly depend on the specific implementation.

One of the most widely known VPN is OpenVPN [46], commonly used to create point-to-point and site-to-site connections. OpenVPN encapsulates the client's traffic in UDP packets over a TUN device installed on each device.

The routing policies, the scalability, the bandwidth, and the fault tolerance of a VPN change along with its implementation. In [47] it is possible to observe a comparison between OpenVPN, Wireguard [48], ZeroTier [49], Tinc [50] and SoftEther [51]. The paper shows that the lightweight WireGuard implementation makes it a suitable approach in terms of scalability even at the edge. In fact, even some of the orchestration platforms shown in §2.5 use WireGuard as the default VPN.

The success of the virtual private networks does not reside only in cloud environments, but their popularity lives at the Edge as well. Distributed devices across multiple heterogeneous locations may use the benefits of a virtual network that extends and flattens the boundaries. Anyhow, because of the dynamic network membership and the visibility limitations intrinsic at the Edge, a fully decentralized VPN approach with P2P virtual networks has been proposed. IP-over-P2P [52] self organizes the VPN membership of nodes in a peer-to-peer fashion adding dynamicity to the packet tunneling and routing policies. Without the need for external entities and access servers, peers autonomously organize the tunneling to enable private and secure communication across multiple network locations.

Thanks to the VPN approach unified with the previously introduced VXLAN, VMs can transparently communicate and move freely across the fabric of the physical topology. The physical layer is completely hidden, the addressing space is virtualized, and all the entities are unaware of their position in the infrastructure.

2.2.3 Overlay networks

Using the presented virtualization techniques (§2.2.1 and §2.2.2) or others like NVGRE [53] or STT [54] it is possible to completely build a new topology, a virtual topology, on top of the pre-existing fabric. The new topology is formally defined as an overlay network, a place where the links between nodes and virtual machines do not correspond to the real fabric. Then, there is a distinction between the Over and Underlay network. The former is the virtually created topology, and the latter is the physical infrastructure. In overlays, what is abstracted as a single jump link may even be a connection that needs to travel across the entire fabric of a site and traverse the whole globe before reaching the destination into another site [55]. An overlay can help managing topology changes, service migrations, and outages [56]. Overlay networks are fundamental building blocks needed to abstract service-to-service communication. Multiple overlay networks can even be built one on top of the other.

Virtualizing the entire networking substrate abstracts the complexity of the connectivity management and allows developers to easily monitor the traffic to improve the deployed services' quality. Moreover, a virtualized environment can help reach the requirements expressed in the SLA, augmenting the system's fault tolerance.

2.2.4 Software Defined Networks

The last revolution in cloud networking, which had an impact on the edge as well, is the Software-Defined Network technology. The entire concept behind this paradigm relies upon the separation of responsibilities between the control plane and the data plane. The control plane is the part of a network that controls how data packets are forwarded, changing the network topology. In contrast, the data plane represents the actual way the data flows in the network [57]. In this approach, the logic and intelligence behind the packet forwarding are entirely dissociated with the actual routing. Some specialized controllers form the brain behind the routing decisions and influence the traffic. SDN compliant routers expose API that enables routing configuration, allowing application developers to customize the routes. This reduces operation costs, provides granular security, and allows centralized network provisioning [58]. In Cloud environments, SDN approaches succeeded because they provide a higher level of abstraction than the traditional approach, avoiding even further intervention in the physical interfaces with a remote and complete software network management.

The most used SDN standard is OpenFlow [59] which includes the OpenFlow controller, OpenFlow-enabled switch, OpenFlow channel, and OpenFlow protocol. Each switch contains one or more flow tables composed of flow entries representing the traffic forwarding rules. The traffic is mapped using matching fields and rules to check the incoming packets. Upon matching a packet, each flow entry contains a set of instructions that defines the routing decision.

Even if there are already many different implementations, like ForCES [60] or Open Daylight [61], and many enterprise and cloud application already uses SDN compliant infrastructures [62], new SDN approaches are still under research. Many Edge computing projects envision the use of SDN to enable application communication in heterogeneous environments [63] [64] [65].

Software-defined networking approaches are very versatile and promising, but they usually require specialized hardware or dedicated nodes along with infrastructure control. These requirements are easily achievable in cloud environments with their fully customizable fabric but not at the edge, where it may not always be feasible to deploy a custom network architecture [66]. Moreover, SDNs require a high level of optimization for the algorithms and the tables size to avoid additive latencies that can cause impact in the QoS of the deployed applications [67].

2.3 Communication at the Edge

Compared to the networking in cloud environments, with symmetric infrastructures, fully manageable fabric, and homogeneous resource placement, the Edge environment may be more challenging. Considering the edge as an extension of the cloud means that the usual IT infrastructure is extended with an additional distributed network of heterogeneous devices [5]. Enabling application communication at the edge poses challenges on different sides, from the creation of the communication link to its maintenance, from the privacy requirement to the traffic balance of the generated data. Resources may be on environments with a limited network, links are not as robust as in the cloud, resources can move, and furthermore, each device doesn't answer with the same latency. Many solutions currently in place to enable communication at the edge uses tunneling [68], and overlay networks [69] to bring all the devices in the same network. The heterogeneity challenge can instead be tackled using context-aware approaches [70], QoS-enabled balancing [71], and dynamic networking architectures like [72]. Many systems running at the edge use well-known cloud-enabled technologies, like SDN-based approaches, [73], but usually requiring specialized components or complex configurations.

2.3.1 Heterogeneity

Figure 2.7 shows an example topology for an Edge environment. It is possible to notice that the topology is homogeneous in terms of hardware capabilities, architecture, network fabric, and sub-networks. Devices may have a local ad-hoc network or may be connected

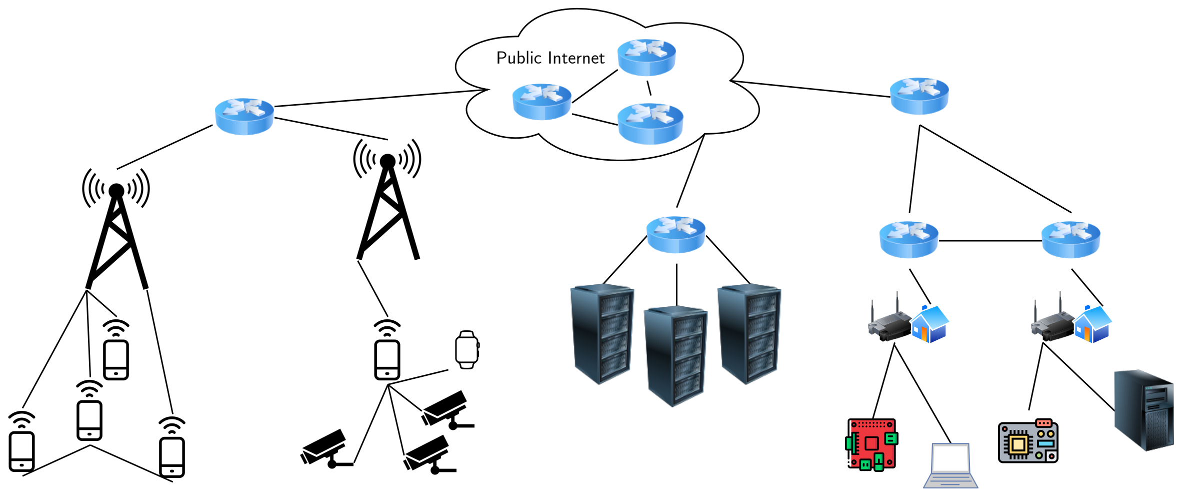


Figure 2.7: Edge Computing topology example

through the cellular antenna [74]. Furthermore, there is the lack of symmetry, which can be a limiting factor when it comes to balancing the requests and schedule the services [75]. Handling the heterogeneity of these environments may require custom routers or nodes which enable the configuration of the control plane, but the fabric is not always accessible and customizable, and the deployment of specialized nodes is not always feasible; sophisticated software solutions are essential to tackle the challenge.

As discussed in [12], the Fog and Edge span in the gap between the cloud and the user, the position of the device influences its capabilities, latency, and availability. For instance, Cloudlets [14] environments, as they are intended to be small regional clouds, uses a network fabric and a networking schema very similar to the one discussed in §2.1. While in Mobile Edge Computing (MEC), as the devices may move along with the end-user, the networking is the same as mobile devices. A base station that covers a cell acts as the gateway for the node's traffic [76]. In these scenarios, technologies like RAN/C-RAN are used. Moreover, IoT devices may even be interconnected through Personal Area Networks (PAN). These networks are composed of a set of devices connected point to point with an ad-hoc network, using protocols such as ZigBee [77] or Bluetooth. Device residing in a PAN may need a gateway device connected to the internet to be able to communicate to the cloud or external users. The gateway device can already be part of a MEC environment or connected over a Metropolitan Area Network (MAN) or a Wide Area Network (WAN). Thus, the transfer medium used may not be predictable, as well as the latencies involved and the bandwidth supported.

2.3.2 Network boundaries

As it is possible to notice from the topology scheme mentioned above, many devices may reside in private networks behind NAT and firewalls. Custom rules and port mapping must be configured into the system in order to enable service interactions. Nevertheless, constrained devices may not be easily exposable; constrained hardware makes it very difficult to allow security [78].

Two containers, namely S_a and S_b that need to reach each other, deployed in two different networks, require a secure tunnel to enable the communication. Using tunneling is possible to create an overlay across multiple nodes that enable S_a to reach S_b. Nevertheless, the reachability of the device hosting the services may be limited, the addresses are dynamic, and multiple nodes may share the same public network address. Thus, a network packet sent from S_a with destination S_b, traveling across the overlay, needs to cross multiple borders and must be subject to multiple address translation operations before reaching the destination. Solutions like [79] propose an SDN-based forwarding technique that uses a gateway switch to interconnect different regions. In contrast, solutions like [80] use site-to-site ad-hoc VPNs to create an overlay across the nodes.

2.3.3 Availability

Thanks to the centralized approach used in cloud datacenters, different High Availability (HA) enabled systems may easily overcome the failure of a node seamlessly [81]. Unfortunately, at the edge, fault tolerance may have to deal with more frequent failures both in terms of networks and hardware. Moreover, if a user expects to resolve a request with a specific IP address, but the node hosting the service went offline, and thus the service exposed behind that address is unavailable, the network request fails. SDN approaches may enable redirection of the traffic to another node containing a service replica seamlessly, but still not always applicable due to the configurations involved and other limiting factors. Then, along with a higher level of service replication needed to maintain the QoS, the traffic must be routed accordingly among the available instances; load balancing may be one of the best approaches to overcome the availability problem in edge environments.

2.4 Load Balancing

Since scaling services and servers is the key to achieve availability, high QoS, and handle large workloads, it is essential to balance the incoming requests among all the available instances.

Balancing happens in two different levels, network and applicative. Balancing at the network layer means choosing a server address across the ones available. Balancing at the applicative layer means selecting a service instance across the ones available. Since it is possible to move the applicative decision to the networking one, the balancing strategies usually match. In fact, each service has a network address, and the target service decision is all about the final destination address. Virtualization and overlay network technologies make services behave as single entities residing in private servers with unique addresses.

The balancing process always involves an entity, the balancer, that decides the final destination. The balancing technique defines how the traffic is handled, while the balancing strategy describes how the destination is chosen.

A well-known balancing technique is the DNS-based balancing [82]. The client rarely uses

IP addresses, but most commonly, domain names identify the destination. Hostnames are resolved by Domain Name Servers (DNS). The client queries the DNS to get an address that represents the final destination. It is possible to configure multiple addresses for a single domain name such that the DNS can balance the traffic by resolving the requests across them. The resolved address expires after a time-to-live (TTL), and the client should attempt another query for that address. This balancing strategy works pretty well for stable entities, with addresses that do not change dynamically. Unfortunately, services are very dynamic; they can scale, move, replicate. This means that the DNS must always be kept up to date. But even lowering the query TTL and keeping the DNS up to date as much as possible, there is no guarantee that the client does not get a failure; a route may turn unavailable before the TTL expiration, making the client unable to contact the final destination. Moreover, decreasing the TTL time creates congestion at the DNS server, and unfortunately there is no guarantee that the client respects entirely the query expiration time issued. The behaviors at the client-side may be unpredictable, and any solution that relies on the client-side trustworthiness shall be avoided.

Another approach to the load balancing problem is constituted by the use of a dedicated component that handles the traffic transparently. The ingress component collects all the traffic and dispatches it to the final destination [83]. This component is accessible with a unique address; thus, the communication is transparent from the client's point of view. These devices may also be replicated to avoid a single point of failure, standard replication techniques involve active-active or active-standby scenarios [84].

Modern load balancing techniques use both proxies at L4, and L7 [85] of the OSI model. The former typically uses only information available at layer 4, thus IP addresses and port numbers. The latter works with layer 7 packets and thus with applicative protocols. On the one hand, an L4 proxy supports all the protocols based on top of layer 4, but it needs to establish a new connection for each packet that comes through. On the other hand, L7 proxies are protocol-specific but handle the traffic from the applicative perspective, making them usually more lightweight. Moreover, this latter proxying schema can make smart balancing decisions based on the protocol-specific packet headers.

Proxies can be placed in multiple locations, inside the nodes, as a service's sidecar, at the entry point of a network, or even to an external network. The choice of the proxy location is merely platform-specific.

Other balancing approaches may even comprise HTTP redirection. In this case, a target public Web Server that receives all the incoming traffic decides the most suitable destination internally and finally sends back to the client the redirection request to the actual destination [83].

2.5 Orchestration Platforms, Frameworks and libraries

Whether they are in a cloud or edge environment, developers use orchestration platforms to glue together transparently their application with the hardware to satisfy the business requirements. Once the virtualized infrastructure is ready, it is common practice to install on top of it another layer that allows managing the applicative software with a reduced operational cost. The goal of this layer is to abstract the single compute nodes to a unique environment where services reside.

The common idea behind each one of these platforms is to abstract the hardware maintenance and let the developer take care only of the business requirements. These frameworks and libraries also enable services to communicate, scale, and migrate autonomously. Given the SLA (Service Level Agreement), then the developer only writes the code and lets the platform handle it.

This section presents a review of some of the most common frameworks, platforms, and libraries used in edge/cloud environments, focusing particularly on the way they carry out service-to-service communication from an applicative level.

2.5.1 Kubernetes

As stated in the official documentation [86], Kubernetes is an open-source container orchestration engine that can automate the deployments, scaling, and management operations for containerized applications. The Kubernetes' smallest units of work are the pods. From a logical point of view, a pod is a group of containers with shared storage and network resources. Pods are potentially deployed across multiple nodes, and together they represent a service issued by the developer.

A Kubernetes cluster is composed of at least one worker node that runs the pods and a set of nodes that constitutes the control plane that manages the workers and the deployed pods. The control plane has a global perspective of the activities within the cluster and can perform high-level decisions about the state of the workers and the deployment. This component performs the scaling, migration, undeployment operations for a pod and as well controls the status of the worker nodes. Through the API, the developer is able to communicate with the control plane and orchestrate the pods within the system. Kubernetes provides as well also several workload resources [87] that enable automation in the management of the lifecycle of a pod. For instance, *StatefulSets* manages the scaling and deployment of a set of pods providing guarantees about their ordering and uniqueness. This workload is helpful to support stateful applications and be able to address them uniquely.

In Kubernetes, the way to expose a set of pods as a network-accessible application is to declare them as a service. Even though each pod has its own address, if an application has several available instances, it must be possible to perform load balancing across them with a single virtual global address. The developer then uses the ClusterIPs that uniquely address a set of pods. The platform takes care of the underlying balancing and routing operations.

Figure 2.8 shows how the Kube-proxy enforces the service abstraction balancing the traffic across the two pods in the cluster. Thanks to the iptable, the proxy component performs all the necessary translations needed to route the traffic across the instances. The platform supports many different algorithms like rr (round-robin), lc (least connection), sed (shortest expected delay), and others. Kubernetes also provides an internal DNS name resolution to enable developers to address services not by address but by name.

The use of network plug-ins enables physical interconnections between servers. A network

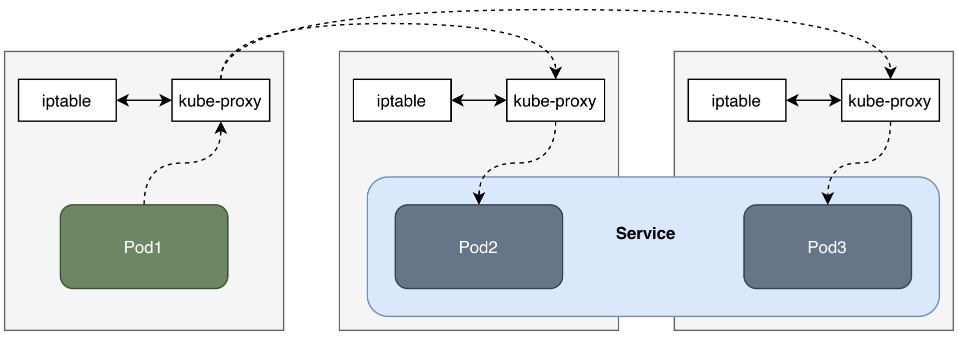


Figure 2.8: Services and proxy in Kubernetes

plug-in in Kubernetes must implement the Container Network Interface (CNI) [88] that "consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers". CNI plug-ins are able to create bridged or tunneled networks spanning across multiple nodes to enable the physical packet exchange. Kubernetes also comes with its Kubenet plugin that creates a locally bridged network. Some examples of well known network CNI plug-in are: Flannel [89], Calico [90] and ACI [91].

The downsides of the Kubernetes approach for the edge are related to its cloud-based nature. In particular, the data model of the Kubernetes always supposes strong consistency of the data across the nodes [92], which is exponentially complex to achieve with a swarm of distributed edge workers. In particular, constrained devices may not even be able to memorize all the information regarding the nodes participating in the cluster. Balancing the traffic may become unfeasible with outdated information and restricted hardware capabilities. Moreover, there is no offline support; the entire cluster goes down if the control plane fails. Even if there is the possibility to deploy the control plane in High Availability mode, with multiple instances across multiple devices, in case of network failure, if the workers are not able to reach those devices, the cluster goes down. This scenario is not supposed to happen in the cloud, or at least not frequently, but it is expected in the Edge. It is important to notice that Kubernetes is also a single clustered approach that needs some extensions in order to be deployed across a federated environment. Lastly, the balancing decisions in this platform are statically configurable. This gives a lower degree of flexibility in the edge. With a heterogeneous set of devices, a developer may want to dynamically change the way a service call is balanced within the infrastructure. The developer must be able to specify packet-wise how the request must be handled.

Overall, the orchestration and networking schema of Kubernetes highly reflects the idea of the underlying topologies shown in §2.1.1. Which makes it an excellent orchestration platform for the cloud but showcases significant limitations for edge deployments.

2.5.2 Kubedge

KubeEdge [93] is an open-source system that allows computation at the edge. Given the cloud nature of Kubernetes, this project tries to provide an approach that better supports

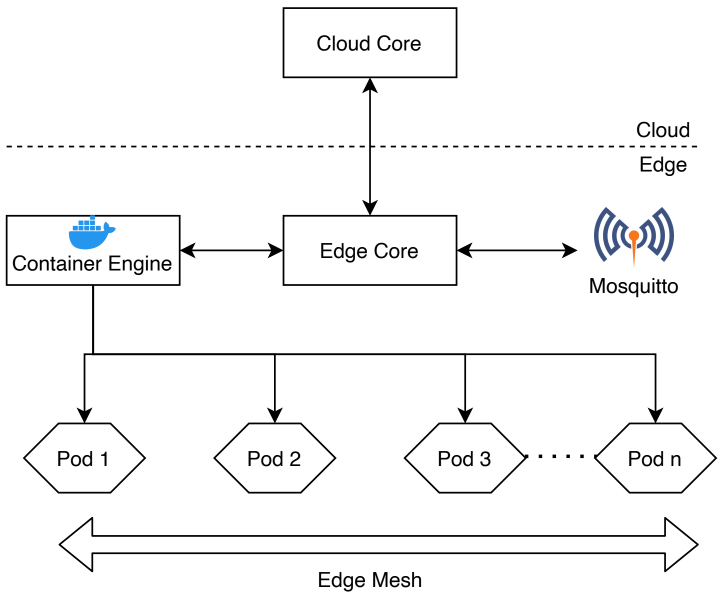


Figure 2.9: KubeEdge architecture

the computing nodes residing closer to the users, allowing developers to deploy pods and services to small, constrained devices. Because of the high maturity of Kubernetes, this approach extends this well-known and consolidated platform to the very edge of the network, enabling seamless integration of cloud applications and remotely distributed ones.

Figure 2.9 shows the principal components of Kubedge and the way they interact. The Cloud Core component is the direct interface with the K8s API server and integrates the edge components and the cloud ones. The communication between the Edge and the Cloud services is carried out with the KubeBus that creates a reliable L4 proxied tunnel [80]. The Edge-Core component is a lightweight agent that runs inside each edge node registered into the platforms. This component provides all the deployment and management operations for the pods, integrates with the container engine, and enables networking capabilities. Pods can communicate across multiple worker nodes through the Edge Mesh, which represents the overlay network abstraction of KubeEdge. The traffic is proxied through the Edge-Core component that translates each packet, enabling the Kubernetes "service" abstraction even at the Edge. The mesh is implemented as a distributed proxied network. Nodes can even reside on private networks and be able to communicate using a VPN that passes through the KubeBus of the Cloud Core components. Finally, this architecture even proposes an MQTT broker used as an event bus.

Despite its experimental status, this approach really targets the edge proposing a real solution suitable to the challenges of such an environment. Unfortunately, KubeEdge currently still needs a Kubernetes cluster to work and thus brings with it all the consistency requirements for the synchronization of the information within edge deployed services and cloud ones. Moreover, this setup still remarks a single cluster infrastructure without the possibility of creating a federated deployment out-of-the-box. In those cases, multi-cluster communication must be enabled with extra plug-ins.

All in all, this platform represents a promising approach to enable a cloud-to-edge continuum that does not impose further learning curves for Kubernetes experienced developers and creates a seamless unified system for hybrid deployments.

2.5.3 Skupper

Skupper [94] is a service that enables L7 Virtual Application Networks. Using Skupper, it is possible to unify federated environments spanning multiple application sites with a virtual network composed of L7 routers.

The routers form a backbone that interconnects all the clusters. For single cluster platforms like Kubernetes, it is possible to use Skupper to extend the communication across multiple namespaces. An L7 router must be installed inside each cluster's namespace, and the packets can flow through it. After a service deployment, Skupper creates a proxy endpoint inside each namespace. The traffic is then routed and balanced through it to make the service available on all the sites. The connection is tunneled across the nodes with a TLS encrypted and authenticated channel. The L7 routers use a cost function to determine the load factor of each site and spread the traffic evenly to avoid failures and congestions. It is even possible to configure resource preferences in case of peaks or normal traffic levels.

The approach proposed by the Skupper project is secure by design and easily brings connectivity in geographically distributed environments. This project, unfortunately, requires each node to maintain a complete view of the platform and the released services. After each deployment, the information is propagated, and consistency must be reached in all the clusters. As soon as the number of clusters and services is limited, the problem is trivial. Still, we envision the lack of scalability for highly distributed platforms, especially if the devices involved are constrained. In fact, as reported in the official documentation, each Skupper instance is always aware of every service that has been exposed to the Skupper network [95]. Anyway, an edge node may not have enough resources to handle the knowledge of the entire network status. Moreover, despite its simplicity, creating multiple clusters interconnected with Skupper still requires some initial configuration on top of the already pre-existent orchestration framework.

This project represents a fascinating approach to connect securely and efficiently different clusters without the need for a VPN. Spanning the connectivity across a federated environment is a crucial feature for the edge as well, but it must come together with flexibility and the support for the heterogeneity.

2.5.4 k3s

K3s [96] is a lightweight distribution of Kubernetes. This Project bundles most of the Kubernetes dependencies in a single lightweight executable that can be deployed even to the small edge and IoT nodes. K3s is the perfect solution to bring Kubernetes services on small devices effortless.

The control plane is the *k3s server* which can be deployed on a machine that takes the responsibility of managing the cluster. The worker nodes are instantiated with the *k3s agent* that bundles together the CNI plugin (Flannel by default), the Kube proxy, the tunnel proxy, and the container engine interface. As a spin-off of the Kubernetes project, K3s provides the same networking approach for container to container communication, using the ClusterIP abstraction. After a service deployment is performed, the Load Balancer creates a proxy pod using a DaemonSet on all the nodes. This pod is the proxy for all the service's traffic. The default implementation for physical traffic handling uses VXLAN. Still, it is also possible to configure WireGuard [97] as an additional VPN component to enable traffic through different networks.

K3s provides a lightweight version of Kubernetes suitable for the Edge, with a meager impact on the resources, but still with a high availability setup. This project still brings some of the downsides of the Kubernetes networking scheme. In fact, in K3s, it is still needed to add the overhead of a VPN to handle the traffic spanning across different networks. Moreover, it has a single clustered design, and it does not provide flexibility for the dynamic traffic balancing decisions needed at the Edge.

2.5.5 IoFog

The IoFog Project [98] by Eclipse Foundation enables the creation of a distributed Edge Compute Network (ECN), allowing developers to deploy microservices on it dynamically. One of the main goals of this project is to enable developers to deploy applications in the edge the way they do it in the cloud to minimize the learning curves. Simply running the IoFog agent on each node, it is possible to manage the deployed services. Then, a controller service manages a set of devices. Complex configurations may even require more than one controller. This project also includes a Skupper integration for networking across distributed nodes within different networks.

In IoFog, communication is enabled by the Router component. By default, there is a router for each agent and controller but is even possible to create custom topologies. For each exposed microservice, a proxy component is created to translate the services HTTP requests to the AMQP protocol used by the routers. Microservice interactions take place through the ioMessages. The developer can use a custom SDK to build services compliant with the IoFog messaging protocol and avoid dealing manually with complex service discoveries mechanisms.

The approach proposed by this project consists of a straightforward but custom procedure for services traffic handling. Despite the easy-to-use SDK offered, the developers must change their services' code to deploy them inside the infrastructure. The communication is constrained according to the methods offered by the platform to the developer and can't be extended. Furthermore, there is no dynamic customization in the way the traffic can be balanced or routed.

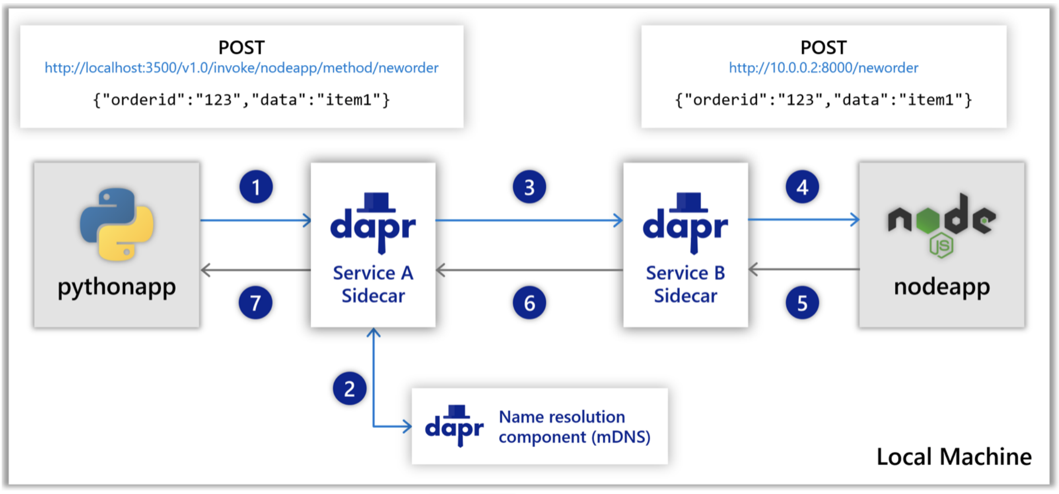

POST
http://localhost:3500/v1.0/invoke/nodeapp/method/neworder
{"orderid":"123","data":"item1"}
POST
http://10.0.0.2:8000/neworder
{"orderid":"123","data":"item1"}
pythonapp
dapr
Service A Sidecar
Service B Sidecar
nodeapp
1
2
3
4
5
6
7
Name resolution component (mDNS)
Local Machine

Figure 2.10: Dapr communication example

2.5.6 Dapr

Dapr [99] is a runtime that helps in building stateful and stateless applications enabling flexible microservice interactions. The proposed architecture foresees a sidecar deployed together with each service. The service can then use the Dapr API to communicate with the sidecar, which handles the message dispatching and the storage management.

This project can be deployed with Kubernetes as well. In that case, each pod contains the microservice container together with the Dapr sidecar container, both interacting through HTTP or gRPC protocols. Among the Kubernetes cluster also the Dapr building block component pods must be deployed. They manage the state, resolve the routes, inject the sidecar and share the information across multiple workers.

Figure 2.10, from the official documentation [100], shows an example of service communication with Dapr. In order to enable interactions between a Python and a Nodejs application, the sidecars take care of the entire message exchange procedure. The Python service simply invokes the sidecar's API method with a POST request. The Service A sidecar resolves the request using the internal mDNS and deciding the target sidecar as the destination. The destination sidecar then simply forwards the request to the Nodejs app, which will answer back later on.

In the case of communication distributed across multiple nodes, the sidecars use the orchestration platform's underlying topology abstractions. From the developer's point of view, a set of microservices deployed with Dapr forms a self-managed fully-connected mesh.

Dapr is a flexible solution to enable services communication through the pre-existing infrastructure. Moreover, it also allows state management, and it integrates as well with a pre-existing Kubernetes cluster. Unfortunately, even with this approach, the developer must adapt the code of the microservices before the deployment. Moreover, even if Dapr has proven to be very scalable, each service must bring a sidecar container, which drains further resources in the machine. The sidecar's impact on cloud environments may be negligible, but on constrained devices may not. Dapr also needs a pre-existing orchestration framework and adds on top of it the overhead of the state and communication management.

2.5.7 EdgeIO

EdgeIO [101] is a lightweight orchestration framework for edge computing. With its multi-layer architecture, this framework allows developers to deploy workloads to nodes in a multi-cluster federated environment. Worker nodes can receive containerized applications as well as unikernels [102]. As shown in fig. 2.11 EdgeIO is composed of a Root Orchestrator and one or more than one Cluster Orchestrator, one for each cluster. Then, each cluster contains at least one worker node, which is the place where applications are deployed and runs.

The system and the cluster manager components are responsible respectively for the management of the clusters and the workers. At the top layer of this hierarchy, the clusters are considered a large pool of resources. The Cluster Orchestrator completely abstracts the single worker nodes, showcasing only the aggregated collection of resources and services. The scheduler components are instead responsible for the service placement. When the developer starts a deployment, the application must be scheduled first across the clusters and then across the nodes. The Node Engine component is what enables a node to accept workloads and manage the execution environment.

In this framework, applications may be scaled up, down, and migrated across nodes and clusters, even enabling the possibility of a cloud-to-edge continuum seamlessly. Furthermore, by design, workers can even be deployed within networks featuring limited external visibility on scenarios with restricted control over the fabric.

In order to avoid high learning curves for the developers, microservices do not require any additional code to be deployed into this framework. Providing a deployment descriptor containing the image name along with the service level agreement (SLA) for the application, the framework will schedule and deploy the requested instances in the worker nodes.

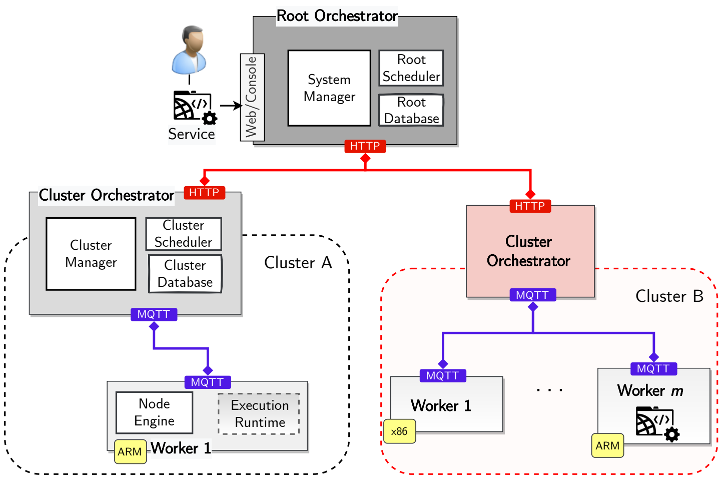


Figure 2.11: EdgeIO architecture without any networking component

Because of the research and experimental nature of this project, its flexibility, and its compatibility with the edge requirements, it is the reference architecture for the microservice networking model presented in this thesis. Moreover, since the current proposed EdgeIO's implementation does not provide the possibility to enable interactions between the deployed services, this thesis installs the proposed components into the platform to create a real-world scenario for a sound evaluation. The result is a unified mechanism that enables fault-tolerant service balancing across multiple nodes, supporting several virtualization technologies while being versatile in any network condition.

3 EdgeIO Networking

This chapter introduces a solution to enable flexible interactions within services deployed in a heterogeneous edge infrastructure. Due to the promises and the research nature of EdgeIO, this solution remarks its original three-tier architecture and integrates the new components to the existing ones. Anyway, the proposed solution by design can be seen as a plug-in that can be used in other contexts and with other platforms as well. Starting with a sequence of requirements (§3.1) that the platform must achieve, the chapter continues with the big-picture (§3.2) of the architectural components introduced and a detailed description for each one of them (§3.3 to §3.5). The second part of the chapter describes the flow that services follow to be deployed and to be able to communicate with other services (§3.6). The chapter continues with an experimental proposal for extending this architecture to enable even external traffic to reach services deployed internally (§3.7). To conclude, it is presented how this architecture has been implemented in practice and the current state of the work (§3.8).

3.1 Requirements

The proposed networking components are based on the hierarchical structure of the current EdgeIO implementation and integrate some extra components to enable the networking capabilities. The design of such components was driven by a strict requirement list that gathers together the main principles that the networking in such an environment must envision. The requirements described in this section have been conceived after an exhaustive analysis of the edge's potential limitations and opportunities. The architectural schema proposed in 3.2 is the direct consequence of these requirements.

R1 *Modularity* The Edge evolves rapidly, and with that, even the approaches can be different over time. Thus, it is crucial to design the components that will be part of the infrastructure modularly and loosely coupled. It must be possible to update or even change the elements without disrupting the design. The networking stack must be designed as a plug-in that can be easily be replaced.

R2 *Fault tolerance* By design, the architecture must be fault-tolerant. This goal must be achieved on two sides, replication, and independence. The root orchestrator and cluster orchestrator components should always be available. Particularly the networking components can't afford failures since they enable communication across different services. To achieve high availability and thus improve fault tolerance, the primary strategy is replication. The proposed components must be scalable and can be replicated. Having multiple instances of the same component drastically reduces the possibility of global failure and increases the request handling amount. On the other side, it can still happen that the entire root or cluster orchestrator machines are unavailable for many reasons, including network failure, hardware failure, and so on. In the Edge, it is not possible to make strong assumptions regarding the underlying infrastructure.

For this reason, the architecture must enforce independence between the components. A worker must enable service-to-service communication independently of the overall infrastructure status. Even if the cluster orchestrator is unavailable, the worker must be able to the best of his effort to enable services to talk to one another using the locally available information. A worker must contact the cluster for new routes only if necessary and must keep enough information to eventually route the traffic anyway, even if the rest of the architecture is not responding.

R3 *Transparency* The connectivity must take place transparently. The developers do not have to change the code while deploying a container in EdgeIO; the communication components must handle the traffic without requiring sidecars or complex API usage. While supporting the vast majority of protocols, the networking components must ensure high QoS during the traffic routing operations. An application must run seamlessly in edge, fog, or cloud environments, without refactoring. Moreover, external clients must be able to access the services inside the platform using standard networking techniques. Expecting a client to behave differently is a very strong assumption that this work can't afford. Thus, it is assumed that the client-side application must use standard DNS resolution approaches and only standard protocols for the connectivity. It is also acceptable to slightly relax the latter requirement by restricting the client to using a sub-set of the most common protocols – like HTTP/HTTPS, gRPC, etc.

R4 *Low overhead* While designing the architecture, it must be considered that the hardware capabilities at the edge are constrained. If the components use a big share of the resources themselves, there is no more room for user applications. The goal here is to find a balance between the resource consumption on the worker-side or at the cluster/root-side. The more is delegated to the workers, the lesser resources will be available for the applications. The more we charge to the cluster orchestrator, the lesser is going to be the number of nodes supported by the platform. The more we delegate to the root orchestrator, the lesser will be the amount of cluster that it will be able to handle. The trade-off of the hierarchical structure of EdgeIO resides in finding the perfect balance in the number of operations. It must be considered as well that the more the networking relies on components outside the worker's network boundaries, the higher is going to be the latency introduced.

R5 *Low latency* By design, the networking components must grant low latencies in the operations performed. Since the beginning, it is clear that the proxying decisions will cause latency in the communication. Having high latencies almost nullifies the benefits of load balancing. Meanwhile, to achieve the perfect balancing decision, all the instances must be considered. Even here, a trade-off is in place. An excellent recipient estimation must be performed while avoiding spending too many resources and too much time on the decision.

R6 *Extensibility* The proposed network plug-in must envision extensibility by design. It must be possible to easily extend the behavior of the network and as well the routing decisions. For instance, given a set of the initially supported routing algorithms, it must be easy to add new ones.

R7 *Infrastructure independency* Edge computing usually involves a large set of devices with heterogeneous software, hardware, and network capabilities. For instance, each device can have different Instruction Set Architecture (ISA), like ARM, ARM64, x86, etc. Moreover, a device can be located behind NAT, have firewall restrictions, limited or no control over the network infrastructure. The goal here is to build the network component so that it can be portable across many distinct kinds of devices in many different locations and still make them able to communicate. The only relaxation possible for this principle can be the OS. It is, in fact, not confining to suppose Linux as the target OS since it represents the majority share in the Edge and IoT devices panorama [103].

R8 *Isolation* Each application can be composed of many services, which must be mutually isolated. Thus, each service must be isolated from the other services of different applications. Therefore, each app can have its own namespace, and every service within the app must also be able to declare its own sub-namespace. Finally, a service can have multiple independent instances that must be individually addressable.

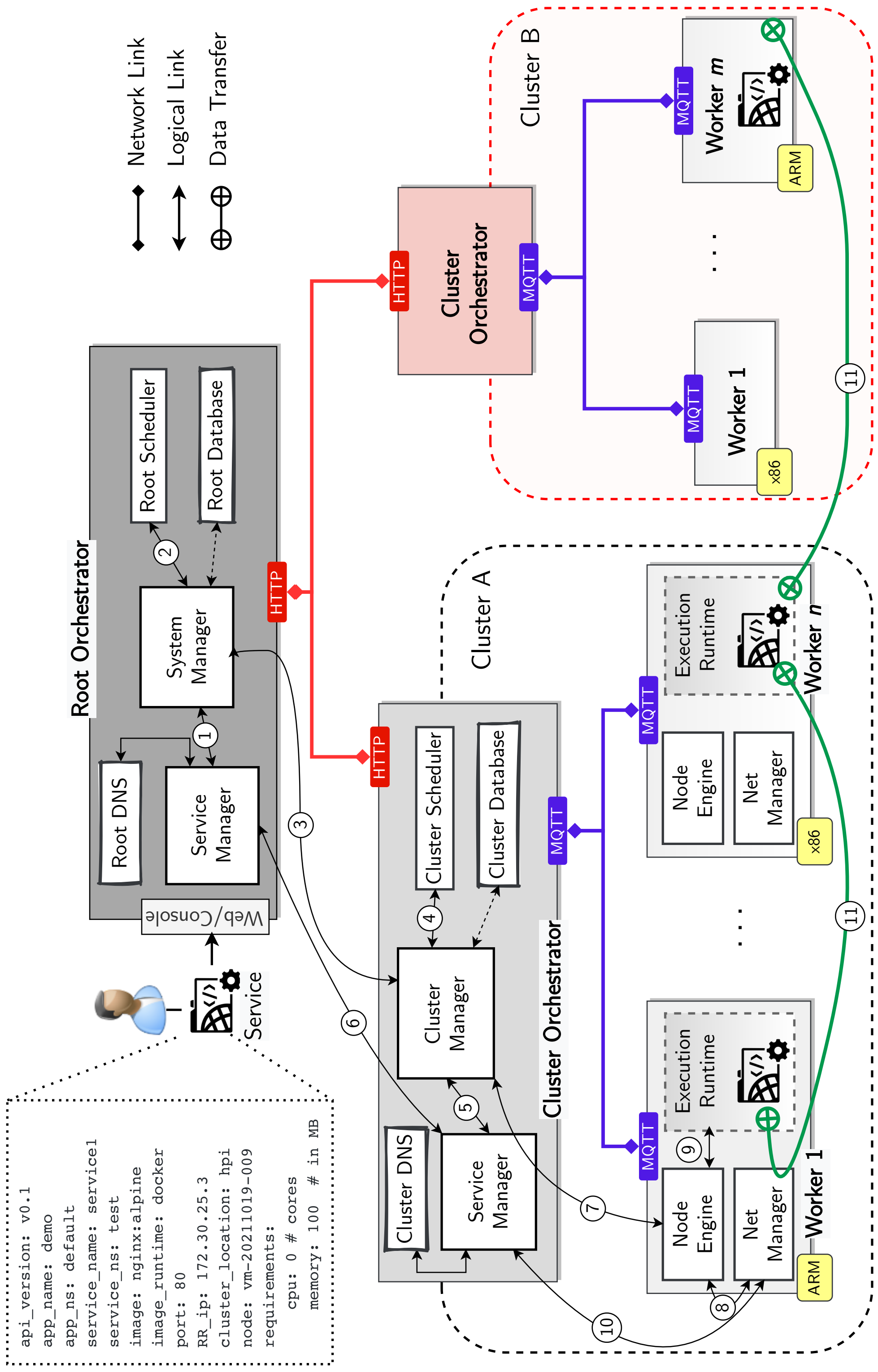
R9 *Flexibility* The Edge is not just a cloud closer to the user. Due to the heterogeneity and the geographical location of the instances, routing must be flexible and provide as well intelligent decisions. It must allow the user to choose how to select an instance that satisfies the requirements to accomplish the expected QoS.

3.2 System Architecture

Starting from the base architecture of EdgeIO, shown in §2.5.7, this section now explains the modification proposed by the schema in fig. 3.1.

On top of the root orchestrator, there are now two new components, the DNS (§3.7.1) and the Service Manager (§3.3). The former enables service discovery from clients residing outside the platform. The Service Manager instead represents the point of contact between services living in different clusters. When a service performs a service discovery operation, the root orchestrator's role is to enable communication within clusters informing where the requested service instances are deployed.

Like in the original architecture proposal, the cluster orchestrator mirrors the root orchestrator components. Thus, there are a DNS and a Service manager. The former is now an internal DNS of the cluster. The top-level DNS of the root orchestrator resolves each query with the address of the cluster's DNS. The latter is a cluster-level Service manager. The service names, IP addresses, and instance locations are resolved inside the cluster's service manager to grant



```
api_version: v0.1
app_name: demo
app_ns: default
service_name: service1
service_ns: test
image: nginx:alpine
image_runtime: docker
port: 80
RR_ip: 172.30.25.3
cluster_location: hpi
node: vm-20211019-009
requirements:
    cpu: 0 # cores
    memory: 100  # in MB
```

Figure 3.1: EdgeIO architecture with the new networking components

confidentiality. Therefore, the difference between the root and the cluster components resides in the domain and the isolation level. At the root level, the components see a cluster as a unique device, abstracting the actual underlying workers' infrastructure. The cluster-level components deal instead with the existing set of worker nodes.

This architecture proposal wants to avoid overloading the cluster and delegate as much workload as possible to the nodes. For this reason, the entire communication management is delegated to the nodes. The service managers are only used as a centralized brain to resolve the service discovery query lazily issued by the worker nodes. This approach avoids the overhead of a consensus or synchronization mechanism between distributed entities while enabling each worker to fetch the service data needed for the communication with a just-in-time strategy. Each worker is equipped with a set of internal components that enable lazy service discovery and autonomous balancing capabilities to enable communication between the deployed services regardless of the virtualization technology.

3.3 Service Managers

Within the general edge computing infrastructure and the requirements discussed above, it is essential to architect the components so that the information needed to perform routing decisions and isolate the services is structured, easy to access, and maintains the hierarchy. Furthermore, it is also necessary to manage the nodes so that it is easy to route the traffic and scale the network. The Service Managers components address precisely these problems. The Service Manager placed in the root orchestrator is the component with a general view of all the services deployed inside the systems. If an app has been correctly deployed in EdgeIO, this component knows it. The Service Manager placed in the cluster orchestrator has instead a local view of the services deployed within the cluster boundaries.

Upon the deployment, each service is decorated with a set of information needed by the platform to enable the communication abstractions. The information maintained for each service is shown in table 3.1.

Table 3.1: Information maintained by the Service Manager components

ID	App	App ns	Service	Service ns	ServiceIPs	InstanceIPs
<int>	<str>	<str>	<str>	<str>	<sip_list>	<iip_list>

For each quartet App, App namespace, Service and Service namespace, a list of Instance IP and Service IP is maintained, namely *sip* and *iip* list.

The *sip* list, shown in table 3.2, contains the set of all the supported ServiceIPs representing that specific service. A ServiceIP represents all the service's instances and enforces a particular routing rule while redirecting the traffic to that address. The addresses in this list are described later on in §3.6.1. During the deployment, using a *deployment descriptor*, the developer can manually choose these addresses in the pre-established range. For each ServiceIP type not specified in the deployment descriptor, a new one is generated and assigned by the Service Manager. Each service must have one ServiceIP for each supported routing policy.

Table 3.2: sip_list format

ID	Type	Address
<int>	<RoundRobin/Closest/...>	<x.y.z.w>

The *iip* list (table 3.3) contains, for each instance of a service, the *instance address*, the *namespace address*, the *worker address* and the *tun port* of the tunnel. The *instance address* is the public address that a service can use to access a specific instance of another service belonging to the same namespace, ignoring the routing policy completely. This is particularly useful for state-full applications. The *namespace address* is an address that is used internally by EdgeIO to contact the instance through the overlay network. Each node has an internal subnetwork, and each service is internally addressed with the IPs assigned from this subnetwork. This address is not going to be used by the services. Instead, it is the address used by the internal proxy component after it decides which instance will be the recipient for a packet. The *worker address* is the public address of a worker node. The tunnel component uses this address to create the overlay across the nodes. Note that it is not required for each worker node to have a unique public address. It is quite common to have multiple nodes behind NAT in this environment. Each node exposes one *tun port* and using the couple address+port is possible to contact specifically the target worker. From a security point of view is possible to protect the nodes granting the traffic to the tun port of a node only to a set of well-known addresses representing other workers, thus avoiding public internet traffic to reach the exposed port.

Table 3.3: iip_list format

ID	Instance address	namespace address	worker address	tun port
<int>	<x.y.z.w>	<a.b.c.d>	<e.f.g.h>	<int>

Each service in EdgeIO is part of a hierarchy of namespaces that enforce isolation among applications and services. As shown in fig. 3.2, a top-level namespace contains several applications. Each application can have multiple services inside multiple namespaces. The Service Manager stores the namespace information for each service from a global perspective. Even if two different instances of the same service belonging to the same namespace are deployed across multiple clusters, logically, they must be able to communicate. The *root service manager* shares with a *cluster service manager* the routes of a service spanning across multiple clusters, only **if** and **when** they are required. In fact, the routes are reclaimed lazily when a service needs to contact another one. The route retrieval operation is called *table query* and is triggered by the worker node §3.6.6.

As mentioned above, the namespaces grant isolation across applications. Nevertheless, the developer must have the possibility to extend the communication across multiple App

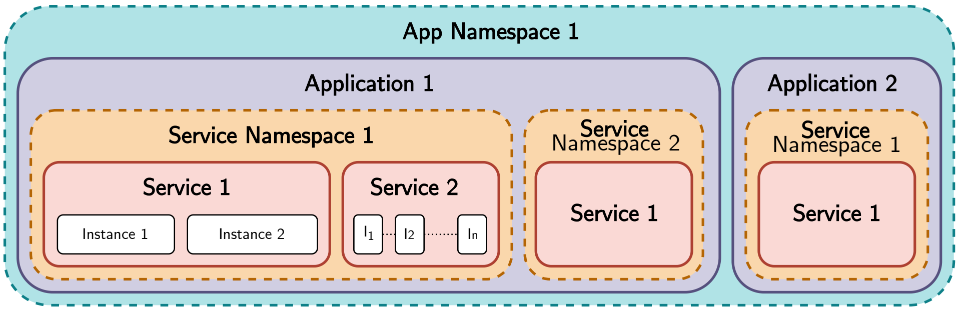


Figure 3.2: namespaces hierarchy

NS and Service NS. The Service Manager component enables this possibility of exposing an API to the System Manager to manage the internal firewall rules. This allows a developer to extend the reachability of his service to another *namespace*. When the Service Manager has to resolve a route for a table query, it limits the routes to those authorized by the internal firewall o the ones belonging to the same namespace. The Service Manager interacts with the System Manager as if it is a plug-in. Then, it is possible to contact the System Manager directly with his exposed REST API or use the web GUI of EdgeIO to set up the permissions. When it comes to plugging a new worker node into EdgeIO, the platform must generate a new subnetwork, extending the overlay that spans across several nodes to it as well. The internal *namespace address* shown in table 3.3 resides exactly in this node's private network. The internals of the EdgeIO's subnets mechanism is detailed in §3.4.

To enforce **R1** (Modularity), the root and cluster service managers are connected using a dedicated API that is used to communicate with the System manager as well. Thanks to the containerized nature, these components respect **R2** (Fault tolerance) and **R7** (Infrastructure independency) because they can be easily replicated and scaled regardless of the hypervisor implementation. **R4** (Low overhead) is respected by design, thanks to the lazy route resolution scheme, while **R5** (Low latency) is satisfied thanks to the cluster-level caching. **R6** (Extensibility) **R3** (Transparency) and **R9** (Flexibility) are satisfied by the design choice of using multiple ServiceIP for each service. In fact, this design grants the possibility of extending the platform with new routing policies and the services to communicate using standard mechanisms but with a new flexible semantic. The requirement **R8** (Isolation) is satisfied thanks to the namespace hierarchy imposed by the platform.

3.4 Subnetworks

An overlay that spans seamlessly across the platform is only possible if each node has an internal sub-network that can be used to allocate an address for each newly deployed service. When a new node is attached to EdgeIO, a new subnetwork from the original addressing space is generated. All the services belonging to that node will have private *namespace* addresses belonging to that subnetwork.

The current implementation uses the subnetwork cut described in fig. 3.3. The network

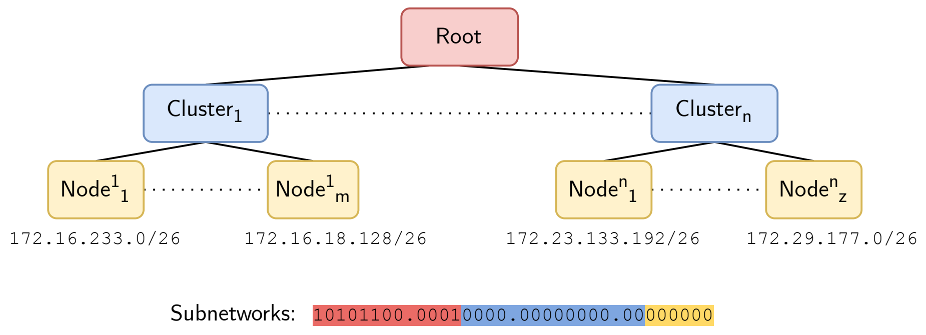


Figure 3.3: Address space used for the subnets

`172.16.0.0/12` represents the entire EdgeIO platform (fig. 3.3 - red). From this base address each cluster contains subnetworks with a netmask of 26 bits that are assigned to the nodes. (fig. 3.3 - blue). Each worker can then assign namespace ip addresses using the last 6 bits of the address (fig. 3.3 - yellow). The address `172.30.0.0/16` is reserved to the ServiceIPs.

This network cut enables EdgeIO to have up to ≈ 15.360 worker nodes. Each worker can instantiate ≈ 62 containers, considering the address reserved internally for the networking components. These workers and container addressing limitations will be overcome after the prototyping phase of these networking components, switching to the `10.0.0.0/8` addressing space. Note that, thanks to the fact that even here, the service manager component is respecting **R1** (Modularity) and **R6** (Extensibility), changing the addressing space does not involve any modification to other components.

When a node joins EdgeIO, an `hello` message is sent, and a new registration takes place. Figure 3.4 shows how the subnetwork request is carried out during the initial node's registration. When a worker node registers to its cluster, contacting the cluster manager via the WebSocket-based handshake, it triggers the assignment of a new subnetwork. With the "registration complete" message, the node must receive as well a new IP address in the form `a.b.c.d/x` that represents the new network. The *cluster manager* asks such address to the *cluster service manager*. This component can contain a pool of subnetworks ready to be assigned to new workers. If it's that the case, the *service manager* returns a new subnet address to the *cluster manager* that can then complete the node registration. If the *cluster service manager* does not have any subnetwork left, it asks a new one to the *root service manager*. In this architecture, the root level has global knowledge of all the subnetworks that have been assigned and can easily generate ones with no overlap.

3.5 Net Manager

In this section, the components that enable the networking inside a worker node are detailed. In the proposed architecture, each node bears with it a very high amount of responsibilities. The entire communication is kept as decentralized as possible, and the worker nodes try to arrange themselves to enable seamless interactions. This avoids the developers the need to

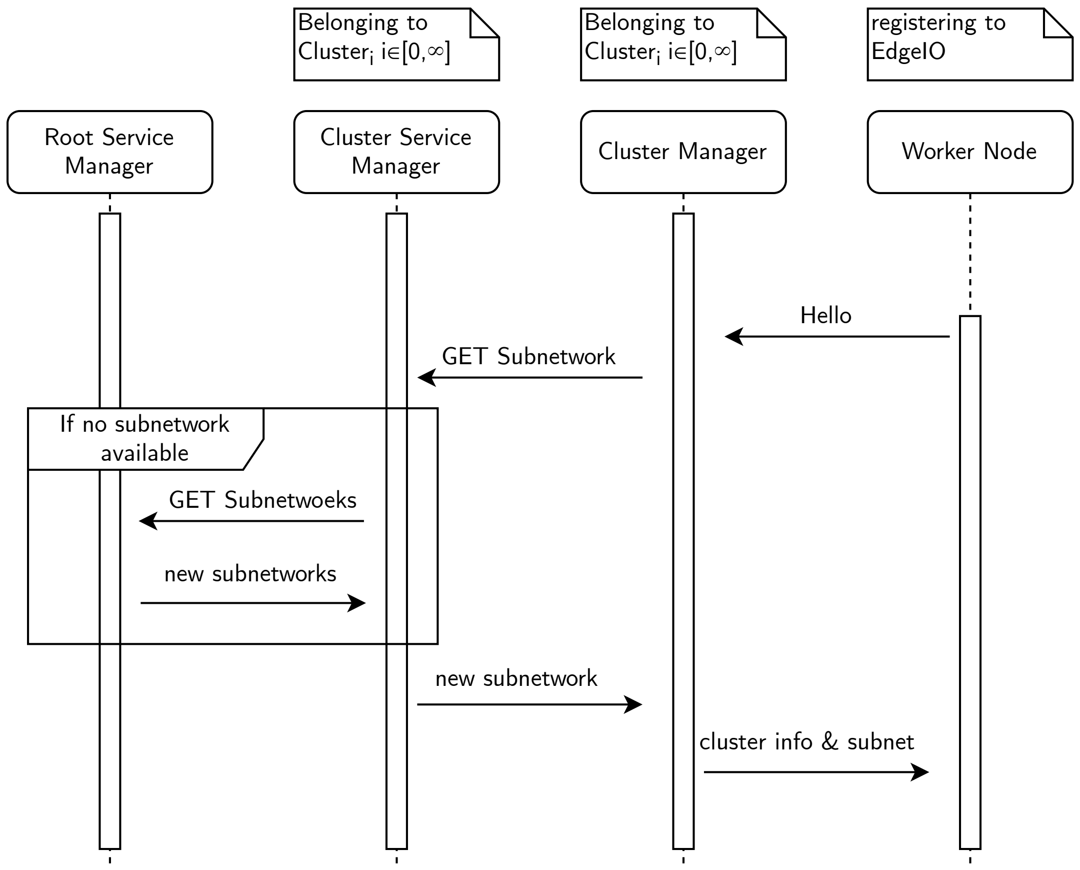


Figure 3.4: Subnet request sequence diagram

adapt the application's code. Moreover, the communication model enforced has to be efficient and fault-tolerant. Small nodes placed in local networks can contribute as worker nodes but can't really out of the box satisfy the requirements mentioned above. The platform must abstract the computation capabilities of these nodes, connect them to an overlay network, and handle eventual failures both of the network, of the node, and the services deployed inside them. It is essential to provide a fallback strategy for all the things that can go wrong during the services' and platform's lifecycle.

In EdgeIO, each node is initialized with the component called *Node Engine*, which already monitors the internal resources and advertises them asynchronously to the Cluster Orchestrator. The consistency of the information kept by the cluster is eventual, and there can be discrepancies between the data known by the cluster and the actual reality. In the meantime, services still work and expect to communicate and operate within the infrastructure. Then, while it is possible to trust the information provided by the cluster, is always needed a "*plan B*" for when things go wrong.

The burden of enabling and maintaining the connection across services is taken by the *Net Manager*, which is composed of a set of sub-components detailed in §3.5.1. The *Net Manager*

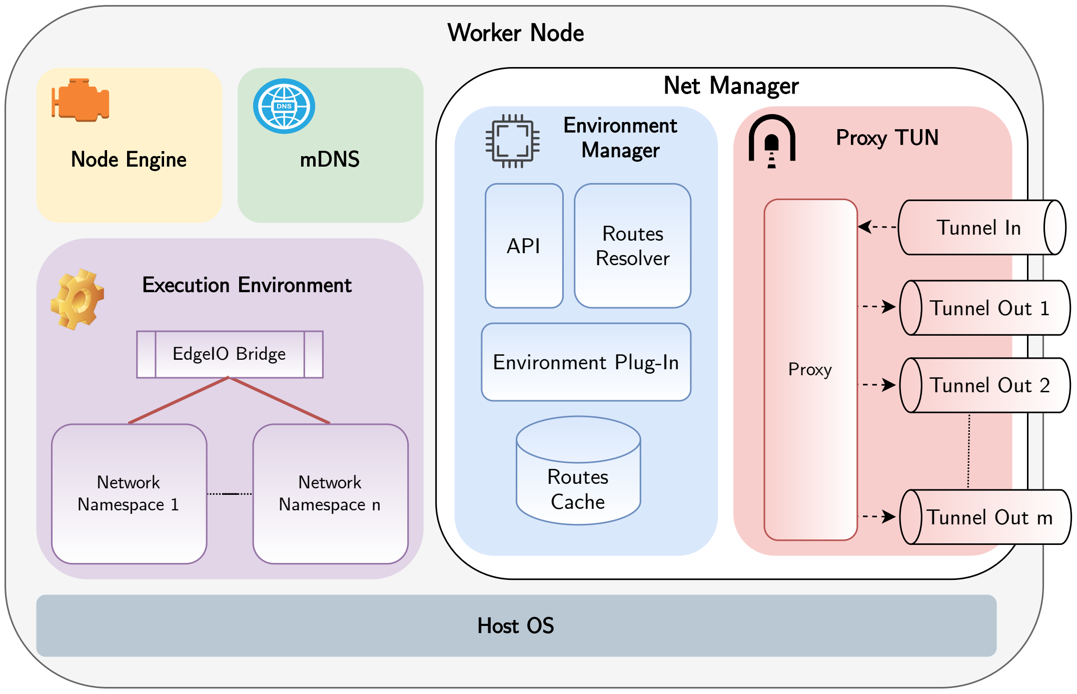


Figure 3.5: Architecture of a worker node

is treaded as a plug-in that manages all the networking aspects of the node. The interface exposed by this component, detailed in §3.5.2, is used to cooperate with the *Node Engine* for the deployment/undeployment operations of services and the node initialization.

3.5.1 Node internal architecture

Figure 3.5 shows all the components that are installed inside a worker node to enable the management and communication of the applications. Following the principles introduces in §3.1 the internal components of the worker are separate from the Node Engine and cooperate via API call to enable the service's networking.

The **Environment Manager** is the point of contact between the Node Engine and the rest of the network components. At startup, the Node Engine sends a registering message to its Cluster Orchestrator using WebSocket. The orchestrator answers back with the message broker credentials and a new subnetwork for the node. Given the new subnetwork's IP, the Node Engine uses the initialize command exposed by the Environment Manager to instantiate the new components that enable the networking. Through the initialization command, the environment manager creates the EdgeIO Bridge, initializes the mDNS, and creates a ProxyTUN process using the environment plug-in component. The custom bridge is used to interconnect all the network namespaces of all the services. Note that the services are not only containers but can be Unikernels, VMs, or bare metal applications as well. Each service resides in its isolated network namespace and is connected to the bridge using a virtual ethernet (Veth, red links in fig. 3.5). Upon receiving a deployment command of an application, the Node Engine creates the instance inside the Execution Environment. It uses the Environment Manager API's to request the activation of the networking capabilities for that service. The Environment Plug-In creates a new Veth pair, assigns to it a unique internal address from the ones available in its subnetwork, and attaches one side in the EdgeIO Bridge component and another side to the Network Namespace of the new service. This operation brings all the services linked to the same bridge and enables communication across the different virtualization technologies. The Environment Manager also provides a Routes Resolver and Cache component. The resolver is used to keep the route cache populated and up to date, while the ProxyTUN uses the routes cache to understand which worker is suited for tunneling a packet. If the cached entries are not enough or outdated, a cache refresh mechanism called *table query* takes place (§3.6.6). To keep the routes up to date, the Routes Resolver registers interests to the Cluster Orchestrator for a particular route. Every time a route is modified because of internal service migration, scaling up or down operation, or failure, the component is notified. Whenever the tunnel fails to deliver a packet correctly because the service is unavailable on the other node, the resolver requests a new cache refresh operation with a table query.

The **Proxy TUN** component can be seen as the gateway of the system. All the inbound and outbound network traffic is forwarded to/from the TUN process, which is then responsible for each packet's routing, proxying, and balancing. This component exposes one Inbound tunnel over a preconfigured port and can handle *m* different outbound tunnels, one for each other worker node where the communication is needed. The proxy component operates at layer4, supporting all the protocols used for service to service communication. This component translates the destination address of the service with the address of one of its instances. As explained in §3.6.1, the addresses used for the communication are semantic addresses which imply as well a routing policy. This component then selects the destination using the logic described by the address chosen by the developer. For instance, if the ServiceIP used as recipient implies a RoundRobin rule, the destination is one random instance among the ones available in the cache. In the packet, the original destination *ServiceIP* is replaced with the new destination namespace address. The source address is instead replaced with sender's *Instance IP* to enable the reply from the destination. The InstanceIP refers uniquely to one instance of a service. This mechanism delegates the responsibility of choosing the correct instance to the worker node exploiting his limited local knowledge. Even if the choice is just an estimation of the perfect routing decision, this avoids high overhead in the node and a single point of failure in the Cluster Orchestrator. Once the packet is ready to be forwarded, the worker node IP address and inbound tunnel port are fetched from the local cache. If a tunnel to the destination worker node is already active, the packet is forwarded with that tunnel. Otherwise, a new tunnel is established. If the maximum active tunnels number is reached, using the Least Recently Used (LRU) policy, one tunnel is closed. Periodically a garbage collector runs in the node cleaning up from the cache the routes that have not been used recently and closing the tunnels that are not required anymore because of inactivity within the last *inactivity_threshold* seconds. Lastly, the mDNS component is attached to the EdgeIO Bridge and enables the services to use domain names instead of IP addresses for communication. The naming schema described in §3.6.2 allows an application not only to choose an instance of another service but to specify the routing policy directly. This increases the flexibility in the Edge enormously, exploiting the physical location or real-time capacity of the surrounding workers. The ServiceIP for a demanding task that must be handled from an instance with high hardware capacity can be resolved automatically by the mDNS using the keyword *highcapacity* directly inside the domain name of the service.

By design, the internal components of the worker node try to respect the requirement **R1** (Modularity) acting independently with respect to the Node Engine. The entire networking is a Plug-In that exposes an interface to add or remove applications from it. Very similar to the concept of CNI in Kubernetes [88]. This modularity enables EdgeIO to easily change the networking component in the future or even upgrade it and swap it under the hood with another version. The only requirement here is that the nodes to communicate must all have the same network component or at least a compatible one because the technique adopted is not standard. The requirement **R6** (Extensibility) is respected because not only the internal mechanisms but also the routing policies' logic is extensible too. The ServiceIPs supported by the platform can be simply extended by increasing the routing algorithms available inside the proxy. Adding new ServiceIPs and new routing logic only needs modifications to the ProxyTUN component. To alter the Table Query operation's caching logic or the API, the developer must modify only the NetManager. Each component is responsible for one thing in this design: changing the environment or managing the packets. The way the components are designed inside the network manager is very similar to the way the data plane and the control plane responsibilities have been split in SDN approaches. This fact confirms that this strategy can result in a good design choice in the long run.

In case of packet delivery failures, the ProxyTUN component tries to find out another suitable destination for the packer. If there isn't any in its cache, it tries to resolve the route with the *Cluster Service Manager* again. Since the consistency of the information in the worker is eventual, it can happen that a route has been modified or is just not available anymore. For each network failure combined with a cache failure, the ProxyTUN requires a *table query*. If any other instance is available within the cluster, the *Cluster Service Manager* notifies it. Otherwise, the query is propagated to the *Root Service Manager*. If any suitable instance is available within the entire infrastructure, it is notified to the worker. If no instance is known due to voluntary undeployment or because the app does not exist, then a *route not found* error is thrown. If the suitable instance is not found due to node crash or service crash, the default fallback mechanism of the orchestrator will deploy the instance again as soon as possible. This recursive fallback mechanism enables the NetworkManager and the ProxyTUN components to respect the **R2** (Fault tolerance). While designing EdgeIO since the beginning has been taken into account that workers are edge devices and the failure is not an exceptional case like the failure of an entire rack in a cloud environment. Contemplating failure by design was then non-optional for the networking components as well. They must consider the failure of the node, the service, and the network. It is crucial to find out an alternative path in case one is currently failing. Cooperation with the service migration and scaling features is needed. This is why at the root level, in case of node route discovery

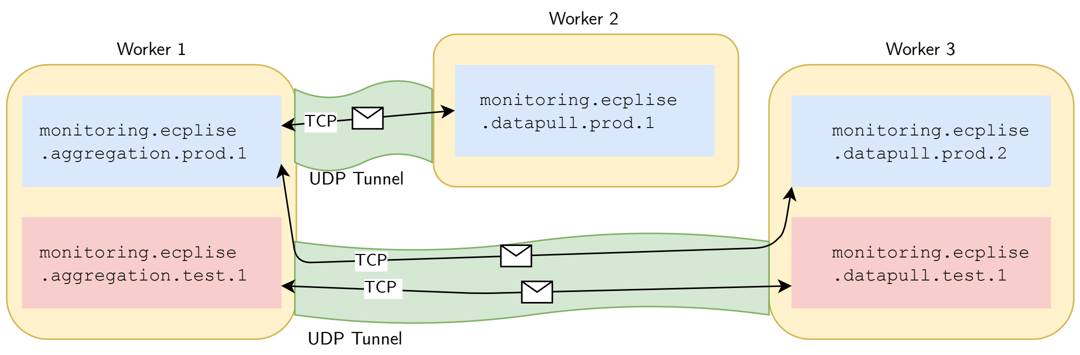

Worker 1
Worker 2
Worker 3
monitoring.ecplise .aggregation.prod.1
monitoring.ecplise .aggregation.test.1
monitoring.ecplise .datapull.prod.1
monitoring.ecplise .datapull.prod.2
monitoring.ecplise .datapull.test.1
TCP
TCP
TCP
UDP Tunnel
UDP Tunnel

Figure 3.6: Packet transmission

failure, the problem is moved to an orchestration problem; the network can't handle it. Fault tolerance partly also solves **R3** (Transparency), but still, it is not enough. To achieve this requirement and thus to manage all the networking aspects without involving developers' code changes or requiring the developer's intervention, the fallback mechanisms and the fault tolerance are only part of the story. The main advantage of using L4 proxying combined with semantic addressing is that apart from choosing the suitable *domain name* of a service or an appropriate IP address, all the operation happens under the hood. The balancing, route retrieval, routing, and tunneling do not involve the developer. Even traffic from different virtualized environments is treated seamlessly. The developers are only involved if they want to exploit a smart routing policy, dynamically changing the way the *ServiceIPs* or the *domain names* are chosen. The developers can use any of the protocols based on top of network layer 4. Thus, any protocol that uses TCP, UDP, or QUIC packets can be seamlessly handled by the networking components. The QoS is maintained by exploiting the characteristics of the original service protocol. For instance, the UDP channels and the proxying do not impact the delivery of a TCP packet in case of packet loss in the tunnel. As shown in fig. 3.6, the original data flow from the service will automatically grant packet retransmission, making the entire networking stack of these components transparent to the service.

The network components' overhead impact and the routing decisions' effort are reduced at two levels. On the one hand, the *Cluster Service Manager* uses a priority list and sends only the most relevant instance routes for each service to avoid flooding the worker node. The worker can only make approximate routing decisions among a few of the available instances, but this pruning speeds up the process making the routing lighter. On the other hand, the worker maintains only a limited number of open tunnels and cleans up the internal caches periodically, removing the routes that have not been used recently, thus reducing the selection space for future routing decisions. Limiting the decision space is the key to reduce the overhead, making this design able to respect **R4** (Low overhead).

Handing over the maintenance of the communication and the tunneling to the workers has been done to avoid congestion at the cluster level. Each packet travels directly from worker to worker, maintaining a distributed communication, and still, the hierarchical structure enables easy maintenance and monitoring. Another advantage of the distributed approach is that the traffic is not balanced through third-party remote locations. Still, it is handled directly inside the node with low overhead helping the reduction of the latencies as well and thus enforcing **R5** (Low latency). Even if the recipient's choice is approximate, there is a tradeoff between choosing the perfect destination and spending too high computing time. Even here, the reduction of the decision space seems to be the most suitable path.

The components installed in a worker node require flexible management of the networking namespaces. For this reason, Linux is the target OS for this design. This architecture also enables the worker node to bypass NAT and firewalls by just exposing one port. There is no need to have full control over the infrastructure or to install SDN-compliant routers. Moreover, the current implementation can support different ISA, like ARM and x86, as long as it is possible to compile the binaries. Therefore even **R7** (Infrastructure independency) is satisfied. When the infrastructure management is so limited that it is not even possible to open ports and thus expose the node outside the NAT, it is possible to extend the design mentioned above, enabling tunneling through the *cluster service manager* but introducing overhead and latencies. Anyway, the modular design of the architecture makes any further tunneling and routing approach easily implementable.

At the worker level, each application is virtualized and deployed with a separate networking namespace. The physical isolation depends on the virtualization technology chosen. In contrast, the networking isolation depends on the *routes sharing policy* enforced by the *Environment Manager* and the *Service Managers*, as discussed in §3.3. Applications belonging to the same namespaces can communicate directly; otherwise, a new rule must be added using the API exposed at the root level. This design solves **R8** (Isolation). Lastly, the possibility of dynamically influencing the routing decisions is a swiss knife for all the use cases in the edge **R9** (Flexibility). The routing policies can take into consideration the geographical location, capacity, and network conditions of the services. As discussed before is even possible to easily extend these policies, expanding the possibilities almost indefinitely. Of course, the more complex a routing policy is, the higher is going to be the general overhead, and the more are the metrics that must be stored in the node. Balancing is not a trivial task, and for this reason, it is essential to be careful with the complexity introduced at the worker level.

3.5.2 Environment Manager's API

The *Environment Manager* exposes a REST API that can be used to integrate the *Net Manager* with the EdgeIO's *Node Engine* but even with other similar platforms. As mentioned above this enforces **R1** (Modularity) and **R6** (Extensibility). The Exposed APIs are detailed in the rest of this subsection.

Initialization

To startup the *Environment Manager*, the endpoint shown in the following code at line 1 must be contacted using a POST request. The request must contain a JSON body with the subnetwork IP that the *Environment Manager* must use for the current node (line 5).

```
POST /api/register HTTP/1.1                                              1
```

```
Content-Type: application/json
data:
        {
                subnetwork:string
        }
```

Upon receiving this request, the *Environment Manager* creates the local bridge using the new subnetwork, initializes the mDNS and the ProxyTUN. The mDNS is initialized by setting the default data resolution address to the system's local resolver, the *Environment Manager*. The ProxyTUN is initialized by creating a new TUN device and attaching the new proxy's process to it. Finally, the routes to redirect all the traffic belonging to the private namespaces to the TUN device are installed into the node. The *Environment Manager* answers to this request with a *200 OK* or a *failure code* in case of internal startup problems.

Deployment and Undeployment

The deployment or undeployment of applications highly depends on the virtualization technology that is being used. The networking components can be attached to the network namespace only after the new container/unikernel creation process is complete. When the *Environment Manager* deploy API is called, it plugs into the namespace a new Veth pair and the routing rules for the ServiceIPs subnetwork. The API exposed by this component is summarized in fig. 3.7.

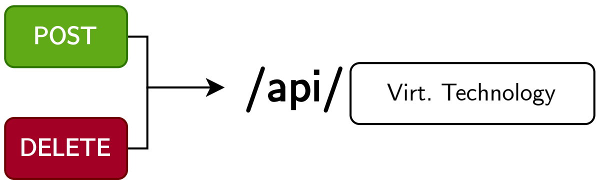


Figure 3.7: Deployment API scheme

The node engine can use the POST method to install the networking component on a service namespace or a DELETE method to remove it from one. The API supports different virtualization technologies by design, and this means that as long as the methods are implemented is possible to plug together even a unikernel with a docker container.

The following code snippet represents the request and response for deploying a docker container, respectively, and the request for its undeployment.

Docker deploy request

```
POST /api/docker HTTP/1.1
Content-Type: application/json
data:
        {
                containerID:  string
```

```
            serviceName:   string                                           6
            instanceNumber:  int                                            7
            serviceIP:[                                                     8
                   {                                                        9
                          IpType: string                                    10
                          Address:string                                    11
                   }                                                        12
            ]                                                               13
      }                                                                     14
```

Contacting the endpoint (line 1) with a POST request is possible to pass over the information needed to plug the networking components to the namespace. The *containerID* (line 5) is the docker ID of the container instantiated by the *Node Engine*. The field *app name* (line 6) represents the fully qualified name of the service. The *instanceNumber* (line 7) is the ordinal number that represents a specific service's instance; as explained in §3.6.1 and §3.6.2. Finally, the *serviceIP* array is a list of *IpType* (line 10) and *Address* (line 11) tuples describing for each ServiceIP the routing policy that it expresses. Note that in the serviceIP array, a special address of type *instance* is passed along with the other "real" ServiceIPs; this address is an instanceIP, see §3.6.1. In this API, the InstanceIP is treated just as a ServiceIP with a special routing policy: *route always to a specific instance*.

Docker deploy response

```
Content-Type: application/json                                              1
data:                                                                       2
      {                                                                     3
            serviceName: string                                             4
            nsAddress:  string                                              5
      }                                                                     6
```

In case of success, the response to the deployment request contains the fully qualified name of the target service (line 4), and the *nsAddress* (line 5). This very last information is crucial because it represents the physical address of the service, not to be confused with InstanceIPs and ServiceIPs. This address is private and assigned by the subnetwork of the node. It is used by the proxy and the tunnel to address the real physical namespace of the destination node, and it is never used by the developer, see §3.6.5.

Docker undeploy request

```
POST /api/docker HTTP/1.1                                                   1
Content-Type: application/json                                              2
data:                                                                       3
      {                                                                     4
            serviceName:string                                              5
      }                                                                     6
```

The service undeployment operation is triggered using the endpoint in line 1 with the DELETE method. The required parameter is only one, the *serviceName* (line 5), which uniquely identifies a service deployed inside a node. Note that in this platform, by design, each node can only have one instance for each service.

This API is automatically extended with the same schema to other virtualization technologies accordingly.

3.6 Service deployment and communication

This section describes how the services communicate together in detail through the proposed naming schema and the ServiceIPs. The high-level architecture described in §3.2 presents a distributed infrastructure where the worker nodes enable this point-to-point communication. Nevertheless, from the point of view of the developer may still not be clear how the services can be deployed and how an application can perform a network call to contact another one. The proposed networking approach provides this naming abstraction that addresses services and allows the possibility to specify a routing technique altogether. The DNS names are resolved to ServiceIPs, which enforce their own semantic with the L4 proxy mechanism. Packets then travel around the network using the overlay that connects all the nodes. The service-to-service addressing problem is discussed with a bottom-up approach in §3.6.2 and §3.6.1. Subsequently, §3.6.3 describes how the developer can setup the communication with the deployment descriptor of a service. Then this section continues explaining what happens when a service requires a route, how it is resolved §3.6.6 and how the overlay network looks like §3.6.5. Finally, a real-world full app deployment example is provided, and the entire process is analyzed in detail.

3.6.1 ServiceIPs

In this networking scheme, the core logic behind the routing decision made for each packet is represented by the *ServiceIP*. This address does not indicate the absolute position of a client, unlike the host addresses. This address is entirely virtual and represents the fully qualified service name with all its instances, like the cluster address of Kubernetes [104]. Moreover, it also states a semantic describing how the instances must be chosen. To summarize, a ServiceIP is an address representing two things:

- Fully qualified service
- Routing policy

This address has the exact shape of a regular IPv4 address, four dot-separated octets of bits humanly represented as integers in the range [0,255]. In order to distinguish such addresses from the rest of the IP addresses, a unique address range must be reserved. Every address residing in the preconfigured range is recognized as ServiceIP. By itself, this address does not symbolize any additional information. Due to the limitation in the addressing space, it is hard to encode/decode services and dynamic routing decisions with this format. The approach used is instead the one of generating extra metadata that is bound to the address and then exchanged by the platform's components. If an address resides in the ServiceIP range, then the corresponding metadata must be fetched. From this point of view, a ServiceIP is an identifier used as the key to fetch the actual information in a DB. This abstraction is needed in order to let the services use the standard networking stack without the need to change its current implementation, but with the possibility of introducing additional logic in the way the L3 communication is carried out. At deployment time, each service is bound to as many ServiceIP addresses as the implemented routing policies are. For example, if in the system there are four implemented routing policies, a service with *n* instances is represented by:

- 4 ServiceIPs
- *n* Instance IPs
- *n* node namespace IPs

The main difference between a ServiceIP and an Instance IP is that while the former addresses all the instances with a specific routing policy, the latter addresses only a particular instance of a service. Instead, the namespace IPs are just physical addresses representing the position of the isolated instance's namespace generated by the node at deployment time. The namespace IPs are never even used by the developer. One might ask if namespace IPs and instance IPs couldn't be the same thing. The main point in favor of keeping them separate is that while a namespaces IP is dynamic, the instance IP is assigned at deploy time and is fixed for the entire lifecycle of a service's instance. Since a service instance can migrate from one node to another, the underlying physical subnetwork can change, and the namespace IP with it as well. Note that before the deployment, each node *must* create a separate network namespace to enable isolation at the hypervisor level and enable local addressing of the packets. Having an instance IP address that uniquely describes a service's instance, regardless of its position and current node implementation details, is what enables these components to support state-full services. Instead of load balancing, a developer can simply address the traffic to a specific instance and keep the state consistent. In stateless applications, the ServiceIP is the correct way to proceed, letting the proxy handle the burden of balancing the traffic. Suppose a developer wants to make a networking call with any protocols based on the top of L4. In that case, it only has to specify as destination address a ServiceIP. The platform handles the complexity of finding the ideal route according to the semantics expressed by the address. If the service with the specified IP is not found at all in the platform, then the usual *no route to host* error is returned. The protocol used to contact the destination ServiceIP completely masquerade the underlying infrastructure making the communication completely transparent. If multiple requests are performed to the same address, the traffic will be balanced according to the policy expressed. Note that if the node is reaching the destination for the first time, the initial latency may be slightly higher because of the preliminary *table query* §3.6.6. The latency of the following network calls is lower because the routes are already in the node cache. Note also that if a ServiceIP has not been used for a long time, the node may garbage collect the metadata associated with it. This can lead to sporadic network call having a slightly higher latency than expected. For a deeper understanding of the latency impact, refer to chapter 4.

The ServiceIPs do not have to be manually inserted by the developers. They are, in fact, automatically assigned at deploy time. If a developer wants to set a custom address, it is possible by specifying it in the deployment descriptor along with the policy bound to it, further details in §3.6.3.

Overall the ServiceIP abstraction is what makes this networking approach suitable for the edge. In a cloud environment, there is no need to specify dynamically the position of a service or the way the traffic must be routed, but it can be statically configured at deploy time. At the edge, because of the high heterogeneity in the hardware and the geographical location of the devices, it is convenient to have a mechanism that enables the developer to specify how a network request must be handled. If an API call is time-consuming, it must be forwarded balancing across several nodes or sent directly to the node with the highest capacity at the moment. If the following API call is instead a lightweight operation but with a low latency requirement, the developer may want to use a ServiceIP that represents the service with a *closest* policy to be able to find the nearest instance to the currently executed code. Furthermore, the edge continuously evolves, and the research on this field constantly finds new approaches to the routing and placement problems. Then, even adding new experimental policies is by design possible and encouraged in these networking components.

3.6.2 Service naming schema

Even if the ServiceIP abstraction is powerful and flexible, human beings are not quite comfortable with IP addresses. Moreover, if a ServiceIP for each routing policy is not manually specified in the deployment descriptor, the platform automatically generates them. This makes unpredictable the address that must be used to contact a service. While developing a service, it is much easier to write a human-readable name to address other applications. Many other platforms like [86] offers already the possibility to manage a service using a custom domain name that represents a specific replica [105] or an entire cluster of instances [104]. Then, using just the service information already known at deploy time is possible to ask the local DNS to resolve a query and dynamically see the IP address of the service. This makes the code more readable and makes it easily configurable and more robust to changes. With that in mind, the standard naming schema for service addressing used by other platforms is not powerful enough to describe the ServiceIPs. It must be possible directly from the domain name to choose the routing policy semantically expressed by these addresses.

The proposed naming schema (code snippet 3.1) includes the fully qualified name of a service and then allows to specify an instance number (to bypass the routing mechanism altogether) or a routing policy (to let the platform handle the traffic for stateless applications).

$$appname.appns.servicename.servicens.instance.policy.local \tag{3.1}$$

i *appname*

Following the namespace hierarchy described in §3.3, this naming schema begins with the application's name. There can be many different applications in the platform, and each one must have a name at deploy time. The name given to the application is case insensitive and will later be used to address it. From this perspective, an application is a wrapper of one or more services.

ii *appns*

Every application belongs to a specific namespace. The namespace can be, for instance, the name of the company or a particular execution environment for the application. An example of a fully qualified app name can be "`pipeline.tum...`" which denotes a pipeline application in the Tum' namespace. A namespace can contain many different applications, and as mentioned above, all the applications belonging to the same namespace are allowed to speak together.

iii *servicename*

The service name represents the identifiers of a service belonging to an application. Even this field is a case-insensitive string. Many services can be part of the same application, and this name uniquely identifies one of them within its own namespace.

iv *servicens*

The service's namespace is used to isolate it with respect to other services and is mandatory within this addressing schema. The couple "*servicename.servicens*" uniquely identifies a service within an application. An example of a fully qualified service is "`pipeline.tum.capture.production...`", which addresses the capture service within the production environment of Tum's pipeline.

v *instance*

The instance field is mandatory and can be filled with an `<int>` or with the keyword `any`. This field enables compatibility with the state-full applications and allows developers to choose a specific instance of a service. Each service upon deployment is marked with an ordinal number that goes from 0 to n. Each service will always have the instance 0, which is the first instance deployed into the system. In the case of state-full services, they can be manually scaled up or down using the CLI or the API at the root level. Each new instance has a unique ordinal number that can be used to address it. In this scenario, the instance's scaling choice is entirely delegated to the developer since it knows the logic behind the service. This mechanism has been introduced solely to give support to these types of services. If the developer does not want to specify an instance but rather give the load balancing responsibility to the platform, it is possible to use the keyword `any`. Note that since the state-full instances are manually scaled up or down, the developer knows the instance number since the beginning of the deployment.

vi *policy*

The `policy` field is the main difference of our approach with respect to many others. Standard naming schemes do not contemplate the possibility of specifying the routing decision directly from the domain name. Here, in our proposal, the routing policy is part of the domain and is treated as such. Because of the availability of many different semantic addresses, a fully qualified service name resolved with the `closest` policy has a different IP address than the same resolution but with the `RoundRobin` policy. This enables the developer to specify the service and policy directly inside the domain name of the service. This keyword can be any of the supported routing policies. While writing this master thesis, the policy that we had in mind are: `rr`,`closest` and `hc`. Which respectively are round-robin balancing, closest instance available, and highest capacity instance. It is possible to extend the design to support many more or even a combination of these policies, like `closesthc` that chooses among the `k` closest instances the one contained in the node with higher capacity. If the instance number has been specified, then the routing policy field must be `none`. Note that a routing policy is applied only if the domain name contains the `any` keyword in the *instance* field. In order to enforce this constraint, domain names like "`a1.a.s1.s.0.rr...`" are not resolved. Only the ones in the form "`a1.a.s1.s.any.rr...`" or "`a1.a.s1.s.0.none...`" are resolved instead.

vii *local*

The `.local` keyword is needed in order to address the resolution to the local mDNS. This is a required keyword for mDNS technology.

A complete example of the high-level usage for the proposed naming schema is shown in the code snippet 3.2. This example showcases how to build the service name to perform a GET request to the `/api/status` endpoint exposed at port 8080 of a service belonging to the Tum's test app.

$$http : //test.tum.login.develop.any.rr.local : 8080/api/status \tag{3.2}$$

From the point of view of the developer, by performing this HTTP request from another service deployed inside the platform, one of the *develop* instances of the *login* service will be chosen accordingly with the *rr* (Round Robin) policy.

Under the hood, what happens is that the domain name of the query must be resolved to an IP address. The *.local* keyword redirects the DNS resolution to the node's local mDNS shown in fig. 3.5. In detail, the request starting from the node's namespace via the Veth reaches the hypervisor's namespace where the mDNS instance is deployed. This component internally contains the known domains as shown in the example table 3.4. This component maintains one entry for each complete domain constituted of a fully qualified service name, instance number, and policy. This means that for each fully qualified service name, there can still be many possible resolution addresses, and they are all defined by the couple *instance.policy*.

What is returned by an mDNS query is a ServiceIP representing the policy expressed. Figure 3.8 shows the IP selection process intuitively.

If the domain name contains a routing policy and the instance is `any` then the returned address is a ServiceIP. In the eventuality that the domain name does not contain a routing

Table 3.4: mDNS resolution table example

Domain	Address
`test.tum.login.dev.0.none`	`172.30.0.0`
`test.tum.login.dev.1.none`	`172.30.0.1`
`test.tum.login.dev.2.none`	`172.30.0.2`
`test.tum.login.dev.any.rr`	`172.30.0.3`
`test.tum.login.dev.any.closest`	`172.30.0.4`
...	
`test.tum.login.prod.any.rr`	`172.30.0.45`
`test.tum.login.prod.any.closest`	`172.30.0.46`

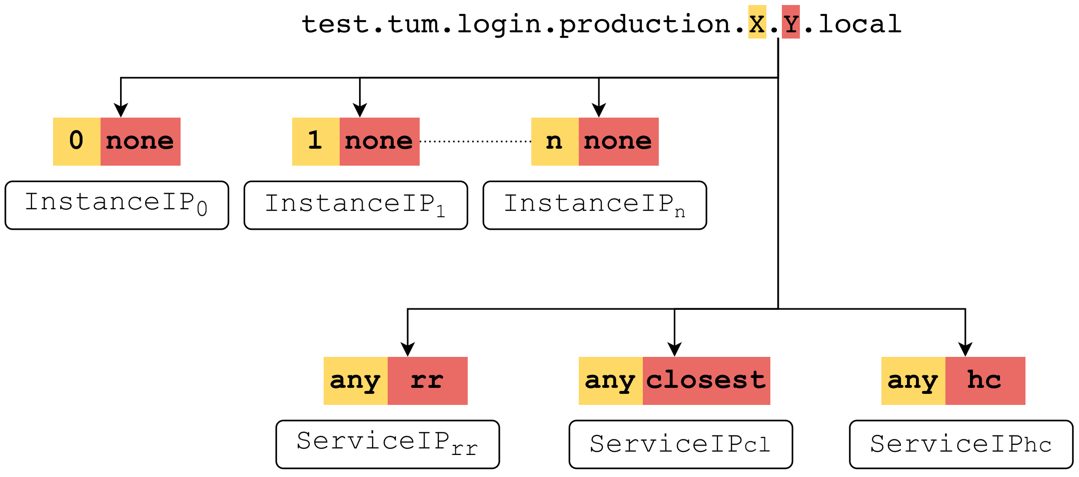


Figure 3.8: mDNS resolution logic

policy but an instance number instead, the returned address is an instance IP. Not that the instance IP is just a particular case of a ServiceIP. The former represents a service with a routing policy, and the latter represents a service with a routing policy that always redirects to the same instance.

When a service performs a domain name query, the mDNS does not always have the correct answer in its cache. In the current architecture, the mDNS uses the *Environment Manager* as a client to resolve any missing route. The environment manager then initializes a standard *table query*. Since the information required by the mDNS to resolve the domain name is also needed later on to determine the tunneling and proxying decisions, for efficiency reasons, the *table query* is performed once, but the results end up in both the *Environment Manager*'s *Routes Cache* and the mDNS store. The *table query* mechanism is detailed in §3.6.6.

3.6.3 Deployment Descriptor

The services' deployment depends entirely on the deployment descriptor. This file fully describes the service with its name, the runtime used, and more, going from the image containing the code to memory size and CPU requirements. The descriptor is passed with the deployment API to the *Root System Manager*, which parses it, stores the services information internally, and performs the deployment of the first instance. Starting from the EdgeIO's original deployment descriptor template, we integrated some additional information, some of them mandatory and others optional, enabling the service to service communication and allowing the developer to understand how to address other internal services.

Deployment descriptor example

```
api_version: v0.2                                   1
app_name: demo                                      2
app_ns: default                                     3
service_name: service1                              4
service_ns: test                                    5
image: docker.io/library/nginx:alpine               6
image_runtime: docker                               7
port: 80                                            8
addresses:                                          9
        RR_ip: 172.30.25.3                          10
        Closest_ip: 172.30.25.4                     11
        Instances:                                  12
                from: 0                             13
                to: 10                              14
                address_from: 172.30.25.5           15
cluster_location: hpi                               16
node: vm-20211019-009                               17
requirements:                                       18
    cpu: 0 # cores                                  19
    memory: 100 # in MB                             20
```

Following the deployment descriptor example shown above, a clear focus of the new components introduced by this thesis will now be discussed. In the deployment descriptor v0.2, an application is now addressed not only by a unique name but with four different *mandatory* fields representing respectively the namespace and app hierarchy described previously in §3.3. From line 2 to line 5 there is the information necessary to form the fully qualified service name, shown in §3.6.2. Before the deployment operations begin, a service needs one ServiceIP for each routing policy and one InstanceIP for each instance. These addresses are assigned automatically by the platform, and thus, they are almost random from the developer's point of view. Suppose, for any reason, that the developer may want to specify a custom ServiceIP for some routing policy or assign an address to a specific instance. In that case, it is possible to do so using the deployment descriptor. The *optional* field in line 9 shows how to impose the platform custom virtual addresses for the services. For instance, it is possible to express a ServiceIP representing the Round Robin routing policy as shown in line 10.

Assigning addresses to the instances is a bit more complex. The instance number is a dynamic concept that needs to be addressed during the execution of the scaling operations. What a developer can do is to set an address generation rule to the *Service Manager* component as shown in line 12. Specifying an instance number *from* (line 13) and *to* (line 14) the developer can assign a contiguous set of $to - from$ IPs starting from the base address specified in line 15. All the new instances falling in the specified range will have a predictable address that follows this rule. In any case, all the addresses and ranges specified in the deployment descriptor must be available. No other service must be using those; otherwise, a deployment error is raised.

The proposed modifications to the deployment descriptor enable another layer of flexibility and management capabilities to the developers. Even if by design it is possible to address services only using the fully qualified service name, we wanted to give complete control over the infrastructure. Providing customization of the internal addresses enables the possibility of avoiding DNS queries and any extra related overhead. Moreover, the state-full applications are supported by default, giving even a controlled way to specify the address generation rule. Thanks to the independence in the underlying infrastructure and virtualization technologies, the newly introduced fields are invariant with respect to the runtime chosen for the deployment. The addressing of a service residing in a docker container is not different from that of a unikernel. Thus, the deployment descriptors are only going to differ in terms of the *runtime* and *image* fields. Giving complete control over the ServiceIPs also allows the developer to avoid deploying an mDNS for each node, reducing the extra resources consumption. As a general rule, the fewer components reside in a small edge node, the better. This is why exploiting the IPs configuration provided in this descriptor can help to improve the platform's impact on constrained devices.

3.6.4 Packet Proxying

The component that decides which is the recipient worker node for each packet is the ProxyTUN. This component is implemented as an L4 proxy which analyzes the incoming traffic, changes the source and destination address, and forwards it to the overlay network. This proxy component has been implemented at Layer 4 of the OSI model. This decision gives us a universal approach that enables routing for all the protocols based on UDP, TCP, and QUIC. It is well known that operating at L4 also brings disadvantages, like the per-packet overhead and the impossibility of making "smart" decisions. While the former is the price to pay for the universality of this proxying mechanism and can only be mitigated through optimization, the latter disadvantage can be overcome via the proposed semantic addressing approach. While the system can't analyze application-layer headers and figure out information regarding the packet, the developer can dynamically address heavy traffic and low latency traffic differently since the beginning. Part of the overhead caused by a possible L7 packet traffic analysis is transferred to the developer's choice for a convenient ServiceIP.

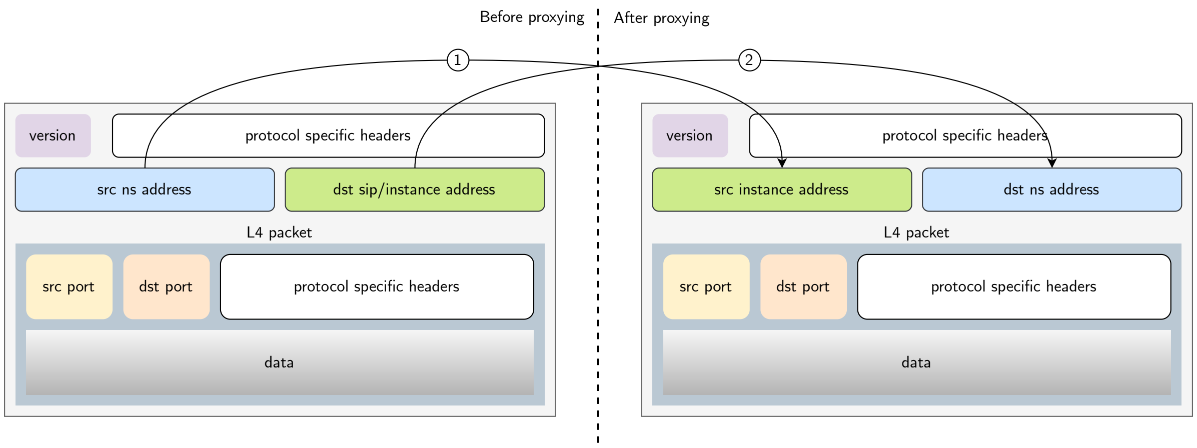


Figure 3.9: L4 packet before and after passing trough the proxy

The default proxying solution does not remove the possibility of introducing L7 proxies, but by default, the chosen solution tries to be as universal as possible.

A packet approaching the proxy has a namespace IP as the source address and an IP belonging to the subnetwork of the Service and Instance IPs as a destination. The L4 packet also has a couple of source and destination ports used to maintain a connection and contact the correct application on both sides. The proxy's job is to substitute the source and destination addresses according to the routing policy expressed by the destination address. As shown in fig. 3.9 the incoming packet at L3 contains the addresses pair and the L4 payload with source and destination ports. The proxy converts the namespace address of the packet, belonging to the local network of the node, with the InstanceIP of that service's instance ①. This conversion enables the receiver to route the response back to the service instance deployed inside the sender's node. The proxy always knows the InstanceIP for all the internally deployed services; this information is transferred along with the service at deploy time 3.5.2. The destination address is instead converted to a namespace address ②. If the original destination address is an InstanceIP, the conversion is straightforward using the information available in the proxy's cache. When the original destination address is a ServiceIP, the following four steps are executed:

1. Fetch the routing policy
2. Fetch the service instances
3. Choose one instance using the logic associated with the routing policy
4. Replace the ServiceIP with the namespace address of the resulting instance.

In any case, if the cache data are not enough, a table query is performed (§3.6.6).

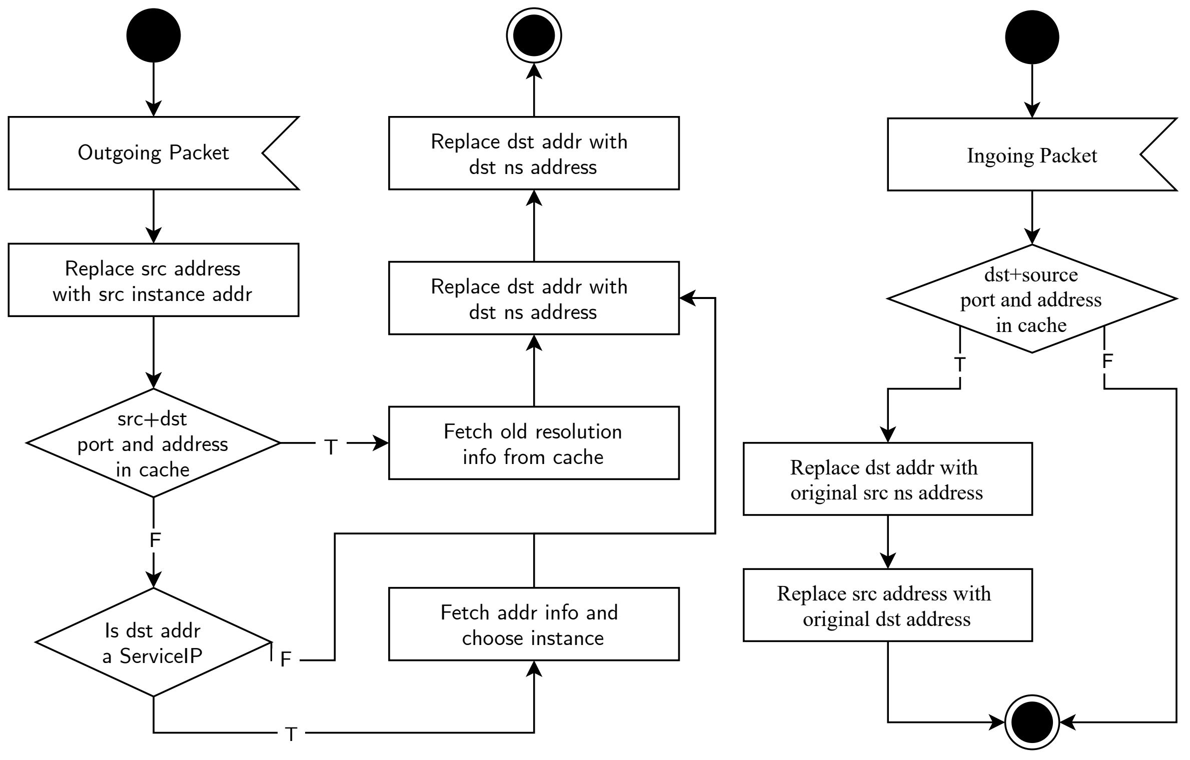
The proxy uses the ports numbers to cache the current chosen destination for that connection and reverse proxy the incoming traffic correctly.

After the correct translation of source and destination addresses, the packet is encapsulated and sent to the tunnel only if the destination belongs to another node (§3.6.5), or it is just sent back down to the bridge if the destination is in the same node. The asynchronous nature of the information propagation is necessary to lower the overhead in the system as much as possible. Still, it can bring some discrepancies between the internal representation of the world and the actual situation. The system cannot guarantee to be respecting the routing policy in absolute terms. It enforces the routing policy accordingly to the information contained locally. The local data is updated sooner or later by the *Cluster Service Manager* if any changes happen in the system. For instance, if the ServiceIP is associated with a *closest* policy, but the internal cache of the node has not been updated yet with the latest most immediate instance, the node may decide to send the packet to another node according to its local knowledge. It may also happen that the destination instance does not exist anymore. In this particular scenario, the *Environment Manager* of the destination node sends back a *route not found* error that is handled by the source node's *Network Manager*, resulting in a synchronous new *table query* that syncs up the local knowledge.

To summarize, the high-level algorithms that this proxy follows are the ones represented in fig. 3.10. In particular, fig. 3.10a is the algorithm used to proxy a packet coming from inside the node and directed to another service. In contrast, fig. 3.10b depicts the reverse proxy algorithm used to handle packets coming from outside the node and directed to a service deployed internally. Note that the reverse proxy can perform the reverse translation only if the route has already been translated before. If no translation record is in the cache, it just skips the packet and sends it as it is. Another essential detail is that the reverse proxy algorithm must consider the inverted destination and source for the translation. What was once the source port and address is now the destination port and address in the incoming packet.

3.6.5 Overlay network

After the packet sent from the service passes through the proxy and the addresses are translated, it is finally ready to be encapsulated and sent to the final destination. The overlay network is logically placed on top of the real physical topology and enables the possibility of having multiple nodes interconnected. The overlay network is maintained by the proxyTUN. This process executes both proxying (already discussed in §3.6.4) and tunneling of the packets to avoid multiple context changes in the system. As shown in fig. 3.1 each node exposes one port which is the *Tunnel In*. This tunnel is the one that accepts all the incoming packets, performs reverse proxy if necessary, and forwards them to the internally deployed services through the bridge. The *Tunnel Out* channels are used to forward the packets to the destination worker. Each *Tunnel Out* is directed to the *Tunnel In* of another node. Figure 3.11 shows an example of an overlay network across two clusters composed of a total of 6 nodes. The node $N_{i,j}$ is the $i-th$ node of the $j-th$ cluster. We represent with $|I_j|$ the number of nodes in the $j-th$ cluster, and with J the set of all the clusters. In this overlay implementation, each node $N_{i,j}$ has logically $n-1$ outbound links, one for each other worker, where:


Outgoing Packet
Replace src address with src instance addr
src+dst port and address in cache
T
F
Is dst addr a ServiceIP
F
T
Fetch old resolution info from cache
Fetch addr info and choose instance
Replace dst addr with dst ns address
Replace dst addr with dst ns address
Ingoing Packet
dst+source port and address in cache
T
F
Replace dst addr with original src ns address
Replace src address with original dst address

(a) Outgoing Proxy (b) Reverse Proxy

Figure 3.10: Proxy algorithm overview

$$n = \sum_{j \in J} |I_j|$$

Anyway, for scalability reasons, a node can't maintain all these tunnels concurrently. In fact, in fig. 3.11 we distinguish between *logical* and *active* links. A tunnel is maintained active only as soon as there are services using it. If a tunnel remains inactive, the internal garbage collector will close it. When a new proxied packet is ready to be tunneled, the information regarding the final destination is retrieved. If the tunnel to the destination node is already active, the packet is forwarded there; otherwise, a new tunnel must be created first. The final destination is addressed using the *worker ip* and *tunnel port* information that comes along as result of the *table query* (§3.6.6) performed by the proxy component. Then, we can define L as the set of all the *logical links*.

$$L = \{j, k \in J; i, l \in I_j; i \neq l : (N_{i,j}, N_{l,k})\}$$

Consequently, the set A of the *active logical links* is:

$$A \subseteq L$$

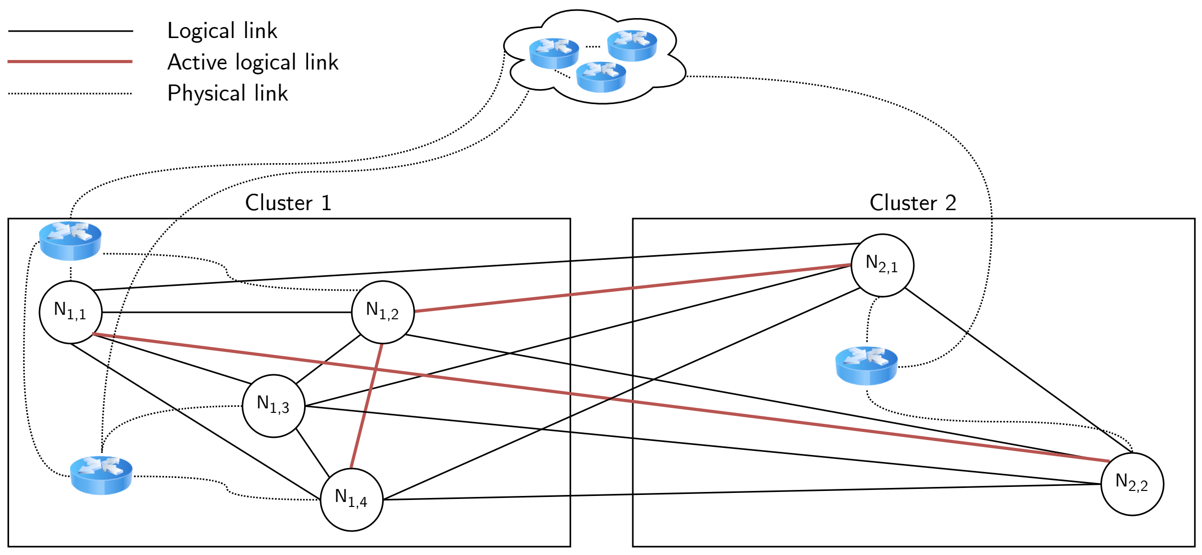


Figure 3.11: Overlay network across 2 clusters and 5 nodes

In order to avoid overloading a node, there is a maximum of k active tunnels that is possible to maintain. The old tunnels are then evicted using the *LRU* policy. With that said, the set of the *active logical links* becomes

$$A \subset L \;\; iif \;\; k < n$$

In fig. 3.11 it is possible to note that the overlay topology can be significantly different compared to the physical links. The overlay is a pure virtual layer that completely abstracts the underlying infrastructure. The packets flowing in the overlay are routed through the physical links transparently. A tunnel packet flowing through the *active logical link* $(N_{1,2}, N_{2,1})$ is just a UDP packet having $N_{1,2}$'s IP as source address with a random new source port, and $N_{2,1}$'s IP as destination IP with the respective *Tunnel In* port as destination port. The payload of the UDP tunnel packet is the original packet sent by the service with the translated addresses accordingly to the proxy algorithm. The UDP tunnel packet is sent through the network and goes over the public internet if necessary.

It happens very often in the Edge that more than one device resides behind firewalls and NATs. This means that multiple nodes share the same address and have limited visibility over the network. Anyway, on each router, one port for each one of the *Tunnel In* must be exposed and mapped to the NAT table. Then, using the public worker node's IP address and the unique *Tunnel In* port, it is possible to address each one of the workers participating in any cluster. In order to protect the nodes from the public internet, all the unrecognized packets are discarded. It is also possible to set up the firewall so that it filters out the connection belonging to unknown networks. The traffic belonging to the same platform can be identified using a couple of public and private keys to encrypt the packets, but this is left out to future work at the moment.

To summarize, each packet sent out with the tunnel is wrapped inside a new regular UDP packet and routed around the real underlying topology, all using only standard technologies. Since the tunnel packet contains the original service's packet as a payload, even using UDP as the tunnel protocol, the QoS is not altered. In fact, if the actual protocol used is, for instance, TCP, and the packet is lost during the tunneling process, the client will send another one upon not receiving the ACK within the expected time frame. From this perspective, the UDP tunnel can be seen as a wire plugged between two nodes where the packets can flow. The reliability problems of the wire are absorbed by design from the networking protocols used.

3.6.6 Table Query

The table query operation is the process that enables the route sharing mechanism for a specific service. A table query operation starts from a node and is propagated up in the hierarchy. This operation can be initiated from two different components, the mDNS or the proxyTUN. The former uses the table query when the information available locally is not enough to complete the domain translation. The latter uses this operation when the data in the local cache is not enough to understand the packet's final destination or the address for the recipient worker. In both cases, the result of a table query operation depends on the visibility of a service according to the namespace settings and on the content available in the *Service Managers* (table 3.1).

Figure 3.12 contains the sequence diagram describing in detail the table query operation when initiated by the mDNS. The table query initiated by a proxyTUN component is analogous in terms of procedure. The information retrieved is the same; it differs only in the format of what the final components get. In fact, for every table query, the worker's *Net Manager* receives all the information regarding a service; the mDNS and the proxyTUN just need a different piece of it. In this example, if the mDNS does not have enough information to resolve the domain query issued by the service $X2$, it performs the usual IP multicast request for the resolution. The only component that can answer the request in this architecture is the local *Environment Manager*. If the required information is available in the local cache, then the *Environment Manager* immediately answers the request. Composing all the names and namespaces of the service is possible to form the first four elements of the domain name. Using the available instance IP or ServiceIP information is possible to create the last part of the domain name. If the records in the manager's cache are not enough to satisfy the request, the table query process finally begins. The environment manager sends a request to the cluster's service manager. This initial request has the sole purpose of getting the first available information as soon as possible to resolve the pending request. We can then define the table query as a 2 step algorithm.

1. Fast resolution
2. Interest registration

The first request issued from the worker is the *fast resolution*. No matter how, the information needed to resolve the query must be retrieved as fast as possible, even if the result is not

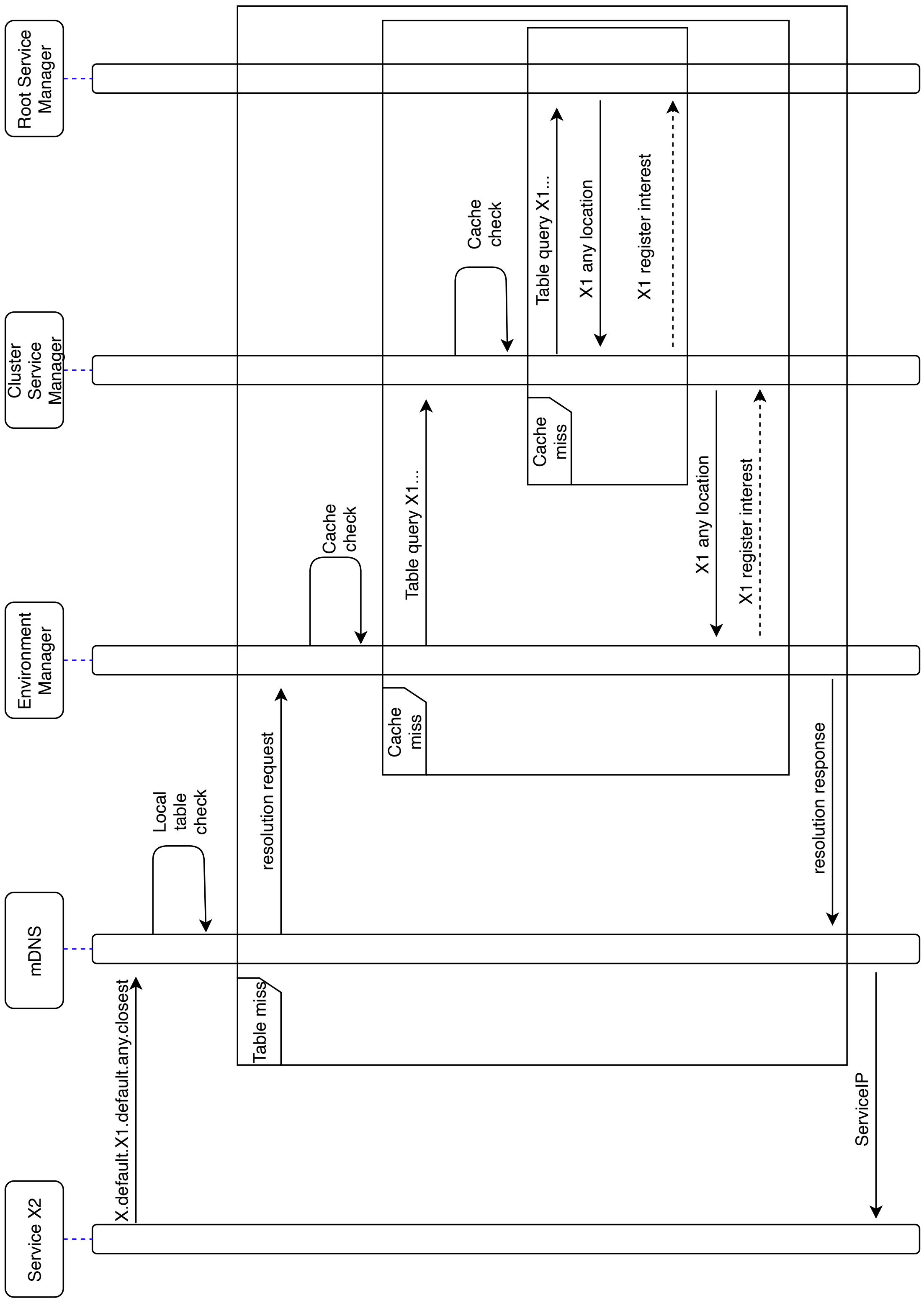


Figure 3.12: Sequence diagram of a table query started by the mDNS

optimal to fulfill the routing policy. After the *fast resolution* request is completed, the *interest registration* enables the worker to specify an interest for the routes to the service. Then, as soon as any additional information regarding the service object of the interest is available, it is propagated down to the node. The worker's cache is initially empty. It contains one record after the first *fast resolution* table query and is finally fully populated over time. This approximation initially reduces the possibility of always finding the optimal routing decision but speeds up the process consistently. Note that the algorithm used by the managers between the root and the cluster level is similar to the one used between cluster and node level. If a cluster's *service manager* does not have the information locally, it requires a *fast resolution* to the root's *service manager* and then registers an interest for that route. The root *service manager* has the global knowledge of the services currently deployed; thus, it is always able to resolve a query.

It is crucial to reduce the load at the root level, making the cluster manager able to resolve the routes autonomously when possible and even the node to resolve the queries without contacting the cluster for every service call.

From the standard table query mechanism described above, there are at least four exceptions to consider:

1. **Already registered interest**: A cluster may receive a query for which it already has an interest registered, but due to the asynchronous nature of the platform, the information available to resolve it is not enough. In this case, the table query is propagated to the root, but a new interest is not registered.

2. **Service unavailable**: If the table query resolution is propagated up to the root, but even the root's *service manager* is not able to resolve it, there can be two reasons. All the instances crashed, or the service simply does not exist. In the former case, the service is unavailable until the platform deploys a new instance. The latter is a request that will never be fulfilled. In both cases, what happens is a *route not found* error returned down in the hierarchy and no subsequent interest registration.

3. **Deployment interest**: When a service's instance is deployed to a cluster, the interest is immediately registered. A cluster must be aware of all that concerns the services deployed inside it. If a service's instance is deployed or scaled-down, the information must be propagated to all the clusters that have an interest registered for the service. This maintains the consistency of the data across the platform.

4. **Record list pruning**: A worker node can handle only a limited amount of records for each service; otherwise, the routing decision gets too heavy to bear. Before handing over the result of the table query to a node, the cluster's service manager performs a pruning. It sorts the entries by passing over the k most relevant instances for each routing policy. This is called $k - pruning$, where k must be configurable in the cluster service manager configuration file. A good default choice for k can be in the order of the dozens.

The registered interests have an expiration time; after that, a renewal request is sent to the cluster or the worker node to renew it. If the interest is not renewed, then it is decades. The worker node renews the interest if any of the services still require the route. If the route has not been used recently, the interest is no longer needed, and the renewal request is rejected. The cluster's service manager accepts the renewal request if and only if at least one of the following conditions is satisfied:

1. At least one of the services deployed within the cluster is still interested in that route.
2. At least one of the services deployed within the cluster is the service target to that route

The first condition is needed to enable the cluster's *service manager* to resolve the interests of the workers. The second condition is necessary for consistency with respect to the **Deployment interest** explained above.

3.6.7 Deployment

The new components proposed in this thesis use the topology enforced by the original EdgeIO's architecture. Thanks to this, even the deployment of an application is almost unchanged from the developer's point of view. Anyway, the underlying implementation changes significantly. The deployment operation requires more steps because of how the routes and the namespaces are assigned, and new messages are exchanged between the infrastructure's components. Using fig. 3.1 shown in §3.2 the entire process involved during the service's deployment will now be presented from the top to the bottom of the architecture, detailing every passage.

① - Initially, the developer prepares the deployment descriptor. The file is injected into the system using the web console or the API. The component used to inject the deployment descriptor is the *System Manager*. In this example the *System Manager* receives the deployment descriptor for the service `demo.default.service1.test`. The developer also specified a round-robin address. Upon receiving the deployment command, the *System Manager* contacts the networking plugin, in this case, the *Service Manager*, sending over the service descriptor. The Service manager decorates the service with all the IP addresses that the components down the hierarchy may need. The entire IP address generation process happens at this level exploiting the global knowledge of the network. The *Service Manager* first of all verifies if any other service already uses the `172.30.25.3` address. Suppose the given IP is available; it is used as Round Robin ServiceIP for the current service. It then generates one ServiceIP for each of the remaining routing policies. Suppose we have two other routing policies, then the service is decorated with a total of three ServiceIPs, two generated by the platform and one assigned by the developer. The decorated service is given back to the *System Manager*.

② - In the second step, the decorated service is passed to the *Root Scheduler* that decides the target cluster according to the requirement expressed in the deployment descriptor and the available resources for each cluster. Suppose the chosen cluster is *Cluster A*.

③ - The service is handled to the *cluster A's Manager* using the HTTP communication channel.

④ - At this stage, the service is ready to be deployed, but no instance has been assigned to any worker node yet. The current state of the service is *CLUSTER SCHEDULER* and the manager contacts the *Cluster Scheduler* to scheduler the service directly to a suitable worker node. After the scheduling decision, the new service instance has a target node, and the service obtains the state *NODE SCHEDULED*

⑤ - Once the service's instance has been scheduled, the *Service Manager* is informed and assigns an address to the service. In this stage, the *Service Manager* just scale up one instance into the internal tables. Then it asks the root's *Service Manager* for a new InstanceIP address to be assigned to the new instance. Finally, it returns the decorated service with the new instance and respective IP to the *Cluster Manager*.

⑥ - The root *Service Manager* is contacted for the generation of the InstanceIP address for the new service's instance. The address is generated from the same range of the ServiceIPs. This endpoint is called from the cluster's *Service Manager* for each new instance that gets scaled up or down. Only the manager component at the root can generate or free up addresses so that the consistency in the address generation remains solid.

⑦ - The instance of the service is finally ready to be placed in a worker node. The Node Engine is the entry point of the node to enable the service deployment.

⑧ - Upon receiving the service's instance, the *Node Engine* contacts the *Net Manager* for the initialization of the networking components needed to enable the communication for the specified container. It uses the API described in §3.5.2.

⑨ - While the networking components are initialized, the *Node Engine* completes the deployment finalizing the creation of the container in the *Execution Environment*.

⑩ - Upon the container is created, the *Net Manager* assigns a namespace address from the available node subnetwork to the Veth plugged into the container's namespace. It also injects a custom route to redirect all the traffic belonging to the custom network through the bridge. The newly generated namespace's address is advertized up to the cluster's *Service Manager* which also forwards it to the root's *Service Manager*. During this step, the route interest in the service that has just been deployed is registered as well. Finally, the service's instance gains the status *DEPLOYED*.

⑪ - When the service needs to communicate to another one using a ServiceIP or an InstanceIP, the traffic is routed through the *Net Manager* as explained in §3.6.4 and §3.6.5.

3.6.8 Migration

When necessary, the services within the platform can be migrated. There are two possible scenarios, migration initiated by the cluster or by the root.

1. Cluster service migration - The cluster orchestrator initiates the migration operation using directly the API exposed by the *Node Engine*. Consequently, the *Node Engine*

executes the undeploy command of the networking components' using the DELETE method of the API (§3.5.2) and stoping the execution of the container. The *Environment Manager* frees up the namespace address used by the container and advertises the change to the cluster's *Service Manager*. The cluster performs a temporary scale-down operation that lasts until the new instance is ready. As soon as the new worker's *Node Engine* receives the deployment command, it instantiates the newer container with the respective namespace address assigned by the *NetManager*. Then, the cluster's *Service Manager* advertises the new internal namespace address up to the root orchestrator.

2. Root service migration - The main difference with respect to the former case is that now the migration operation can involve multiple clusters. Then, the instance is entirely undeployed from a cluster and obtains a fresh deployment to another one. In this case the preservation of the InstanceIP is guaranteed by the *Root Orchestrator*.

Whether the migration operation is implemented by stopping a container and then starting up a new one or pausing and unpausing it, the network migration operation remains unaffected. Thus, as soon as the container migration is complete and the new routes are advertised to the nodes that have an interest registered, the network traffic continues to flow seamlessly. Note that the IPs used by a service are always the same; the platform abstract all the complexity and the underlying changes letting the application use always the same Service or Instance IP. The only possible disruption may be caused by the fact that changing the underlying position of the container, the cache of some workers may not be instantaneously updated. In those cases, the failure is handled by the fallback mechanism of the proxy. In particular, when contacting stateless services, the impact of the migration drops to zero if there are other instances available. With stateful services, the traffic hangs until the migration is completed.

3.6.9 Scaling up and down

The scale-up and down operation is relatively simple from the point of view of these networking components. The design already contemplates multiple instances. During the scale-up, the service acquires a new InstanceIP, a new namespace IP, and a new worker IP-port couple. The instance IPs are generated as usual by the root's *Service Manager* under the request of the cluster. The worker node IP and port are already well known by the cluster, and the namespace IP is generated upon deployment directly by the worker node. The scale-down operation removes all these instance-related addresses from the cluster and the root while the platform performs the undeployment. In both cases, the changes are advertised to the clusters and workers with an active registered interest in that service.

3.7 External traffic handling

Enabling external traffic to reach the services residing inside the platform highly depends on the possibility of exposing the nodes. Suppose a client resolves a hostname to an IP address.

In that case, the address must remain unchanged for a significant period of time, be reachable over multiple requests, and be able to satisfy the requested service. Unfortunately, the edge does not provide such guarantees; nodes can move, crash, or be replaced. Thus, addresses may change over time. Moreover, some nodes may reside behind different networks, each with further limitations. This section exposes a hierarchical DNS resolution mechanism §3.7.1 for external clients and two experimental balancer proposals (§3.7.2 and §3.7.3) to enable the communication originating outside the platform's boundaries to reach applications deployed inside.

3.7.1 DNS resolution

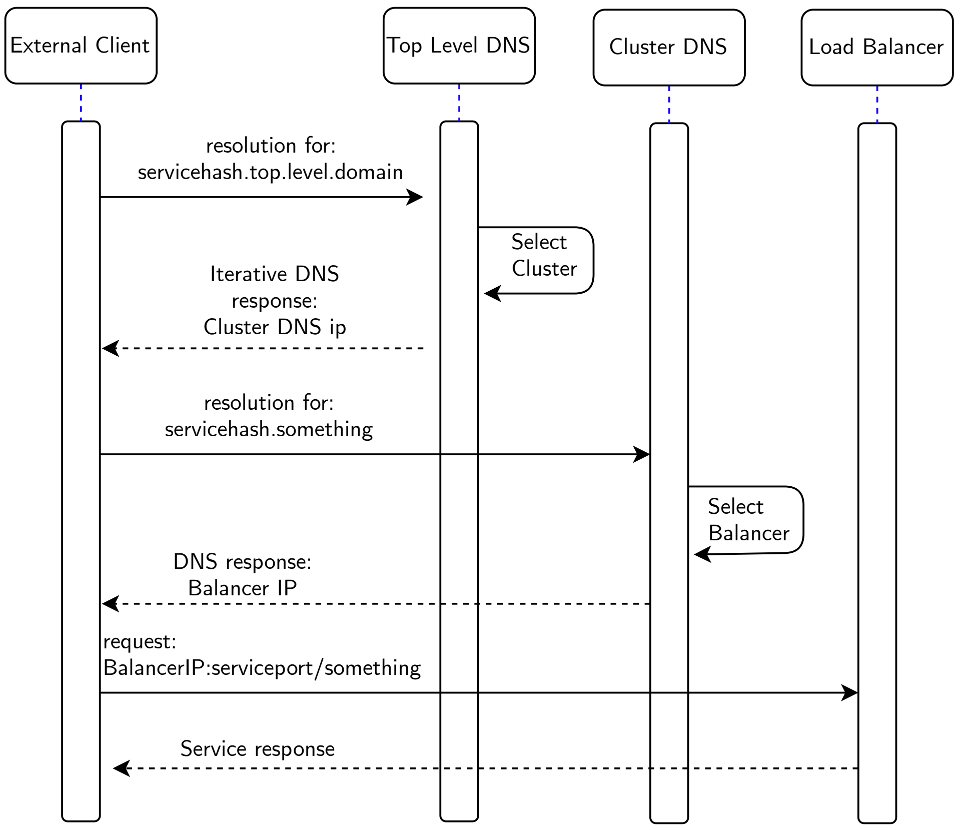

External Client
Top Level DNS
Cluster DNS
Load Balancer
resolution for: servicehash.top.level.domain
Select Cluster
Iterative DNS response: Cluster DNS ip
resolution for: servicehash.something
Select Balancer
DNS response: Balancer IP
request: BalancerIP:serviceport/something
Service response

Figure 3.13: Sequence diagram the external DNS resolution

Figure 3.13 shows the procedure to resolve the service name for an instance deployed inside EdgeIO using a hierarchical set of DNS. The root DNS answers an iterative query with the address of the DNS server within the cluster that contains an instance of the required service. The cluster-level DNS answers back with the address of a balancer. The balancer is the entry point for any communication and can be internal or external to the platform. The internal balancer role is taken by the *Service Manager*. The external balancer proposal is detailed in §3.7.4. A service that must be exposed outside the platform must contain an `expose:true` flag in the deployment descriptor. The system then assigns the *service hash* that must be used for the name resolution and a random port for the external service port mapping. The service hash is generated from the fully qualified service name composed of the app name and namespace plus the service name and namespace.

3.7.2 Redirection approach

The result of the DNS query is the address of a component that acts as a balancer for the traffic. The traffic addressed to the balancer can be tunneled or redirected to the destination. In the edge, nodes may change their position dynamically or can turn unavailable. The *Cluster Orhcestrator* must act as an entry point that abstracts the complexity of the underlying infrastructure. If the protocol used to contact the service is HTTP, it is possible to perform redirection [106] to bounce the traffic to the chosen instance. If the instance becomes unavailable, the following HTTP call is redirected to another node.

This approach is lightweight but requires the availability of a public address for a worker node and the possibility of exposing arbitrary ports. Moreover, the HTTP-redirection works only with the HTTP protocol making this approach not universal.

3.7.3 Tunneling approach

To enable universal communication from client to service is possible to exploit the balancer as a tunnel. As shown in fig. 3.14 the external client that resolved the domain name to the IP of a cluster balancer can contact that address to enable the communication. In this case, the balancer resolves the request internally, forwarding the result back to the client once it is available. This approach is still powerful enough to masquerade the topology complexity. It does not even need the exposure of a public IP address and port from a worker node, but just the usual tunnel. The downside is that the traffic handling can cause high overhead on the cluster and may be a bottleneck.

3.7.4 High Availability

High availability of the balancer can be achieved to avoid a single point of failure and increase the workload amount that can be handled. It must be possible to instantiate an external balancer with a dedicated address during the deployment of a service. The external balancer is the entry point for all the traffic directed to a specific instance and can perform redirection or tunneling. The external entity must anyway synchronize the routes, exchanging information with the internal balancer. This approach is very similar to the one proposed in [107] and is left out to future work.

3.8 Implementation Details

The design of the networking components exposed during this master thesis has not been fully implemented because of the time limitations. This section points out the current state of

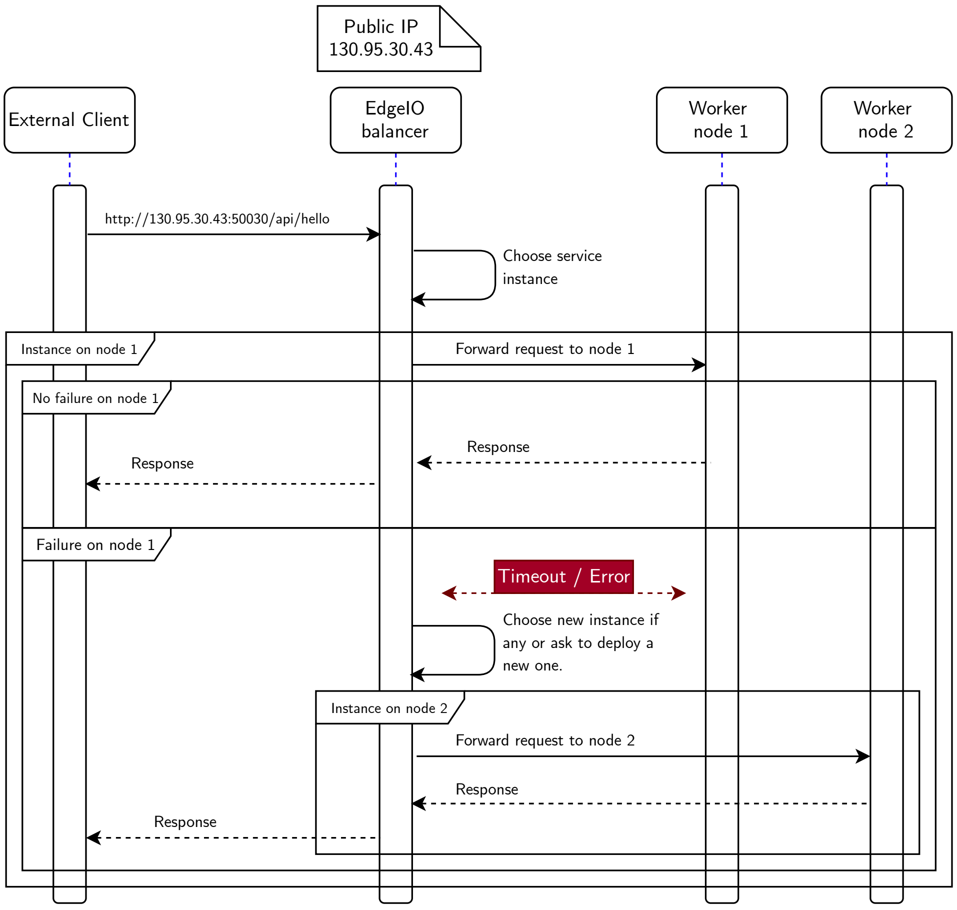


Figure 3.14: Tunneling for external traffic

work and the implementation choices made for each component.

3.8.1 NetManager

The components residing inside the worker node have been wholly developed, exception made for the mDNS. The NetManager has been coded as a single executable binary file developed using GoLang. This component takes advantage of two classes, the ProxyTUN and the EnvironmetManager, that implement the respective components described in the high-level architecture. The EnvironmentManager uses Gorilla [108] to expose the REST API used by the *Node Engine*. The bridge and the local routes are installed using directly the bash commands with the *IP* Linux utils. The TUN process is generated using the *go water* library [109]. The only supported virtualization technology right now is Docker, thanks to the *tenus* library used to plug the Veth pairs and retrieve the docker network namespace from the container ID.

Figure 3.15 contains a flowchart describing the current NetManager's implementation. After the startup, three async goroutines are generated. One handles the ingoing traffic, one

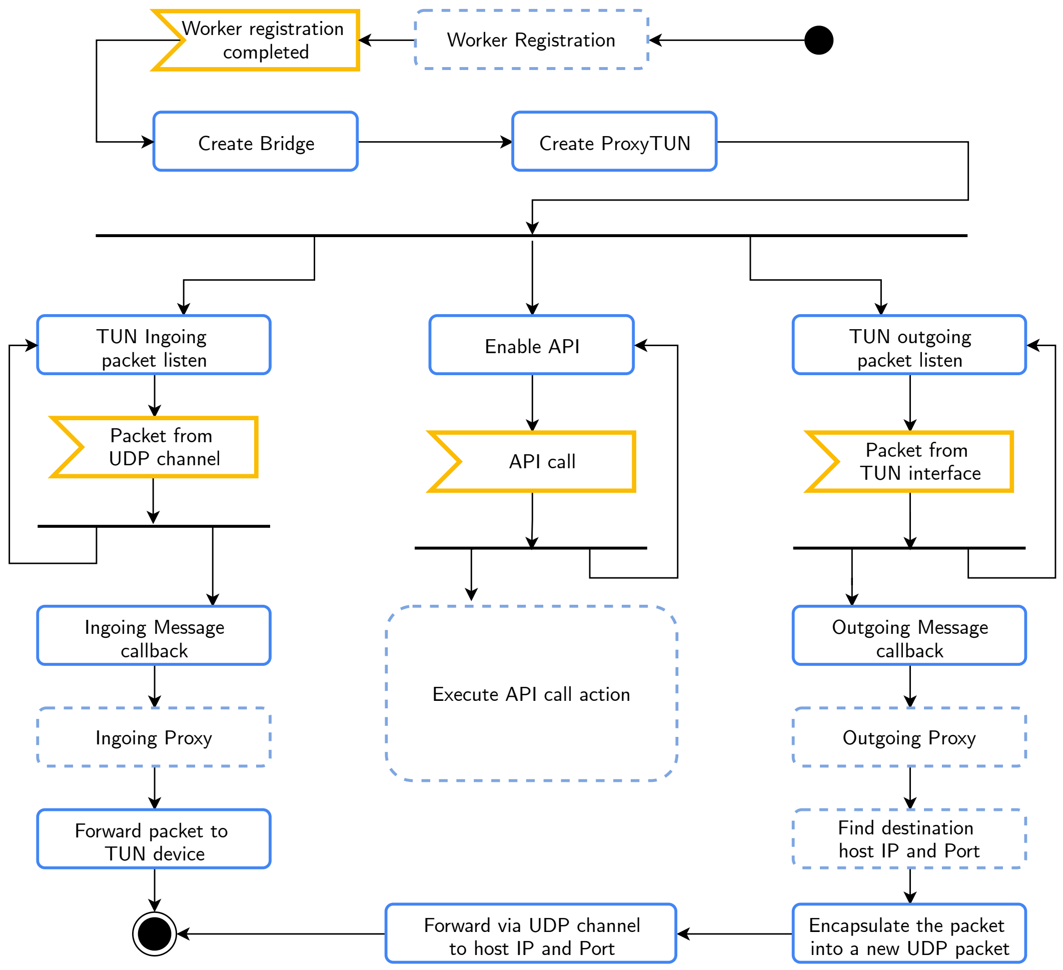

Worker registration completed
Worker Registration
Create Bridge
Create ProxyTUN
TUN Ingoing packet listen
Enable API
TUN outgoing packet listen
Packet from UDP channel
API call
Packet from TUN interface
Ingoing Message callback
Execute API call action
Outgoing Message callback
Ingoing Proxy
Outgoing Proxy
Forward packet to TUN device
Find destination host IP and Port
Forward via UDP channel to host IP and Port
Encapsulate the packet into a new UDP packet

Figure 3.15: NetManager implementation flowchart

the outgoing traffic, and one enables the REST API for the *Node Engine*. The Ingoing and Outgoing callbacks are the user-space handlers for the packets. They decapsulate the packet and hand it over to the proxy for the conversion operation. After the conversion is finished, the packet is forwarded to the final destination. The main reason why the proxy, the tunnel, and the environment manager are implemented in the same process is to limit the number of system calls and the context switch.

Figure 3.16 shows the context changes involved to proxy a single packet flowing from inside the node to an external location. Unifying the processes halved the commutation cost improving the overall performance. The *NetManager* is exported as an executable binary compiled for ARMv7 and Amd64 architectures.

What right now still needs to be implemented is the internal mDNS. Even though it is beneficial from a developer's perspective, the mDNS is not really necessary for the project's prototyping phase. We left it out for the moment since it can be plugged in afterward. To enable the service-to-service communication, we used the ServiceIPs.

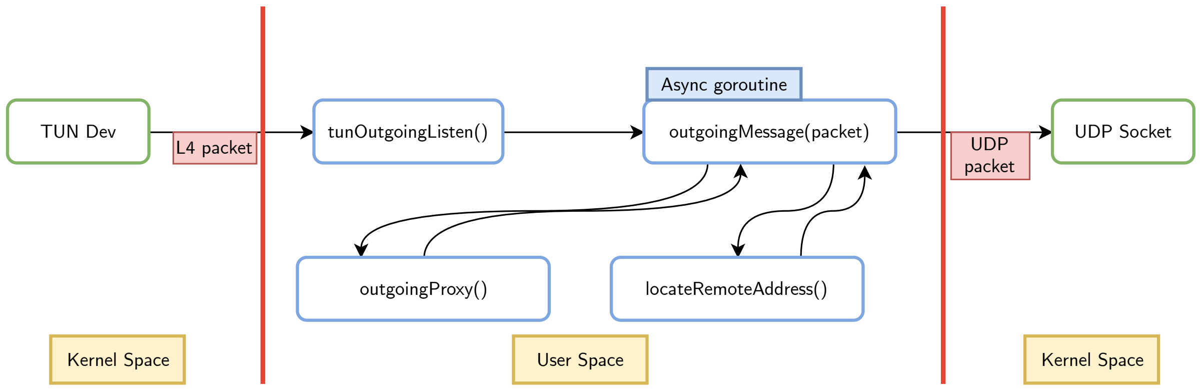


Figure 3.16: Context changes for the outgoing packet proxying mechanism

3.8.2 Service Managers

Due to time constraints, the *Service Managers* have not been fully implemented yet. The firewall rules and the interest registration functionalities are still missing. Right now, with a table query, all the known routes for the service are returned. Moreover, the current implementation just uses an additional class inside the Python 3.8 implementation of the clusters and root *System Managers*. In the future, these components need to be separated in a unique container to enable replication, scalability, and improve fault tolerance. The *Service Managers* communicate in each layer using REST APIs. A new implementation that uses MQTT as the primary protocol to interface clusters and nodes is a work in progress.

3.8.3 Service to Service communication

Service-to-service communication has been almost fully implemented as described in the architecture. The only current limitations are scalability and addressing. The former restriction is in place due to the early stage of the EdgeIO's development; in fact, the platform does not currently support scale up and down operations even though the network components do. The second limitation is due to the lack of the mDNS in the workers. Right now, the only way to address any service is through ServiceIPs. Moreover, at the moment, the only implemented policy for the ServiceIP is round-robin. The closest policy is under active development and uses the cluster positioning information to determine the most immediate instance.

3.8.4 External Communication

The external communication and the external DNS have not been developed yet. The current plan is to use a well-known mature and open-source DNS as the main driver. The DNS is a very sophisticated technology, susceptible to many attacks [110]. Using a standard and mature technology can speed up the development phase and reduce the involved risks. Given that, the only components that still need active development are the traffic balancers.

4 EdgeIO Network Evaluation

This chapter aims to showcase the pros and cons of the design decision of this project, showing in which scenario this approach is expected to be sensate and highlight the path for further development of the platform.

4.1 Experimental Setup

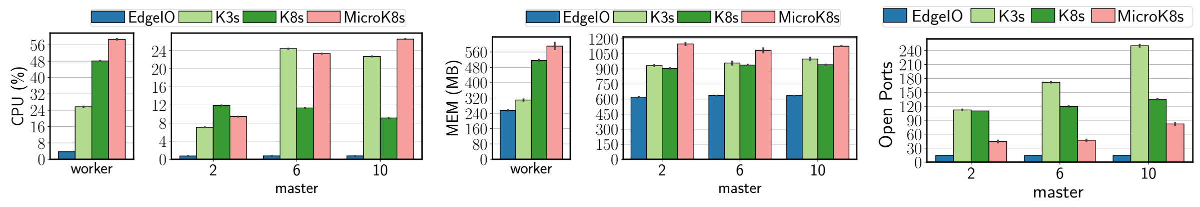
The testbed used to compare the proposed implementation consisted of 50 VMs being part of the Hasso Plattner Institut (HPI) cloud resources. The VMs were divided into four categories, from the smaller in size to the bigger.

- 17 VM of size S, with 1GB of memory and 1 CPU
- 17 VM of size M, with 2GB of memory and 2 CPUs
- 13 VM of size L, with 4GB of memory and 4 CPUs
- 4 VM of size XL, with 8GB of memory and 8 CPUs

All the VMs were Linux-based machines running Ubuntu18.04 LTS over an x86 architecture. For each platform the worker nodes were only S (§4.2.1,§4.2.2 and §4.2.3) or M machines (§4.2.4 and §4.2.5) to simulate constrained devices being part of the Edge. The Cluster manager were instead L nodes. The XL machine has been used only for EdgeIO's root orchestrator. Since the other platforms used for this comparison are single clustered, the evaluation has been carried out only against a single cluster deployment of EdgeIO. Thus, the root orchestrator is idle almost all the time. Each experiment has been repeated ten times over multiple days running only one framework at a time.

4.2 Evaluation results

The implementation discussed in §3.8 is now evaluated in this section. Using the test infrastructure presented in §4.1 we run some comparison tests against other mature orchestration platforms like Kubernetes [86], K3s [96], and MicroK8s [111], with their default CNI plug-ins. Then, we also compare the proposed tunneling approach against the well-known WireGuard VPN [97]. The goal is to obtain comparative data of the proposed solution in terms of resource usage (§4.2.1), latency (§4.2.2), bandwidth (§4.2.3), and packet loss (§4.2.4). Finally, we deployed a real-world image detection pipeline used for platforms benchmarking (§4.2.5). The deployment of such an application serves as a proof of concept to highlight how the current implementation can handle distributed heavy workloads.



(a) CPU utilization (b) Memory utilization (c) Open networking ports

Figure 4.1: System load comparison

4.2.1 System Load

The resource usage impact of the proposed networking components installed inside EdgeIO is now compared against K3s, MicroK8s, and Kubernetes. We created a script that continuously performs deployment and undeploy of one service in the system; while doing so, we measured the CPU, memory, and ports usage in the target worker node and the cluster master. We also repeated the test, increasing the number of worker nodes attached to the master, rising from 2 to 10 workers. The results are shown in fig. 4.1. As discussed during this master thesis, the deployment operation now involves a few new steps for IP addresses generation and routes propagation. These further activities may introduce additional system load both on the master and worker sides. Anyway, as shown in fig. 4.1a and fig. 4.1b, globally, the memory and CPU utilization of the platform remains much lower compared to others, even scaling up the number of nodes. Then, in terms of resource usage, EdgeIO turned out to be very lightweight. Figure 4.1c compares for each cluster manager the average amount of open ports needed to maintain the worker nodes active and manage the internal components. Even here, EdgeIO maintains a meager amount of open connections due to the asynchronous and independent nature of each component. In the current implementation, the networking stack use REST APIs for the *table query* resolution. In the future, we plan to integrate this operation with MQTT as well, reducing the number of active connections even further.

4.2.2 Latency

The presented networking components have also been evaluated in terms of the latency introduced in the communication. In the proposed schema, each packet that passes through the ProxyTUN arrives at the destination inevitably with an additional latency. It then must be considered the extra cost of transferring a tunneled packet and the handling involved by all the rest of the components. Each platform manages networking packets differently and introduces delays on different layers, even caused by high resource usage, for example. We decided to compare the latency introduced by the proposed networking components against the default networking approach of K3s, K8s, and MicroK8s. We repeated each test in 2 scenarios. After deploying a python client that requests a web page, we deployed first just one server, and then nine different servers able to answer. The client performed 100 requests in succession and measured the interval needed to obtain a response for each of them. The output of the client is the cumulative time required by all the 100 requests to

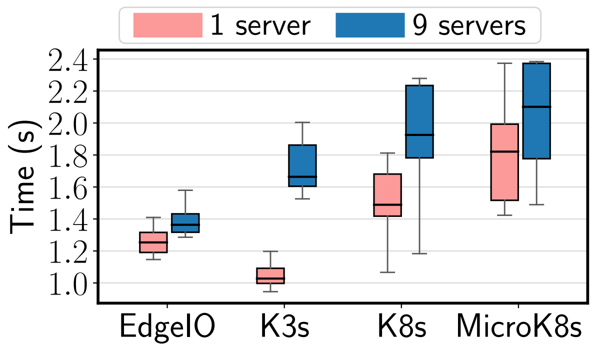


Figure 4.2: Client-server communication latency

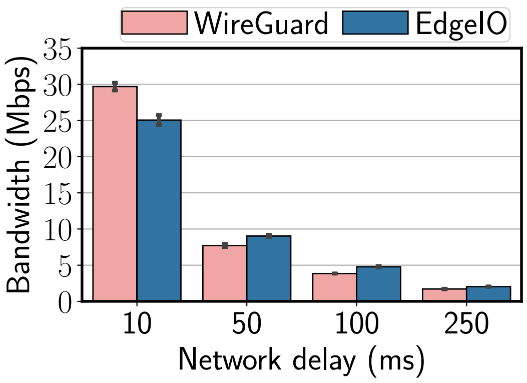


Figure 4.3: Bandwidth vs Latency

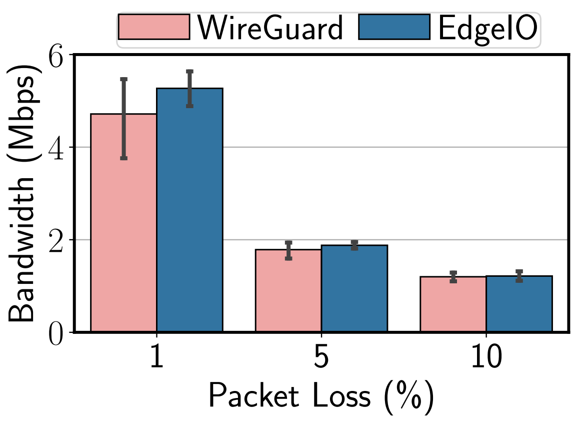
be full-filled. The results are shown in fig. 4.2. The servers were all deployed on VM of the same size, load condition, and homogeneous round-trip network time. Thus, the time difference between the test with one and nine servers represents the extra latency involved by the routing decision multiplied by a factor of 100; because of the cumulative output result. The results show that in the two workers setup, with just one client and one server, the lightweight VXLAN based approach of K3s is 0.3 seconds faster on average. By default, K3s works only within the local network, while the proposed components in this thesis already use tunneling to provide external communication. This leads, in general, to higher latency and explains the difference in this comparison. Anyway, on a ten worker setup, with one client and nine servers, the low impact in the routing decisions makes EdgeIO with the new networking components the clear winner. The lower latency in this multiple node setup is undoubtedly thanks to the locally distributed routing logic. In fact, each worker decided the final destination autonomously and routed the traffic directly without involving the master. As soon as a node has enough information, the decision is taken directly at the very edge of the network, making this approach ideal for such an environment.

4.2.3 Bandwidth

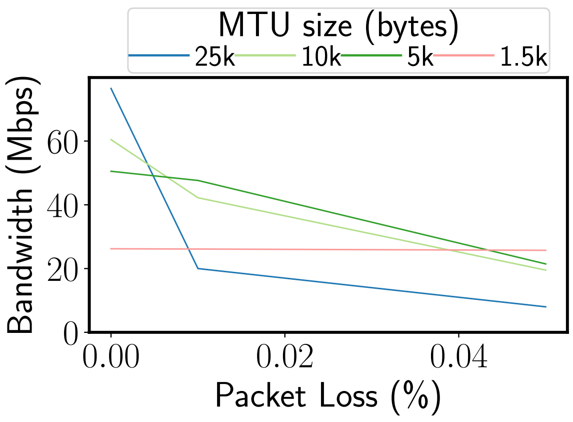
Another essential aspect part of this evaluation is the bandwidth measurement. For fairness, in this test, we compared the tunneling approach of this master thesis against WireGuard, one of the lightest VPN in the market.

We also wanted to simulate as much as possible a real world deployment by systematically increasing the network delay during each bandwidth simulation test. In fact, as shown in [112], edge nodes may bring high latencies because of the heterogeneous networking conditions in which they may reside.

Using one node as the client and one node as the server, we measured the time needed to transmit 100MB of data and inferred the bandwidth. With EdgeIO, we deployed two services on two different worker nodes and let them communicate using the ProxyTUN. To



(a) Bandwidth vs Packet Loss



(b) Bandwidth vs MTU size

Figure 4.4: Bandwidth measurement with packet losscomparison

compare the obtained results, we configured another two nodes with the WireGuard VPN and deployed the server and the client docker containers on them manually. The results are shown in fig. 4.3. In general, we can see that the overall bandwidth results are quite comparable, despite the fact that our approach does not only performs tunneling but also involves an L4 proxy decision for each packet. Modifying the internal MTU size, it is possible to decrease the overall impact of the L4 per-packet proxying, thus reducing the computation needed for each frame. The default MTU size used by the platform is 2000 bytes, meaning that the virtual ethernet size that connects each service to the internal TUN devices reads packets of size 2000bytes. The TUN device then sends the packet to the overlay using the standard MTU of the network medium. Increasing the MTU size between the *ProxyTUN* and the containers imposes a lower amount of translations to the proxy, consequently increasing the bandwidth. Anyway, as shown in §4.2.4, increasing the MTU makes the system less resilient to packet loss. In the current implementation, the MTU size is entirely customizable via the *NetManager*'s configuration file.

4.2.4 Packet Loss

The packet loss may be substantial in edge environments because of the poor connectivity to which nodes can be subjected. Then, the network components must be able to deliver the packets even during these harsh conditions. The presented UDP tunnel approach by itself is resilient to packet loss as much as the originating service protocol is. Meaning that if the services are communicating with TCP, they simply retransmit each lost packet. Thus, if a tunnel packet is lost, the service will retransmit a packet of the MTU size configured.

In order to decrease the latency introduced by the proxy component, the system provides the possibility to tune the MTU size of the internal links. It is possible to increase the MTU of the Veths that connects the service's namespace to the system bridge. Increasing the MTU size decreases the number of translations the proxy must perform. This improves the bandwidth on normal conditions but increases as well the size of the packets to be retransmitted in case of packet loss, causing the opposite effects. Suppose a service is sending a packet through a virtual link with an MTU of 60000 bytes. This packet is fragmented by the TUN device to fit the real link MTU. If even a single fragment is lost in the tunnel, upon not receiving the ACK, the service retransmits the entire packet again.

Figure 4.4b shows the impact of packet loss in the bandwidth between two nodes connected directly with less than 1ms of latency and an increasing amount of MTU. These results motivated the choice of a default internal MTU size of 2000, which has been used to generate the results shown in fig. 4.4a. Comparing once again the proposed tunneling approach with WireGuard, it is possible to notice that the results are proportionate in terms of bandwidth. Note that an orchestration platform like K3s still needs to use the tunneling approach of WireGuard on top of the already existing communication strategy, summing up the latencies and the overhead of both approaches. The solution proposed in this thesis merges both solutions out of the box.

4.2.5 Application deployment

Finally, to verify the capabilities of the proposed networking schema, we decided to deploy a real-world distributed application on the newer version of EdgeIO featuring the new network stack. The application is a benchmarking video analysis pipeline, proposed by Simon Bäurle in his Master's Thesis [113]. The pipeline, as shown in fig. 4.5 is composed of 4 services, a video source, a frame aggregator, an object detector, and an object tracking. Once the video source starts, using the gRPC protocol, the service sends all the frames to the Video Aggregation, which then communicates with the other two services for the tracking and detection tasks. The use of deep learning models makes these services high demanding in terms of hardware. Thus, the hosting platform must be as lightweight as possible to leave out more resources to the application.

At deployment time, the services were decorated by the system with a custom ServiceIP, allowing them to communicate freely. The platform worked as expected and showed us the strength and the weakness of the proposed approach. First of all, the low impact of the EdgeIO's components left a higher amount of resources available to the pipeline, which showed us performances around 10% better than the ones measured over K3s. It has been noticed anyway that with high video frame rates, then, with a high amount of packets flowing into the network, the CPU usage of the network components reached peaks of even 30%. Handling each L4 packet with the current routing logic may lead to a high overhead in the resources during massive packet translation operations. The universality of this approach, in this case, penalized resource consumption. Anyway, the goal of this test was to showcase that the components can actually handle real-world applications and highlight where further research and development must still be done.

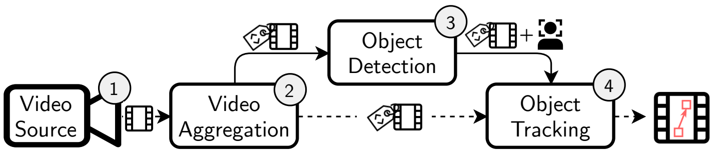


Figure 4.5: Application pipeline architecture

4.3 General remarks

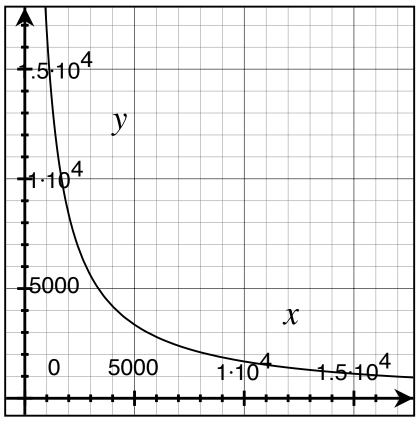
This section comments from a general perspective some of the design decisions for the proposed networking communication schema.

4.3.1 Addressing space

Each service in the proposed networking scheme uses multiple IPs; this decision improves the flexibility as discussed in chapter 3, but may pose a further limit in the number of deployable services because of the reduction in the addressing space. In particular, there is a balance between the maximum amount of services and the maximum amount of instances that form the superior limit in the number of supported deployments. Given an addressing space like `10.0.0.0/8` with 2^{24} available addresses, it is possible to calculate how many different services with different IPs are possible to deploy in such infrastructure. Assuming s services with n instances each, we can plot the equation shown in fig. 4.6a into fig. 4.6b to observe the correlation between service number and instance number within this addressing space.

$$\begin{cases} s(n+4) = 2^{24} \\ s > 0 \\ n > 0 \end{cases}$$

(a) services and instances correlation



(b) supported services (y) with respect to instance number (x)

Figure 4.6: Amount of services and instances supported by a 10.0.0.0/8 addressing space

With a small number of instances, there is almost no limit in the number of services that the platform can handle. It is possible to notice that increasing the average number of instances to a few thousand for each service limits the possible deployments in the order of thousands as well. Predictably, increasing the number of instances consistently up to ∞ makes the number of deployable services drop to zero. Anyway, the limitation in the addressing space can be easily overcome even using IPv6 as the internal addressing space for the services.

4.3.2 Fault tolerance

One of the assumptions that a developer will always make is that services are always able to communicate. Granting this assumption under the complex edge environment has proven to be a non-trivial task that needed considerable effort in terms of the design of the components and their implementation as well. The proposed fallback mechanisms decrease the chance of disruption in service-to-service interactions mitigating the impact of root, cluster, and nodes failures.

In case of temporary root failure, the clusters are almost unable to synchronize among each other. What resides in the local cache of each cluster rests unchanged. Luckily, without the support of the root, all the services can't move, and no new service can be deployed. Then, we can only limit this scenario into two cases:

1. root orchestrator failure before the cluster's consistency is reached
2. root orchestrator failure with cluster's consistency

The first scenario depicts one or more that one cluster left inconsistent, with outdated information regarding the routes and service addresses. These data can't be updated by the root and will eventually be propagated erroneously to the nodes. For each not synchronized route, the nodes won't be able to communicate with the service's instance belonging to another cluster. The communication is left entirely intact within the cluster boundaries but may experience problems when reaching out to other clusters. Note that the design of the hierarchy limits the disruption as much as possible, isolating the failures only to the non-synchronized routes. The second case of root failure is trivial, all the services will be able to communicate independently, but no new routes external to the cluster boundaries can be requested. Then, running services can communicate only to the destinations well known by the cluster.

In case of cluster's orchestrator failure, the only disadvantaged portion of the network comprises the workers attached to the same failed cluster manager. These nodes are defined "*offline*" and are not able anymore to resolve new routes. Anyway, the offline nodes are still able to use all the data belonging to their cache to perform communication with other nodes, even without the cluster manager. Unfortunately, the root can still accept deployment, migration, and scaling operations for new services during a cluster's failure, making the offline nodes unaware of the changes. Services deployed into other clusters can still be able to communicate with *offline* nodes, but *offline* nodes are not able to initiate a communication with a service that is not present in their cache information. Then, offline communication in

	Failure		
Communication	Root	Cluster	Node
Inter Cluster	$\uparrow\downarrow$	$\upharpoonleft\downarrow$	$\upharpoonleft$
Intra Cluster	$\upharpoonleft\downarrow$	$\upharpoonleft\downharpoonright$	$\upharpoonleft$

Table 4.1: Communication capabilities of a service accordingly to the failure of the root's, the cluster's or another node's components

this architecture is still possible, but only within nodes already communicating before the cluster's failure. As with the root failure, each node may not have reached the consistency in time before becoming "*offline*"; in those cases, the proxy's fallback mechanism can't find the new route until the cluster recovers. If a node already knows alternative paths in its memory, the fallback mechanism uses those.

The failure of the node's components is handled as a failure of the entire node, like if the compute machine suddenly crashed. In this case, there is nothing more to do than creating a new instance somewhere else for each of the previously deployed services.

Table 4.1 summarizes the communication issues experienced by a service, namely s_1, communicating to another one, s_2, residing in the same cluster (Inter-Cluster communication) or another (Intra-Cluster communication). The communication is obstructed by the failure of the root orchestrator, the cluster orchestrator, or the node engine where the s_2 instance belongs. The table shows with $\uparrow$ a fully working outgoing link, s_1 is able to contact s_2. $\downarrow$ shows a fully functional ingoing link, s_1 can be contacted by s_2. With $\upharpoonleft$ or $\downharpoonright$ is expressed an outgoing or ingoing communication that **may** work. In this case, the communication can be carried out only if the information in the local worker's is enough to find the destination with the correct path.

Note that during the s_2's node failure (rightmost column), s_1 can only establish the communication to another instance within the platform, if any, but can't receive any message from the s_2 instance located into the crashed node, but eventually only from others.

To mitigate the problems introduced in the above-discussed panorama, the principal solutions are:

1. *Architectural's component replication*: Enabling high availability for the root and cluster components reduces the possibility of failure. With this approach, what can't be avoided is the actual hardware and networking failure. The former can even happen in cloud environments and can only be overcome by replicating the components across different locations. Instead, the network failure must be analyzed on two levels. A network failure between the root and the cluster can be seen as a cluster's failure; it can be avoided with redundant links when possible. In contrast, the network failure between a worker and a cluster is simply a node's failure that is contemplated by design.

2. *Services replication*: This aspect is mainly delegated to the developer, but as a general rule, even in the cloud, replicating a service can highly decrease the disruption risk. As a matter of fact, this is even more important for edge deployment. The platform tries to

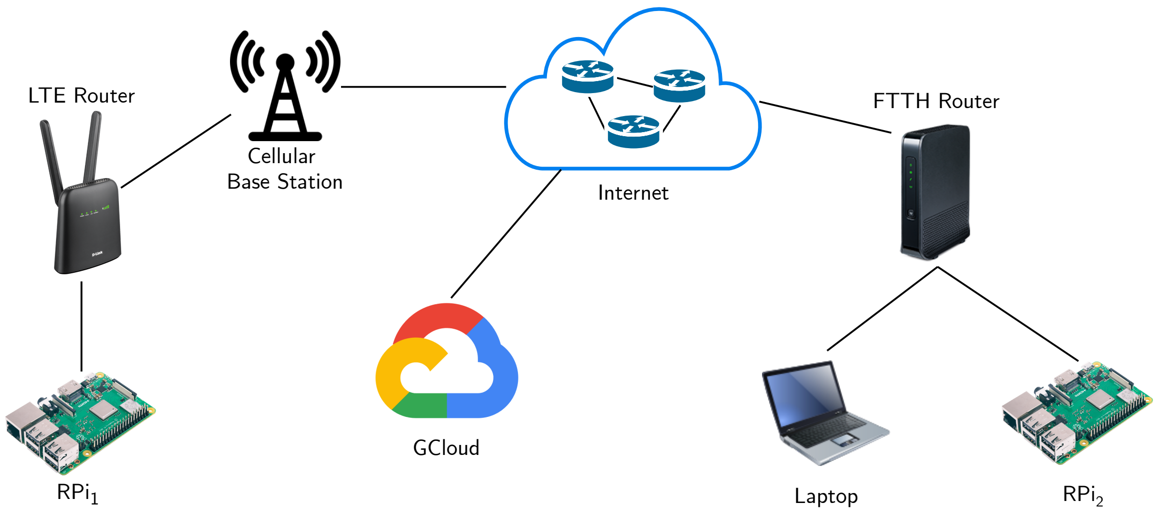


Figure 4.7: Setup used to demonstrate the platform's NAT traversing capabilities

handle the failure as much as possible, but service replication can improve the uptime when nodes move, application migrates, and network capabilities changes.

4.3.3 NAT Traversing

The NAT traversing capabilities of the proposed networking components were tested using the setup of fig. 4.7. We created a cluster installing the EdgeIO's root orchestrator on GCloud, a cluster orchestrator on a laptop, and the node engines on two different RaspberryPis, respectively, RPi_1 and RPi_2. The LTE router only exposes the custom TunnelIn port 50011. Instead, the FTTH Router exposes the set of ports from 10000 to 10010 for the cluster orchestrator and a custom 10011 port for the worker's TunnelIn. This setup enabled a python dummy service deployed on RPi_2 to contact an Nginx server deployed on RPi_1 using its round-robin ServiceIP. With a simple setup like this, it was also easy to enable a firewall rule to grant the ingoing traffic only to the known mutual networks. With more extensive arrangements and considering that the network's IP may not be static, the firewall configuration efforts to protect the devices may be more significant. This can currently be addressed as the more considerable limitation of this approach. Nevertheless, enabling direct distributed communication between the workers introduced very low latencies and higher fault tolerance, a key feature in such a heterogenic environment.

5 Conclusion

5.1 Final Remarks

The networking components presented in this thesis enabled flexibility in service-to-service communication within heterogeneous edge infrastructures. By using semantic addressing, proxies, tunneling, and a hierarchical design, all the service instances are addressable from any edge node in any condition. Thanks to the NetManager, deployed in the worker node, the overlay network is maintained efficiently, minimizing the latencies and maximizing the balancing capabilities. The lazy route resolution algorithm enables any node to maintain only the necessary data in a specific moment of time, limiting the visibility of unnecessary and protected information. The general design of the internal components empowers the possibility of bringing different virtualization technologies simultaneously, making them interact with no limitations. Moreover, each component is independently extensible, easy to upgrade and substitute. With the edge as the target environment, the components contemplate fault tolerance with fallback strategies and disaster recovery mechanisms out-of-the-box. We also demonstrated with an initial prototype the capabilities of these components in a real-world scenario. From the comparisons against other orchestration platforms, both the benefits and the downsides of this solution were analyzed. Along with outstanding traffic balancing performance, the platform offers an unmatched degree of customization and many entry points for further research. With its experimental nature, this proposal wants to explore even further modern solutions for traffic balancing and service communication while maintaining a practical character, aiming the production readiness as the final goal.

5.2 Limitations and Future Work

Thanks to the evaluation done in the prototype implementation and careful analysis of the architecture, we found some limitations in the proposed approach, which forms the base ground for further development of the platform.

First of all, the routing and proxying scheme proposed is entirely based on TCP/UDP, which is at the L4 of the OSI model. Even though this approach is very comprehensive, enabling the vast majority of the industry's most used technologies, this still leaves out some very common edge and IoT protocols like Bluetooth, ZigBee, or LoRa. Thus, point-to-point and ad-hoc networks are not supported as of now.

From the tests performed on a real-world application, we noticed that with a very high amount of traffic, the internal L4 proxy component might cause significant overhead in the system. In particular, the packet-wise translation of all the L4 frames causes multiple context changes, cache read operation, and consequently, the proxy component to consume a higher level of resources than expected. This overhead may be significant on constrained devices despite the light implementation for such components. It is clear that L4 decisions with user-space packet translation may be inefficient to support services requiring consistent traffic volumes, such as live streaming applications. For this reason, we foresee enabling proxying at layer 7, supporting a restricted pool of protocols but with a lower footprint on resource usage. Keeping both proxying mechanisms allows developers to fallback to L4 for unsupported L7 protocols while maximizing the performance for the supported ones.

Another possible limitation of the proposed approach may be the tunnel port exposure. Even if each node only has to expose a single port to enable the traffic flow, sometimes, for security reasons, companies prefer to avoid relaxing firewall rules. As part of the work envisioned for upcoming releases, we aim to introduce cluster-level tunneling for the service-to-service traffic. To overcome the current port exposure limitation, we'd like to allow a less efficient but more flexible instrument that uses the cluster as an external NAT traversing mechanism. Moreover, the tunneling approach may be unnecessary in some deployment scenarios. The overhead caused by the packet wrapping can be avoided on LAN deployments. Then, we'd like to implement tunneling as a service rather than as a default choice for internal traffic handling. We envision as well the possibility of introducing support to the CNI interface. This major upgrade would enable external networking plug-ins for the L3 underlying network management.

Lastly, for the external traffic handling, we also envision a mechanism to plug external balancers at will, enabling each application to declare at deploy time its reference balancer with its own public IP address. This approach represents a secure and scalable way to create entry points to expose services against external clients, masquerading the internal structure of the cluster and avoiding constrained devices to reveal themselves to the public internet.

List of Figures

List of Tables

Bibliography

[1] "Cisco Annual Internet Report (2018–2023)". In: (2020).

[2] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz. "The Cloud is Not Enough: Saving IoT from the Cloud". In: *7th USENIX Workshop HotCloud* (2015).

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. "Fog Computing and Its Role in the Internet of Things". In: *ACM MCC workshop* (2012).

[4] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. "Edge-Centric Computing: Vision and Challenges". In: *SIGCOMM CCR* (2015).

[5] Y. Zhao, W. Wang, Y. Li, C. Colman Meixner, M. Tornatore, and J. Zhang. "Edge Computing and Networking: A Survey on Infrastructures and Applications". In: *IEEE Access* 7 (2019), pp. 101213–101230. DOI: 10.1109/ACCESS.2019.2927538.

[6] D. King and A. Farrel. *Challenges for the Internet Routing Infrastructure Introduced by Changes in Address Semantics*. Internet-Draft draft-king-irtf-challenges-in-routing-03. Internet Engineering Task Force, June 2021. 18 pp.

[7] F. Jalali, K. Hinton, R. Ayre, T. Alpcan, and R. S. Tucker. "Fog computing may help to save energy in cloud computing". In: *IEEE Journal on Selected Areas in Communications* 34.5 (2016), pp. 1728–1739.

[8] K. Bilal, S. U. R. Malik, S. U. Khan, and A. Y. Zomaya. "Trends and challenges in cloud datacenters". In: *IEEE Cloud Computing* 1.1 (2014), pp. 10–20. DOI: 10.1109/MCC.2014.26.

[9] O. C. A. W. Group et al. "OpenFog reference architecture for fog computing". In: *OPFRA001* 20817 (2017), p. 162.

[10] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu. "Data Security and Privacy-Preserving in Edge Computing Paradigm: Survey and Open Issues". In: *IEEE Access* 6 (2018), pp. 18209–18237. DOI: 10.1109/ACCESS.2018.2820162.

[11] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[12] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. "All one needs to know about fog computing and related edge computing paradigms: A complete survey". In: *Journal of Systems Architecture* 98 (2019), pp. 289–330.

[13] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. 2012, pp. 13–16.

[14] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. "The case for vm-based cloudlets in mobile computing". In: *IEEE pervasive Computing* 8.4 (2009), pp. 14–23.

[15] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile edge computing: A survey". In: *IEEE Internet of Things Journal* 5.1 (2017), pp. 450–465.

[16] J. Lee and J. Lee. "Hierarchical mobile edge computing architecture based on context awareness". In: *Applied Sciences* 8.7 (2018), p. 1160.

[17] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi. "Design principles, architectural smells and refactorings for microservices: a multivocal review". In: *SICS Software-Intensive Cyber-Physical Systems* 35.1 (2020), pp. 3–15.

[18] A.-F. Antonescu, P. Robinson, and T. Braun. "Dynamic SLA management with forecasting using multi-objective optimization". In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE. 2013, pp. 457–463.

[19] L. Malhotra, D. Agarwal, A. Jaiswal, et al. "Virtualization in cloud computing". In: *J. Inform. Tech. Softw. Eng* 4.2 (2014), pp. 1–3.

[20] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott. "Consolidate IoT Edge Computing with Lightweight Virtualization". In: *IEEE Network* 32.1 (2018), pp. 102–111. DOI: 10.1109/MNET.2018.1700175.

[21] M. Chiesa, A. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, M. Shapira, and S. Shenker. "On the resiliency of randomized routing against multiple edge failures". In: *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[22] A. Gopalan and S. Ramasubramanian. "IP fast rerouting and disjoint multipath routing with three edge-independent spanning trees". In: *IEEE/ACM Transactions on Networking* 24.3 (2015), pp. 1336–1349.

[23] M. Kwon, Z. Dou, W. Heinzelman, T. Soyata, H. Ba, and J. Shi. "Use of network latency profiling and redundancy for cloud server selection". In: *2014 IEEE 7th International Conference on Cloud Computing*. IEEE. 2014, pp. 826–832.

[24] R. Potharaju and N. Jain. "When the network crumbles: An empirical study of cloud network failures and their impact on services". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. 2013, pp. 1–17.

[25] S. Dwarakanathan, L. Bass, and L. Zhu. "Cloud application HA using SDN to ensure QoS". In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE. 2015, pp. 1003–1007.

[26] B. Wang, Z. Qi, R. Ma, H. Guan, and A. V. Vasilakos. "A survey on data center networking for cloud computing". In: *Computer Networks* 91 (2015), pp. 528–547.

[27] *RackSolutions - How many servers does a data center have?* `https://www.racksolutions.com/news/blog/how-many-servers-does-a-data-center-have/`.

[28] S. Zafar, A. Bashir, and S. A. Chaudhry. “On implementation of DCTCP on three-tier and fat-tree data center network topologies”. In: *SpringerPlus* 5.1 (2016), pp. 1–18.

[29] M. Al-Fares, A. Loukissas, and A. Vahdat. “A scalable, commodity data center network architecture”. In: *ACM SIGCOMM computer communication review* 38.4 (2008), pp. 63–74.

[30] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. “VL2: A Scalable and Flexible Data Center Network”. In: *SIGCOMM*. Recognized as one of "the most important research results published in CS in recent years". ACM, Inc., 2009.

[31] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. “Jellyfish: Networking data centers randomly”. In: *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 2012, pp. 225–238.

[32] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. “Dcell: A Scalable and Fault-Tolerant Network Structure for Data Centers”. In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM ’08. Seattle, WA, USA: Association for Computing Machinery, 2008, 75–86. ISBN: 9781605581750. DOI: `10.1145/1402958.1402968`. URL: `https://doi.org/10.1145/1402958.1402968`.

[33] K. Bilal, S. U. R. Malik, O. Khalid, A. Hameed, E. Alvarez, V. Wijaysekara, R. Irfan, S. Shrestha, D. Dwivedy, M. Ali, et al. “A taxonomy and survey on green data center networks”. In: *Future Generation Computer Systems* 36 (2014), pp. 189–208.

[34] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. “BCube: A High Performance, Server-Centric Network Architecture for Modular Data Centers”. In: *SIGCOMM Comput. Commun. Rev.* 39.4 (Aug. 2009), 63–74. ISSN: 0146-4833. DOI: `10.1145/1594977.1592577`. URL: `https://doi.org/10.1145/1594977.1592577`.

[35] L. Gyarmati and T. A. Trinh. “Scafida: A Scale-Free Network Inspired Data Center Architecture”. In: *SIGCOMM Comput. Commun. Rev.* 40.5 (Oct. 2010), 4–12. ISSN: 0146-4833. DOI: `10.1145/1880153.1880155`. URL: `https://doi.org/10.1145/1880153.1880155`.

[36] C. Systems. “Cisco data center spine-and-leaf architecture: design overview”. In: (2016).

[37] *The case for a leaf-spine data center topology*. `https://searchdatacenter.techtarget.com/feature/The-case-for-a-leaf-spine-data-center-topology`.

[38] M. Chen, H. Jin, Y. Wen, and V. C. Leung. “Enabling technologies for future data center networking: a primer”. In: *Ieee Network* 27.4 (2013), pp. 8–15.

[39] J. Ning, T.-S. Kim, S. V. Krishnamurthy, and C. Cordeiro. “Directional neighbor discovery in 60 GHz indoor wireless networks”. In: *Performance Evaluation* 68.9 (2011), pp. 897–915.

[40] R. Jain and S. Paul. “Network virtualization and software defined networking for cloud computing: a survey”. In: *IEEE Communications Magazine* 51.11 (2013), pp. 24–31.

[41] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. Aug. 2014.

[42] P. Ferguson and G. Huston. *What is a VPN?* 1998.

[43] R. Venkateswaran. "Virtual private networks". In: *IEEE potentials* 20.1 (2001), pp. 11–15.

[44] B Patel, B Aboba, W Dixon, G Zorn, and S Booth. *RFC3193: Securing L2TP using IPsec*. 2001.

[45] I Kotuliak, P Rybár, and P Trúchly. "Performance comparison of IPsec and TLS based VPN technologies". In: *2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2011, pp. 217–221.

[46] M. Feilner. *OpenVPN: Building and integrating virtual private networks*. Packt Publishing Ltd, 2006.

[47] T. Goethals, D. Kerkhove, B. Volckaert, and F. De Turck. "Scalability evaluation of VPN technologies for secure container networking". In: *2019 15th International Conference on Network and Service Management (CNSM)*. IEEE. 2019, pp. 1–7.

[48] J. A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *NDSS*. 2017, pp. 1–12.

[49] *ZeroTier*. `https://www.zerotier.com`.

[50] *Tinc VPN*. `https://www.tinc-vpn.org`.

[51] *SoftEther VPN Project*. `https://www.softether.org`.

[52] A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo. "IP over P2P: Enabling self-configuring virtual IP networks for grid computing". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 10–pp.

[53] M. Sridharan. "NVGRE: Network virtualization using generic routing encapsulation". In: *draft-sridharan-virtualization-nvgre-00. txt* (2011).

[54] R. Kawashima and H. Matsuo. "Implementation and performance analysis of STT tunneling using vNIC offloading framework (CVSW)". In: *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*. IEEE. 2014, pp. 929–934.

[55] L. L. Peterson and B. S. Davie. *Computer networks: a systems approach*. Elsevier, 2007.

[56] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. "Resilient Overlay Networks". In: *SIGOPS Oper. Syst. Rev.* 35.5 (Oct. 2001), 131–145. ISSN: 0163-5980. DOI: `10.1145/502059.502048`. URL: `https://doi.org/10.1145/502059.502048`.

[57] *What is software-defined networking (SDN)? - Cloudflare*. `https://www.cloudflare.com/learning/network-layer/what-is-sdn/`.

[58] S. Tomovic, M. Pejanovic-Djurisic, and I. Radusinovic. "SDN based mobile networks: Concepts and benefits". In: *Wireless Personal Communications* 78.3 (2014), pp. 1629–1644.

[59] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM computer communication review* 38.2 (2008), pp. 69–74.

[60] E. Haleplidis, D. Joachimpillai, J. H. Salim, D. Lopez, J. Martin, K. Pentikousis, S. Denazis, and O. Koufopavlou. "ForCES applicability to SDN-enhanced NFV". In: *2014 Third European Workshop on Software Defined Networks*. IEEE. 2014, pp. 43–48.

[61] S. Badotra and J. Singh. "Open Daylight as a Controller for Software Defined Networking." In: *International Journal of Advanced Research in Computer Science* 8.5 (2017).

[62] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. "A survey of software-defined networking: Past, present, and future of programmable networks". In: *IEEE Communications surveys & tutorials* 16.3 (2014), pp. 1617–1634.

[63] X. Ren, G. S. Aujla, A. Jindal, R. S. Batth, and P. Zhang. "Adaptive recovery mechanism for SDN controllers in Edge-Cloud supported FinTech applications". In: *IEEE Internet of Things Journal* (2021).

[64] X. Li, D. Li, J. Wan, C. Liu, and M. Imran. "Adaptive transmission optimization in SDN-based industrial Internet of Things with edge computing". In: *IEEE Internet of Things Journal* 5.3 (2018), pp. 1351–1360.

[65] R. Vilalta, A. Mayoral, D. Pubill, R. Casellas, R. Martínez, J. Serra, C. Verikoukis, and R. Muñoz. "End-to-end SDN orchestration of IoT services using an SDN/NFV-enabled edge node". In: *Optical fiber communication conference*. Optical Society of America. 2016, W2A–42.

[66] P. P. Ray and N. Kumar. "SDN/NFV architectures for edge-cloud oriented IoT: A systematic review". In: *Computer Communications* (2021).

[67] M. Kuzniar, P. Peresini, and D. Kostic. *What you need to know about SDN control and data planes*. Tech. rep. 2014.

[68] Z. Benomar, D. Bruneo, S. Distefano, K. Elbaamrani, N. Idboufker, F. Longo, G. Merlino, and A. Puliafito. "Extending Openstack for Cloud-Based Networking at the Edge". In: *2018 IEEE iThings and IEEE Green GreenCom and IEEE CPSCom and IEEE Smart Data*. 2018, pp. 162–169.

[69] S. Hamadi, I. Snaiki, and O. Cherkaoui. "Fast path acceleration for open vSwitch in overlay networks". In: *2014 Global Information Infrastructure and Networking Symposium (GIIS)*. 2014, pp. 1–5. DOI: `10.1109/GIIS.2014.6934286`.

[70] S. Nunna, A. Kousaridas, M. Ibrahim, M. Dillinger, C. Thuemmler, H. Feussner, and A. Schneider. "Enabling Real-Time Context-Aware Collaboration through 5G and Mobile Edge Computing". In: *2015 12th International Conference on Information Technology - New Generations*. 2015, pp. 601–605. DOI: `10.1109/ITNG.2015.155`.

[71] S. Forti, F. Paganelli, and A. Brogi. "Probabilistic QoS-aware placement of VNF chains at the edge". In: *Theory and Practice of Logic Programming* (2019), pp. 1–36.

[72] Y. Li, Z. Han, S. Gu, G. Zhuang, and F. Li. “Dyncast: Use Dynamic Anycast to Facilitate Service Semantics Embedded in IP address”. In: *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*. 2021, pp. 1–8. DOI: `10.1109/HPSR52026.2021.9481819`.

[73] Y. Liu, Z. Zeng, X. Liu, X. Zhu, and M. Z. A. Bhuiyan. “A novel load balancing and low response delay framework for edge-cloud network based on SDN”. In: *IEEE Internet of Things Journal* 7.7 (2019), pp. 5922–5933.

[74] M. T. Beck, M. Werner, S. Feld, and S Schimper. “Mobile edge computing: A taxonomy”. In: *Proc. of the Sixth International Conference on Advances in Future Internet*. Citeseer. 2014, pp. 48–55.

[75] W. Zhang, A. Sharma, and T. Wood. “EdgeBalance: Model-Based Load Balancing for Network Edge Data Planes”. In: *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020. URL: `https://www.usenix.org/conference/hotedge20/presentation/zhang`.

[76] J. Wu, Z. Zhang, Y. Hong, and Y. Wen. “Cloud radio access network (C-RAN): a primer”. In: *IEEE network* 29.1 (2015), pp. 35–41.

[77] S. C. Ergen. “ZigBee/IEEE 802.15. 4 Summary”. In: *UC Berkeley, September* 10.17 (2004), p. 11.

[78] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv. “Edge Computing Security: State of the Art and Challenges”. In: *Proceedings of the IEEE* 107.8 (2019), pp. 1608–1631. DOI: `10.1109/JPROC.2019.2918437`.

[79] B. Nguyen, N. Choi, M. Thottan, and J. Van der Merwe. “SIMECA: SDN-based IoT Mobile Edge Cloud Architecture”. In: *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2017, pp. 503–509. DOI: `10.23919/INM.2017.7987319`.

[80] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. “Extend cloud to edge with KubeEdge”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 373–377.

[81] A. Bala and I. Chana. “Fault tolerance-challenges, techniques and implementation in cloud computing”. In: *International Journal of Computer Science Issues (IJCSI)* 9.1 (2012), p. 288.

[82] T. Bourke. *Server load balancing - DNS Load Balancing*. " O’Reilly Media, Inc.", 2001, pp. 4–7.

[83] V. Cardellini, M. Colajanni, and P. Yu. “Dynamic load balancing on Web-server systems”. In: *IEEE Internet Computing* 3.3 (1999), pp. 28–39. DOI: `10.1109/4236.769420`.

[84] T. Bourke. *Server load balancing - Replication*. " O’Reilly Media, Inc.", 2001, pp. 17–21.

[85] *Introduction to modern network load balancing and proxying*. `https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236`.

[86] *Kubernetes*. `https://kubernetes.io`.

[87] *Kubernetes - Workloads resources*. `https://kubernetes.io/docs/concepts/workloads/`.

[88] *CNI - the Container Network Interface*. https://github.com/containernetworking/cni.

[89] *Flannel - a simple and easy way to configure a layer 3 network fabric*. https://github.com/flannel-io/flannel.

[90] *Calico - an open source networking and network security solution for containers*. https://docs.projectcalico.org/about/about-calico.

[91] *ACI - Cisco*. https://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html.

[92] A. Jeffery, H. Howard, and R. Mortier. "Rearchitecting Kubernetes for the Edge". In: *4th ACM EdgeSys* (2021).

[93] *KubeEdge - an open source system for extending native containerized application orchestration capabilities to hosts at Edge*. https://kubeedge.io/.

[94] *Skupper*. https://skupper.io/.

[95] *Skupper - overview*. https://skupper.io/docs/overview/index.html.

[96] *K3s - Lightweight Kubernetes*. https://k3s.io.

[97] *WireGuard - Fast, Modern, Secure VPN Tunnel*. https://www.wireguard.com.

[98] *IoFog - Bring your own edge*. https://iofog.org.

[99] *Dapr - Simplify cloud-native application development*. https://dapr.io.

[100] *Dapr - Service Invocation*. https://docs.dapr.io/developing-applications/building-blocks/service-invocation/service-invocation-overview/.

[101] *EdgeIO - GitHub*. https://github.com/edgeIO.

[102] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: Library operating systems for the cloud". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 461–472.

[103] "IoT Developer Survey 2020". In: *Eclipse Foundation* (2020).

[104] *Accessing Clusters - Kubernetes*. https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/.

[105] *ReplicaSet - Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[106] *Redirections in HTTP*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirections.

[107] *Kubernetes - External Load Balancer*. https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/.

[108] *Gorilla web toolkit*. https://www.gorillatoolkit.org.

[109] *water - a native Go library for TUN/TAP interfaces*. https://github.com/songgao/water.

[110] S. Ariyapperuma and C. J. Mitchell. "Security vulnerabilities in DNS and DNSSEC". In: *The Second International Conference on Availability, Reliability and Security (ARES'07)*. IEEE. 2007, pp. 335–342.

[111] *MicroK8s - High availability K8s*. `https://microk8s.io`.

[112] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, and J. Kangasharju. "Pruning Edge Research with Latency Shears". In: *HotNets* (2020).

[113] *Benchmarking Pipeline - GitLab*. `https://gitlab.lrz.de/cm/2021-simon-masterthesi`.