Universidade de Aveiro
2023
Departamento de
Electrónica, Telecomunicações e Informática,

Vasco Regal Sousa

Multiple Client WireGuard Based Private and Secure Overlay Network

DOCUMENTO PROVISÓRIO

"An idiot admires complexity,
a genius admires simplicity."

— Terry A. Davis

o júri / the jury

presidente / president	**ABC** Professor Catedrático da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)
vogais / examiners committee	**DEF** Professor Catedrático da Universidade de Aveiro (orientador)
	GHI Professor associado da Universidade J (co-orientador)
	KLM Professor Catedrático da Universidade N

agradecimentos / acknowledgements

Ágradecimento especial aos meus gatos

Desejo também pedir desculpa a todos que tiveram de suportar o meu desinteresse pelas tarefas mundanas do dia-a-dia

Abstract

An overlay network is a group of computational nodes that communicate with each other through a logical channel, built on top of another network. Nodes in an overlay network, which are generally end systems, run Internet applications capable of performing more complex operations besides just forwarding and switching traffic. As such, an overlay network can apply routing rules and data encapsulation to create a custom protocol running on top of the links of another network. This document presents a secure and private overlay network solution based on WireGuard, deployed over University of Aveiro's network, which employs very restrictive firewall policies. This solution allows clients, namely the autonomous robots residing in the Intelligent Robotics and Systems Laboratory research unit, to establish Peer to Peer (P2P) communications with each other, regardless of their physical location within the campus, by encapsulating and forwarding encrypted WireGuard traffic to a dedicated relay server as Hyper-Text Transfer Protocol (HTTP) through pre-established channels. This not only aids developers as they can interact directly with the robots through their personal machines, but also makes it possible to deploy solutions capable of communicating through any of the campus' buildings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Network security has become a topic of growing interest in any information system, where companies strive to ensure their communications follow principles of integrity and confidentiality while minimizing attack vectors that could compromise services and data. With such goals in mind, network topologies are subjected to traffic constraints and security mechanisms to secure their systems.

Such is the case in the University of Aveiro (UA), where the network, although covering most of the campus' area, enforces several constraining mechanisms that prevent, for example, the establishment of direct P2P connections between two clients without additional firewall configurations.

The Intelligent Robotics and Systems Laboratory (IRIS-Lab) is a research unit operating in UA's premises which develops projects using autonomous mobile robots, capable of communicating through a wireless network. Currently, the robots are confined to the laboratory's internal network, since, as mentioned above, UA's highly restrictive network prevents the robots from communicating directly.

Overcoming these limitations would be extremely valuable for IRIS-Lab's developments. In fact, allowing robots to communicate directly in a P2P communication would not only enable solutions across multiple buildings but also aid researchers during development, as they would be able to interact with the robots directly through their personal machines.

1.2 Objectives

This dissertation aims to implement a private overlay network manager to be used exclusively by clients connected to UA's network. The concept of an overlay network entails the creation of a communication layer built on top of an already existing network.

In the IRIS-Lab scenario, the overlay protocol should provide operations to achieve a secure, private communication between a group of robots, regardless of their physical location within the campus, requiring no additional firewall rules. Moreover, the authentication and connection to a desired overlay network by the robots must be a seamless operation, requiring little to no manual configuration.

To reinforce privacy and confidentiality, this solution should be hosted entirely within UA's premises, preferably using open source tools.

Therefore, the objectives for this dissertation can be summarized as (i) enabling secure P2P communications between clients connected to UA's network, (ii) automation of client deployment, authentication and configuration mechanisms, creating an abstraction layer for the usual robot operations and (iii) ensure communication overhead in the overlays is suitable for IRIS-Lab's projects requirements.

1.3 Document Structure

This document starts by overviewing overlay and Virtual Private Networks (VPNs) concepts and their respective technology state-of-the-art. Then, based on this research, a secure overlay network manager solution is proposed, covering both its architecture specification and implementation methodology. Finally, a set of conducted experiments is presented, in order to validate and collect insights regarding the solution's performance, security and suitability for IRIS-Lab's scope.

Chapter 2

Background and State of the Art

"O caos é uma ordem por decifrar"

— José Saramago

2.1 Overlay Networks

In the last few decades, the Internet has been subjected to an exponential growth, both in number of users and online devices. To answer this increasing demand and support aspects such as mobility and scalability, Internet applications have diverged from classic distributed systems to more complex network topologies, creating an extremely heterogeneous environment. In such a non-patterned landscape, overlay networks have emerged as a topic of growing interest [1], as conducted research on the matter attempts to create networking solutions capable of addressing the adversities imposed by the modern day Internet [2, 3]. This section explores the fundamental principles of overlay networks and how its abstraction layer is able to produce a topology with the potential to bring a logical order to the chaotic network architecture of the Internet.

2.1.1 Definition and Motivation

By definition, an overlay network is a logical network implemented on top of the links of another existing physical network [4]. In other words, nodes (also called peers) in an overlay network, which also exist in the underlay network, implement their own application-defined routing and datagram processing behaviour. Hence, the Internet application running in the nodes is responsible for the creation and management of the logical links that form the overlay network. Figure 2.1 illustrates this concept.

As the nodes in an overlay network are systems running Internet applications, they are generally capable of performing more computational demanding operations than simply forwarding and switching traffic. Routing traffic through an overlay node allows any application-defined forwarding and data encapsulation operations to be applied to the datagrams, creating a custom communication protocol running on top of the underlay network.

2.1.2 Security in Overlay Networks

The Internet is built on public infrastructure users generally have no control over. Public routers, relay nodes, servers and even physical links always carry the risk of traffic eaves-

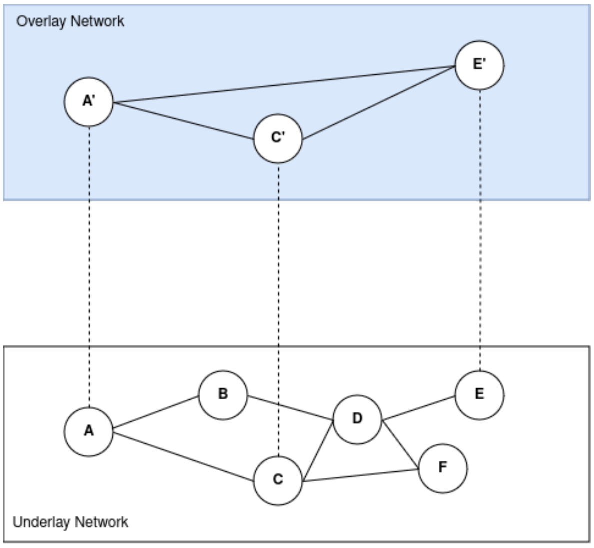


Figure 2.1: Concept of a very basic overlay network. Nodes A, C and E create logical links with each other, forming an overlay network

dropping and tampering by untrusted entities. Although overlay networks are able to create a logical communication structure, traffic can still be routed through this susceptible infrastructure before reaching its destination node. Therefore, to ensure the confidentiality, privacy and integrity of connections established through an insecure medium, data should be properly encrypted and authenticated.

In fact, if the Internet application running in the nodes of an overlay network has no such security considerations, communication will always be subjected to network attack vectors such as Man in the Middle (MitM) and Denial of Service (DoS) attacks and traffic snooping.

To add a confidentiality layer to communication, overlay networks employ security mechanisms which encapsulate datagrams, ensuring only designated overlay nodes can interpret the transmitted information. A common way to achieve this functionality is the use of VPNs.

2.1.3 Virtual Private Networks

Conceptually, a VPN is a virtual network that provides functionalities securing the transmission of data between any two endpoints [5]. VPNs have been a widely used technology for decades, establishing itself as the backbone to secure communications over the Internet.

The following sections aim to analyse how VPN services have evolved, from its traditional design to more recent paradigms, which are able to tackle complexity and scalability issues. Then, it presents an individual overview of notable VPN providers and compares their respective advantages, limitations, trade-offs and suitability for the scope of this solution. This state of the art technological survey aims to pickup on similar works [6, 7], which examine established protocols such as OpenVPN and IPSec, while additionally reviewing modern protocols, focusing on WireGuard. This exploration serves as the foundation for understanding which protocol is the most suitable to secure communications for the scenario at hand.

VPNs on NAT Networks

A Network Address Translator (NAT) is a networking mechanism responsible for translating Internet Protocol (IP) addresses in private networks to public addresses when packets sent from a private network are routed to the public Internet. In the context of VPN communications, this process can prove to be a major constraint, not only due to NAT's tampering of IP packets' fields, namely destination and source addresses, which could potentially compromise its integrity in the eyes of a VPN protocol, but also regarding the dynamically changing public IP addresses which NAT decides to translate private addresses to. In other words, without additional tools, a host in a private network has no knowledge regarding which public IP it will be assigned.

In fact, it is very likely that devices connected to the Internet reside in a network behind both NAT mechanisms and Firewall rules, with no open ports. Also, believing nodes will have a consistent static IP is a very naive assumption, especially when considering mobile devices. NAT Traversal is a networking technique that enables the establishing and maintaining (by keeping NAT holes open) of P2P connections between two peers, no matter what's standing between them, making communication possible without the need for firewall configurations or public-facing open ports. There's no one solution to achieve this functionality. In fact, there are various developments effectively implementing a NAT Traversal solution, such as ICE [8] and STUN [9]. Hence, each VPN service can have its own way of supporting NAT Traversal. Each case is explored separately.

2.2 VPN Providers

2.2.1 IPSec

IPSec refers to an aggregation of layer 3 protocols that work together to create a security extension to the IP protocol by adding packet encryption and authentication. Conceptually, IPSec presents two main dimensions: the protocol defining the transmitted packets' format, when security mechanisms are applied to them, and the protocol defining how parties in a communication negotiate encryption parameters.

Communication in an IPSec connection is managed according to Security Associations (SAs). A SA is an unidirectional set of rules and parameters specifying the necessary information for secure communication to take place [10]. Here, unidirectional means a SA can only be associated with either inbound or outbound traffic, but never with both. Hence, an IPSec bidirectional association implies the establishment of two SAs: one for incoming packets and one for outgoing. SAs specify which security mechanism to use - either Authentication Header (AH) or Encapsulating Security Payload (ESP) - and are identified by a numeric value, the Security Parameter Index (SPI). Although SAs can be manually installed in routers, gateways or machines, it becomes impractical as more clients appear. Internet Key Exchange (IKE) [11] is a negotiation protocol that tackles the problems associated with manual SA installation. In fact, IKE allows the negotiation of SA pairs between any two machines through the use of asymmetric keys or shared secrets.

Transport and Tunnel modes

IPSec supports two distinct modes of functionality: transport and tunnel [10], which differ in the way traffic is dealt with and processed. In the context of VPNs, tunnel mode presents the most desirable characteristics. First, tunnel mode encapsulates the original IP packet, allowing the use of private IP addresses as source or destination. Tunnel mode creates the concept of an "outer" and "inner" IP header. The former contains the addresses of the IPSec peers, while the latter contains the real source and destination addresses. Moreover, this very same encapsulation adds confidentiality to the original addresses.

Transport mode requires fewer computational resources and, consequently, carries less protocol overhead. It does not, however, provide much security compared to tunnel mode, so, to secure data in a communication, tunnel mode's total protection and confidentiality of the encapsulated IP packet carry much more valuable functionalities.

Authentication Header

AH is a protocol in the IPSec suite providing data origin validation and data integrity consisting in the generation of a checksum via a digest algorithm [12]. Additionally, besides the actual message under integrity check, two other parameters are used under the AH mechanism. First, to ensure the message was sent from a valid origin, AH includes a secret shared key. Then, to ensure replay protection, it also includes a sequence number. This last feature is achieved with the sender incrementing a sequence integer whenever an outgoing message is processed.

AH, as the name suggests, operates by attaching a header to the IP packets, containing the message's SPI, its sequence number, and the Integrity Check Value (ICV) value. This last field is then verified by receivers, which calculate the packet's ICV on their end. The packet is only considered valid if there's a match between the sender and receiver's ICV.

Where this header is inserted depends on the mode in which IPSec is running. In transport mode, the AH appears after the IP header and before any next layer protocol or other IPSec headers. As for tunnel mode, the AH is injected right after the outer IP header.

To calculate the ICV, the AH requires the value of the source and destination addresses, which raises an incompatibility when faced with networks operating with NAT mechanisms [13].

Encapsulating Security Payload

The ESP protocol also offers authentication, integrity and replay protection mechanisms. It differs from AH by also providing encryption functionalities, where peers in a communication use a shared key for cryptographic operations. Analogous to the previous protocol, the ESP's header location differs in different IPSec modes. In transport mode, the header is inserted right after the IP header of the original packet. Also, in this mode, since the original IP header is not encrypted, endpoint addresses are visible and might be exposed. As for tunnel mode, a new IP header is created, followed by the ESP header.

Tunnel mode ESP is the most commonly used IPSec mode. This setup not only offers original IP address encryption, concealing source and destination addresses, but also supports the addition of padding to packets, hampering cipher analysis techniques. Moreover, it can be made compatible with NAT and employ NAT-traversal techniques [14, 15].

2.2.2 OpenVPN

OpenVPN [16] is yet another open-source VPN provider, known for its portability among the most common operating systems due to its user-space implementation. OpenVPN uses established technologies, such as Secure Sockets Layer (SSL) and asymmetric keys for negotiation and authentication and IPSec's ESP protocol, explored in the previous section, over UDP or TCP for data encryption.

TUN and TAP interfaces

OpenVPN's virtual interfaces, which process outgoing and incoming packets, have two distinct types: TUN (short for internet TUNnel) and TAP (short for internet TAP). Both devices work quite similarly, as both simulate P2P communications. They differ on the level of operation, as TAP operates at the Ethernet level. In short, TUN allows the instantiation of IP tunnels, while TAP instantiates Ethernet tunnels.

OpenVPN flow

When a client sends a packet through a TUN interface, it gets redirected to a local OpenVPN server. Here, the server performs an ESP transformation and routes the IP packet to the destination address, through the "real" network interfaces.

Similarly, when receiving a packet, the OpenVPN server will perform decipherment and validation operations on it, and, if the IP packet proves to be valid, it is sent to the TUN interface.

This process is analogous when dealing with TAP devices, differing, as mentioned before, at the protocol level.

2.2.3 WireGuard

WireGuard [17] is an open-source User Datagram Protocol (UDP)-only layer 3 network tunnel implemented as a kernel virtual network interface. WireGuard offers both a robust cryptographic suite and transparent session management, based on the fundamental principle of secure tunnels: peers in a WireGuard communication are registered as an association between a public key (analogous to the OpenSSH keys mechanism) and a tunnel source IP address.

One of WireGuard's selling points is its simplicity. In fact, compared to similar protocols, which generally support a wide range of cryptographic suites, WireGuard settles for a singular one. Although one may consider the lack of cipher agility as a disadvantage, this approach minimizes protocol complexity and increases security robustness by avoiding vulnerabilities commonly originating from such protocol negotiation [18].

Routing

Peers in a WireGuard communication maintain a data structure containing their own identification (both public and private keys) and interface listening port. Then, for each known peer, an entry is present containing an association between a public key and a list of allowed source IPs.

This structure is queried on communication initiation. When a peer wants to start a connection, the structure is consulted, and, based on the destination address, the destination peer's public key is retrieved to ecnrypt an initial handshake message for session key exchange. As for data exchanging, after encryption operations (using the session keys), the structure is used to verify the validity of the packet's source address, which, in other words, means checking if there's a match between the source address and the allowed addresses present on the routing structure for that peer.

Optionally, WireGuard peers can configure one additional field, an internet endpoint, defining the listening address where packets should be sent. If not defined, the incoming packet's source address is used.

Cipher Suite

As mentioned before, WireGuard offers a single cipher suite for encryption and authentication mechanisms in its protocol. The peers' pre-shared keys are Curve25519 points, an implementation of an elliptic-curve-Diffie-Hellman function, characterized by its strong conjectured security level - presenting the same security standards as other algorithms in public key cryptography - while achieving record computational speeds [19]. Additionally, WireGuard's Curve25519 computation uses a verified implementation [20], which contributes further to the robustness of the protocol's cryptography.

Before any encrypted data exchange actually happens, WireGuard enforces a handshake for symmetric session key exchange. The messages involved in this handshake process follow a variation of the Noise [21] protocol, which is essentially a state machine controlled by a set of variables maintained by each peer in the negotiation.

Regarding payload data cryptography, a WireGuard message's data is encrypted with session keys and a nonce counter, using ChaCha20Poly1305, a Salsa20 variation [22]. The ChaCha cryptographic family offers robust resistance to crypto-analytic methods [23], without sacrificing its state-of-the-art performance.

Security

On top of its robust cryptographic specification, WireGuard includes in its design a set of mechanisms to further enhance protocol security and integrity.

In fact, WireGuard presents itself as a silent protocol. In other words, a WireGuard peer is essentially invisible when communication is attempted by an illegitimate party. Packets coming from an unknown source are just dropped, with no leak of information to the sender.

Additionally, a cookie system is implemented in an attempt to mitigate Distributed Denial Of Service (DDOS) attacks. Since, to determine the authenticity of a handshake message, a Curve25519 multiplication must be computed, a CPU intensive operation, a CPU-exhaustion attack vector could be exploited. Cookies are introduced as a response message to handshake initiation. These cookie messages are used as a peer response when under high CPU load, which is then in turn attached to the sender's following messages, allowing the requested handshake to proceed when the overloaded peer has available resources to continue.

Basic WireGuard Configuration

Connecting two peers in a WireGuard communication can be done with minimal configuration. After the generation of an asymmetric key pair and the setup of a WireGuard interface, it is only required to add the other peer to the routing table with its public key, allowed IPs and, optionally, its internet endpoint. After both peers configure each other, the tunnel is established and packets can be transmitted through the WireGuard interface. In

	Peer A	Peer B
Private Key	gIb/+...+uF2Y=	aFov...G3l0=
Public Key	FeQI...jHgE=	sg0X...7kVA=
Internet Endpoint	192.168.100.4	192.168.100.5
WireGuard Port	51820	51820

Table 2.1: Nodes to be configured with a WireGuard tunnel

```
# Peer A - interface setup
$ ip link add wg0 type WireGuard
$ ip addr add 10.0.0.1/24 dev wg0
$ wg set wg0 private-key ./private
$ ip link set wg0 up

# Adding peer B to known peers
$ wg set wg0 peer sg0X...7kVA=
    allowed-ips 10.0.0.2/32
    endpoint 192.168.100.5:51820
$
```

```
# Peer B - interface setup
$ ip link add wg0 type WireGuard
$ ip addr add 10.0.0.2/24 dev wg0
$ wg set wg0 private-key ./private
$ ip link set wg0 up

# Adding peer A to known peers
$ wg set wg0 peer FeQI...jHgE=
    allowed-ips 10.0.0.1/32
    endpoint 192.168.100.4:51820
$
```

Figure 2.2: Basic WireGuard Communication Between Two Peers

a practical scenario, given two peers, A and B, with pre-generated keys and internet interfaces, presented on table 2.1, the CLI steps to setup a minimal WireGuard communication, as specified in the official WireGuard documentation are presented in figure 2.2.

Limitations

Although WireGuard proves to be a robust, performant and maintainable protocol for encrypted communication, it still presents some complexity regarding administration agility and scalability, since it (intentionally) lacks a coordination entity. New clients added to a stand-alone WireGuard network imply the manual reconfiguration of every other peer already present, a process with added complexity and prone to errors, if carried out manually.

There are, however, several software products implementing such an orchestrator, which provide functionalities to easily manage WireGuard peers and respective tunnels, such as Tailscale, Netbird or Netmaker. These solutions are explored in further sections.

2.2.4 Protocol Comparison

Security

Regarding security, although all analysed providers are capable of robustly authenticating and encrypting data, recent verifications of WireGuard's cryptographic suite [24, 25] conclude WireGuard offers a smaller attack surface, when compared with OpenVPN and IPSec.

	IPSec	OpenVPN	WireGuard
Security	Robust, but complex	Generally robust, requires proper configuration	Modern, recently verified cryptography
Usability	Requires expertise	Requires expertise	Minimal configuration
Performance	Good, but underperforms compared to WireGuard	Generally implies higher latencies	Overall, outperforms other protocols
NAT-Traversal	Supported, requires configuration	Supported, requires configuration	Seamless integration

Table 2.2: VPN protocol's performance and features comparison

Performance

The concept of performance in VPN applications refers to the impact the protocol has on communications. This dimension can be empirically measured, by analysing metrics like Round Trip Time (RTT), throughput and jitter. CPU utilization is also a valuable metric, as demanding operations can also take a toll on overall network performance, since processing traffic might stall due to the CPU overloading.

The performance claims on [17], which compares WireGuard to other discussed technologies, present results in favour of WireGuard in such performance tests. This conclusion is backed by more extensive research [26, 27], where communication is tested in a wide range of different network scenarios and CPU architectures.

WireGuard's kernel implementation and efficient multi-threading usage contribute greatly to such performance benchmarks. Overall, WireGuard outperforms IPSec and OpenVPN with no notable trade-offs.

Usability

Usability in VPN protocols refers to the ease of use regarding service configuration and management. IPSec's configuration can be quite complex, due to its cryptographic directives and protocol negotiation. Also, the several different modes IPSec can be configured to run in might prove overwhelming for unfamiliar users. OpenVPN's GUI provides a user-friendly platform to easily administrate the software, still requiring, however, a considerable degree of technical expertise. WireGuard, however, is able to combine protocol simplicity, as it enforces a single cipher suite, with straightforward configuration directives. In fact, as we've seen in Section 2.2.3, configuring WireGuard peers requires no knowledge of the protocol's specification.

Table 2.2 summarizes the above analysis.

2.2.5 Summary

This section presents the fundamentals of VPNs and how they are able to ensure integrity and confidentiality to communication through insecure mediums. Then, some of the most notable VPN providers were reviewed and compared, regarding functionalities and overall performance. The result of this analysis suggests WireGuard as the most promising VPN protocol, due to its ease of management and configuration, security mechanisms and verified cryptography.

2.3 WireGuard Coordination

In Section 2.2.3, the lack of a coordination entity on stand-alone WireGuard topologies was raised as a possible limitation. In fact, manually reconfiguring peers in a standalone WireGuard network is a process that scales terribly. Moreover, on highly restrictive networks, creating direct WireGuard tunnels proves to not be as straightforward.

With such constraints in mind, this section explores applications and implementations of coordination platforms built, or with the potential to be built, on top of WireGuard, aiming to create a seamless peer orchestration and configuration process, minimizing human intervention.

2.3.1 Tailscale

Tailscale [28] is a VPN service operating with a Golang user-space WireGuard variant as its data plane. Tailscale operates under a hybrid VPN model. Although a central entity is present, initially resembling hub-and-spoke topologies, its main purpose is to aid clients in configuring themselves and managing connections, serving nearly no traffic.

In addition to orchestrating the configuration and establishment of WireGuard tunnels, Tailscale also employs a set of mechanisms capable of addressing network constraints which would hinder the process of directly creating WireGuard connections. In fact, Tailscale praises itself as being able to make any two clients communicate, regardless of the network structure and security policies standing in-between them. For the scope of this project, where very few is known about UA's network architecture, the ability to bypass such constraints would prove extremely valuable.

This section explores how Tailscale is able to manage the WireGuard links between its clients, creating a WireGuard encrypted overlay network while dealing with security constraints preventing such communications to take place.

Architecture

Tailscale's central entity, referred to as the **coordination server**, functions as a shared repository of peer information, used by clients to retrieve data regarding other nodes and establish on-demand WireGuard connections among each other.

This approach differs from traditional hub-and-spoke since the coordination server carries nearly no traffic, it only serves encryption keys and peer information. Tailscale's architecture provides the best of both worlds, benefiting from the advantages of a centralized entity facilitating dynamic reconfigurations and peer discovery without bottlenecking WireGuard's data exchanging performance.

In practical terms, a Tailscale client will store, in the coordination server, its own public key and where it can currently be found. Then, it downloads a list of public keys and addresses that have been stored on the server previously by other clients. With this information, the client node is able to configure its Tailscale interface and establish direct communications with any other node in its domain.

Overcoming restrictive networks

There are, however, some network security configurations which prevent WireGuard tunnels from being established. For example, firewalls blocking UDP traffic entirely prevent the creation of direct WireGuard tunnels between two peers. In addition to orchestrating WireGuard peers, Tailscale is also able to overcome such constrained networks.

Regarding stateful firewalls, where, generally, inbound traffic for a given source address on a non-open port is only accepted if the firewall has recently seen an outgoing packet with such *ip:port* as destination, Tailscale keeps, in its coordination server, the public address of each node in its network. With this information, if both peers send an outgoing packet to each other at approximately the same time (a time delta inferior to the firewall's cache expiration), then the firewalls at each end will be expecting the reception of packets from the opposite peer. Hence, packets can flow bidirectionally and a P2P communication is established. To ensure this synchronism of attempting communication at approximate times, Tailscale uses its coordination server and Designated Encrypted Relay for Packets (DERP) servers (explored further in the following paragraphs) as a side channel.

Although this procedure is quite effective, in networks with NAT mechanisms, where source and destination addresses are tampered with, this process is not as straightforward, since peers don't know the public addresses NAT will translate their private addresses to. The Session Traversal Utilities for NAT (STUN) protocol offers aid in performing NAT-Traversal operations [9] and its supported by Tailscale to overcome this problem. For a peer to discover and store in the coordination server its own public *ip:port*, it first sends a UDP message requesting an IP binding to a STUN server. Upon receiving this request, the STUN server, which can see the public source address of the packet it received, or, by other words, the address produced by NAT, replies this value to the peer.

There are, however, some NAT devices that create a completely different public address mapping for each different destination a machine communicates with, which hinders the above address discovery process, since the address the STUN server sees might not match when communicating with a different destination. Such NAT devices are classified as Endpoint-Dependent Mapping (EDM) (in opposition to Endpoint-Independent Mapping (EIM)) [29].

Networks employing EDM NAT devices and/or really strict firewall rules, render these traversal techniques useless. To enable P2P communications in such scenarios, Tailscale also provides a network of DERP servers, which are responsible for relaying packets over HTTP. A Tailscale client is able to forward its encrypted packets to one of such DERP servers. Since a client's private key never actually leaves its node, DERP servers can't decrypt the traffic being relayed, performing only redirection of already encrypted data. Tailscale offers a geographically distributed fleet of such servers. However, since traffic is encapsulated as an HTTP stream, relayed communications always imply a degradation of network performance.

Hence, Tailscale connections can be of two types: either **direct** or **relayed**. Tailscale will always attempt to create direct connections. In fact, if NAT-traversal succeeds on both endpoints and there are no restrictions to UDP traffic, a direct WireGuard tunnel can be established. However, on networks employing EDM NAT mappings and/or UDP blocking firewalls, direct connections cannot be formed. That's where relayed connections are created. Clients on relayed connections encapsulate WireGuard UDP packets as Transmission Control Protocol (TCP) streams and forwards them through HTTP to the closest DERP server available, which in turn delivers them to its destination.

Headscale

While Tailscale's client is open source, its coordination server isn't. There is, however, an open-source, self-hosted alternative to Tailscale's control server, Headscale. Headscale

[30] provides a narrow-scope implementation (with a single Tailscale private network) of the aforementioned coordination server, which, in the authors' words, is mostly suitable for personal use and small organizations. Nodes running Tailscale clients can opt to specify the location of the control server, which can be the address of a running self-hosted Headscale instance.

Headscale also supports the self-hosting of DERP and STUN servers. This is useful as it allows the self-hosting of a complete Tailscale solution.

2.4 University of Aveiro Network

Due to security and privacy concerns, the specification of UA's network topology is not publicly available nor is it made available for this dissertation. As such, it is perceived as a black box. There are, however, a few reachable conclusions drawn from observing its behaviour.

In fact, UA's network is highly segmented, where clients are grouped according to their roles. When clients connect to an Access Point (AP) within the campus, they are connected to a Virtual Local Area Network (VLAN) shared by several other clients, in which the private resources are accessible. Regarding NAT, the present mechanisms in the network are unknown, as are their mappings' Time To Live (TTL), which is an important variable mentioned in Section 2.3.1 to perform NAT-traversal techniques.

Any other specification of UA's network is unknown. Moreover, the network's firewall rules can't be altered to suit our needs.

2.5 Robot Operating System

ROS [1] is an open-source meta-operating system which provides an ecosystem for the development and distribution of robot software. ROS is considered a meta-operating system as it operates alongside a standard operating system, introducing a virtualization layer between applications and computational resources [31]. In other words, ROS abstracts developers from the hardware specification the applications will run on. Hence, ROS applications can be ported and reused across multiple devices, without requiring additional hardware-specific configurations or implementations. Moreover, communication through ROS is possible when dealing with devices employing distinct hardware implementations.

At its core, ROS is a combination of hardware abstraction operations, intra-process communication, device controllers and package management. Additionally, ROS includes a set of applications which implement common desired robot functionalities, such as navigation and object recognition. Due to its design focused on reusability and modularity, ROS packages developed by the community can easily be integrated in new applications.

2.5.1 ROS Master-Slave model

Any ROS application is composed of one or more nodes. A ROS node is perceived as the most basic program execution unit, generally serving a specific single purpose in order to optimize reusability. A ROS application also implies the existence of a master node, responsible for providing other nodes (called slaves) with on-demand information regarding

[1] https:www.ros.org/

each other. Slave nodes store their information in the master node and retrieve information regarding other slaves when necessary. Hence, a ROS master will act as a disocvery server, allowing nodes to learn the services running in other application peers and the addresses they should request connections to in order to use them. Communication between a master and its slaves is done using the XML-Remote Procedure Call (XMLRPC) protocol, a lightweight HTTP based protocol.

2.5.2 Messages

Nodes in a ROS application exchange data through messages. A ROS message is essentially a set of fields, each defined by a data type and value pair. Message fields can be defined as basic data types, arrays or even nest other messages. Messages are serialized and trasmitted using variations of TCP and UDP.

2.5.3 Communication

Having described how nodes in a ROS application discover themselves and how data is represented, the following paragraphs overview three distinct communication scenarios. For asynchronous communication, useful for data streams such as infromation captured by sensors, ROS supports a topic based publish-subscribe channel. In these communications, nodes can act as either publisher or subscribers. Publisher nodes, after registering their information and topic in the master, are responsible for pushing messages associated with such topic, delivering them to registered subscriber nodes. Subscriber nodes interested in a topic query the master node for the addresses of publishers registered under that topic. Then, to consume the information, a subscriber node establishes a direct connection with a publisher, where the message stream is made available.

In scenarios where a request-response communication is more suitable, ROS implements a mechanism refered to as services, which operate as a classic client-server model. In fact, a ROS service implies the existence of a server node, which listens and handles client node requests. As a synchronous communication, clients are able to perform requests to such a node server and receive its result as a response. Both request and response are defined as ROS messages, previously explored.

Finally, ROS introduces the concept of actions. Actions can be perceived as an extension of services, suitable for long running tasks where feedback about operation status is required. Similarly to services, a client will request an action server to execute a set of tasks. However, instead of solely responding with the output for a given requests, action servers are able to report to clients the status of the procedure. Moreover, clients can issue an action cancel to a server, which aborts the server's action entirely.

Chapter 3

Proposed Overlay Network Manager

Having outlined the relevant research, background and technologies for this dissertation's scope, this chapter details the development of a WireGuard based overlay network manager.

3.1 Architecture

The solution's main goal is to create a communication protocol capable of establishing a link between two clients residing behind a network with firewall rules preventing direct connections from taking place. To connect peers in such a scenario, the solution introduces a relay server in the network, reachable to any client, which acts as an intermediary responsible for receiving and forwarding traffic from a source to a destination client. In order to ensure clients are able to communicate with such a relay, traffic is encapsulated as HTTP, with the server listening on a configured open port. Additionally, for the relay server to deliver a message to a destination client, nodes establish and maintain a connection with this server, bypassing the established only firewall policy, since a connection is already open and traffic can flow bidirectionally. HTTP is used for encapsulation since it further ensures packets get through the network's firewalls. In fact, HTTP is such a standard and essential protocol that it is almost always allowed through any firewall. Moreover, keeping an open channel between the clients and the relay server proves to be simpler and more manageable when using HTTP, namely due to the protocol's design to handle persistence and keep-alive functionalities.

Figure 3.1 presents the initial scenario. Both clients, while connected to the network, are unable to communicate since the firewalls at each endpoint only allow incoming traffic from already established connections.

In Figure 3.2, the same scenario is presented with the relay server as an intermediary. Here, both clients establish and maintain an open HTTP connection with the server. Using these channels, clients transmit encapsulated HTTP messages containing information about the destination node, which the relay server forwards through the destination client's already open connection.

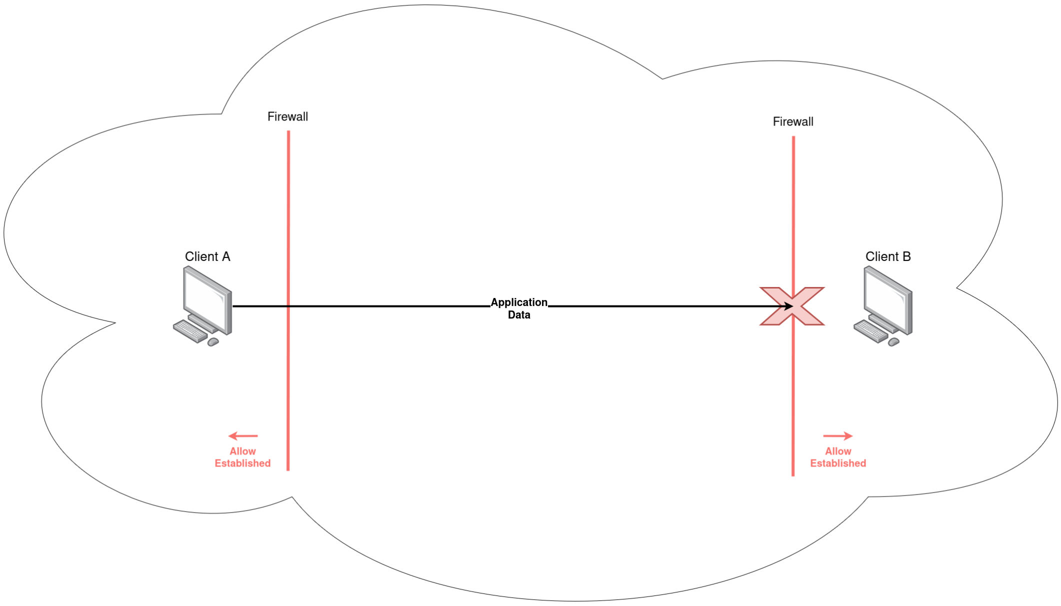


Figure 3.1: Clients are unable to establish any communication due to firewall policies.

3.2 Proposed Solution

In Section 2.3, Tailscale was explored as a WireGuard orchestrator, providing a coordination server and additional mechanisms to overcome restrictive network policies, namely UDP blocking firewalls and dynamic NATs. We've also seen that, when Tailscale fails to establish a direct connection, a relayed channel is established instead. In fact, relayed connections operate very similarly to the outlined architecture in 3.2. Using a fleet of geographically distributed relay servers, refered to by Tailscale as DERP servers, clients are able to encapsulate their WireGuard UDP traffic as HTTP and transmit it to a relay, which then delivers packets to the destination node. While both Tailscale's official coordination server and DERP fleet could be used, this solution aims to be entirely hosted within UA's premises. Therefore, Headscale, covered in Section 2.3.1, is used as an alternative to Tailscale's coordination server. Regarding the relay servers, Tailscale's DERP fleet is also discarded due to poor performance associated with geographic distance and the need for traffic confinement within UA's premises, again opting for a self-hosted alternative. The need for using our own DERP is further clarified in Section 3.3.2.

Hence, an overlay network in this solution is defined by a group of Tailscale clients, orchestrated by a self-hosted Headscale coordination server, where peers establish encrypted P2P communications with each other through the tunnel interfaces. Tailscale's coordination server uses additional infrastructure, also self-hosted in this solution. Regarding relayed connections, Tailscale uses DERP servers for traffic forwarding. Moreover, when dealing with direct connections, for peers to perform NAT-Traversal, required for the discovery process of their respective public addresses, Tailscale uses publicly available STUN servers. Fortunately, Headscale supports the deployment and seamless integration of both these services, allowing

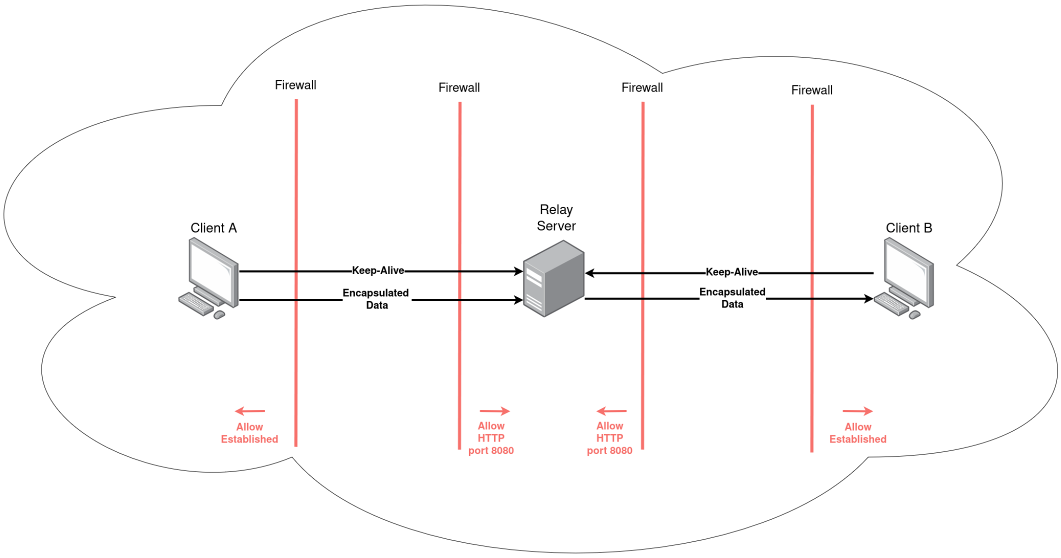


Figure 3.2: Clients establish communication by forwarding traffic as HTTP to a dedicated relay server.

---

the self-hosting of a complete Tailscale solution within UA's premises.

In the University of Aveiro, where the network employs such constraints preventing clients from directly establishing P2P connections with each other, a Tailscale peer is able to transmit its WireGuard encrypted UDP data to any other peer in its group by encapsulating packets as a TCP stream and forwarding them over HTTP to a relay server. In IRIS-Lab's use case, a team of autonomous robots is configured as a group of Tailscale clients, which maintain open connections to the relay server. This allows secure P2P communications to take place from any point in the university's campus. Additionally, developers can use their personal machines to join an overlay network and interact directly with the robots, a process which couldn't otherwise be achieved outside IRIS-Lab's network. Figure 3.3 depicts both this solution's architecture and the communication channels established according to distinct scenarios.

In fact, When a connection is established between two overlay peers, three different scenarios can arise, depending on the networks both source and destination nodes are communicating from. When both clients communicate from IRIS-Lab's internal network, since incoming UDP traffic is allowed and NAT-Traversal is done through the STUN endpoint listening on the coordination server, a direct Tailscale communication can take place. With one client connected to UA's network and one connected to IRIS-Lab's network, as the university's security policies only allow traffic from already open channels, direct WireGuard connections cannot be established. Hence, a relayed connection takes place. To communicate, clients send their packets to a relay server encapsulated as HTTP, which forwards the encrypted message to its destination using the previously open and persisted HTTP connections. To discover the address of available relays, the coordination server advertises the addresses of relay endpoints to the overlay clients. Finally, when both clients are connected to UA's network, the constraints are present on the two communications ends, establishing a relayed connection behaving exactly in the same way as the previous scenario.

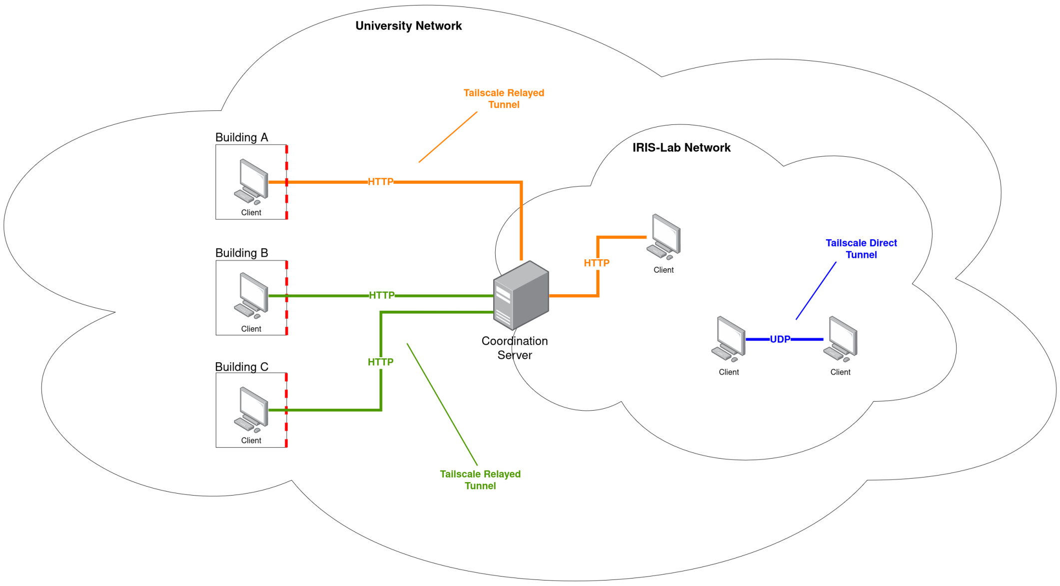


Figure 3.3: Solution's Communication Architecture. Three different scenarios can arise, identified by the blue, green and orange tunnels.

To self-host a Tailscale coordination server, Headscale functions as the solution's central entity. In addition to providing a Tailscale orchestrator alternative, Headscale allows the configuration and deployment of an embedded DERP server, which also exposes a STUN endpoint for NAT-Traversal. As such, all necessary central services are made available to clients through Headscale.

Clients run the open-source Tailscale's node software, a package containing a Command Line Interface (CLI) and a daemon. Tailscale's daemon is the entity responsible for establishing and managing a client's communications both with the orchestrator and other overlay nodes, while the CLI offers an interface to control and execute operations through the daemon.

Figure 3.4 presents the components forming both the Headscale coordination server and a Tailscale client.

3.2.1 Coordination Server

The coordination server is the central entity responsible for offering clients functionalities to configure themselves, peer discovering and aid in establishing communications when dealing with highly restrictive networks. It is composed of three main services, an orchestrator, a packet relaying service and an endpoint to perform NAT-Traversal. This section explores the operations of each of these components and their respective roles in the protocol.

Orchestrator

The orchestrator provides clients with network information and services which aid in the management and establishment of secure communications. In addition to performing client

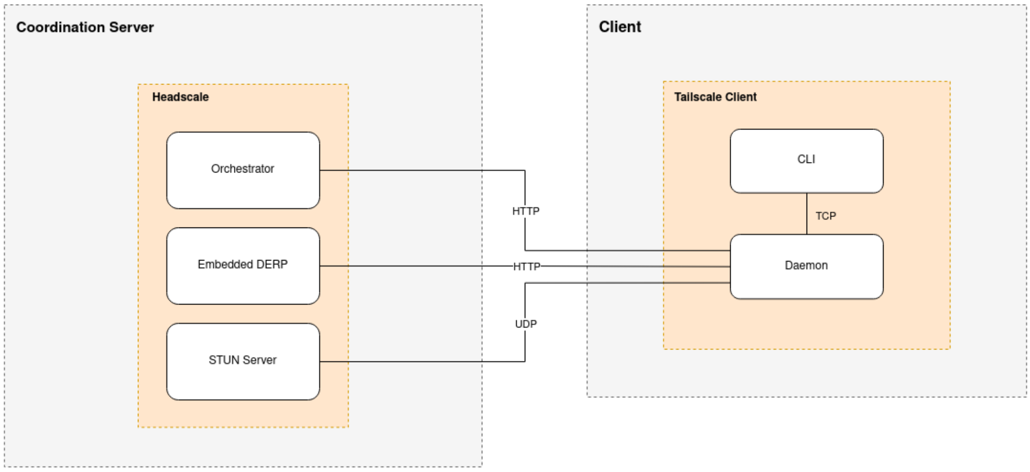


Figure 3.4: Headscale coordination server and Tailscale client components

authentication, authorization and key distribution, the orchestrator also ensures peer discovery, as it advertises nodes' public addresses to overlay clients. Moreover, the orchestrator facultates information regarding both STUN and DERP listening addresses, crucial for clients to perform NAT-Traversal and establish relayed connections, respectively.

Relay Server

In Section 2.3.1, two types of Tailscale connections were presented, direct and relayed. While the Tailscale protocol always attempts to establish direct connections first, on highly constrained networks, where NAT-traversal is unsuccessful or UDP traffic is entirely blocked by firewalls, Tailscale encapsulates encrypted packets as TCP and transmits them to a relay server, refered to as DERP servers, over HTTP. In other words, an overlay node communicating through one of such networks is able to perform this encapsulation process and forward packets to a relay. The relay server is then responsible for delivering the message to its destination. Hence, for relayed communications to be established, the orchestrator must advertise which DERP servers are available for clients to transmit packets to.

While Tailscale's official coordination server offers a fleet of geographically distributed DERP servers, the relay server in this solution is also a self-hosted, since our scope is to exclusevly serve UA's clients. Fortunately, Headscale supports the deployment of an embedded DERP server, which runs alongside the orchestrator's main service.

NAT-Traversal Server

Finally, the coordination server is also required to expose an endpoint for clients to discover which public IP they should advertise to the orchestrator and to perform NAT-Traversal, a process explored in Section 2.1.3. Tailscale clients perform NAT-Traversal using the STUN protocol.

Clients will periodically perform IP binding requests to a STUN server through UDP messages, receiving its respective public IP address and port as a response. This response is then advertised and stored in the orchestrator, allowing overlay nodes to retrieve the address information of any other peer present in the network. As establishing direct communication requires knowledge of peers' public addresses, supporting NAT-Traversal is a mandatory requirement for these types of connections.

3.2.2 Client

The Tailscale client runs a daemon and a CLI to control it. Tailscale's daemon is responsible for managing a client's network connections and respective configurations, while also interacting directly with the coordination server.

3.3 Implementation Details

With the components required to self-host a complete Tailscale solution defined, this section details the implementation process. A solution was developed and deployed in two different environments, serving distinct goals. First, to obtain a better grasp of the requirements associated with self-hosting a complete Tailscale solution and to ensure compatability when using the protocol with ROS applications, a virtual environment, configured in a single host, was used. Then, for validation and performance experiments, a solution was deployed within UA's premises, available to any client connected to the university's network.

3.3.1 Virtual Development Environment

The virtual development environment was designed to create an analogous scenario to the one being tackled, where two clients can't immediately establish P2P connections with each other but can both reach a public server. Hence, this environment hosts a narrow scope prototype of a self-hosted Tailscale solution. This prototype requires the deployment of an Headscale coordination server, followed by the configuration, and respective authentication, of two sample clients, using Tailscale's CLI. With the clients configured with Tailscale addresses, these can be used to run communication tests on ROS applications, through the encrypted tunnel.

Environment Specification

Such an environment was established using Oracle Virtual Box [1], a general purpose virtualizer. The environment is configured with three Linux virtual machines running on minimum resources. To achieve a networking scenario analogous to the one faced by UA's clients, the client virtual machines are attached to individual private NAT Networks and connected via Wi-Fi to an AP, while the coordination server virtual machine is attached to a network bridge on the host's Ethernet interface. Figure 3.5 depicts this topology. This networking configuration creates a situation where both clients have no knowledge of each other nor endpoints to communicate through, but are able to directly reach the coordination server. Obviously, this simulation of the networking conditions can't be entirely accurate, since, as referenced in section 2.4, details regarding UA's network mechanisms are vastly unknown. It is possible,

[1] Oracle Virtual Box, `https://www.virtualbox.org/`

however, to create a similar situation, where clients can't immediately form P2P communications with each other but can all communicate with an external, public server. Moreover, as stated in previous sections, Tailscale's protocol efficiently deals with most network constraints preventing direct communication. In other words, it is only known clients can't communicate, the reasons behind why they can't are abstracted by Tailscale.

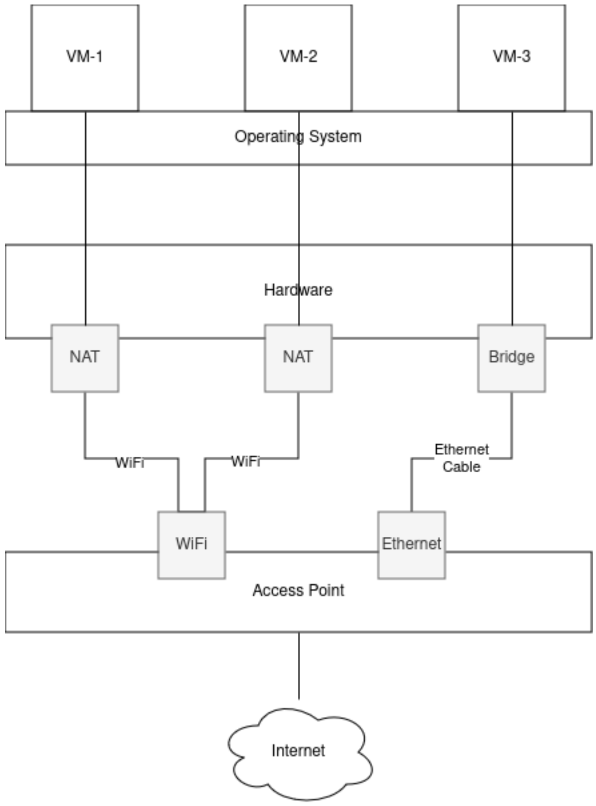


Figure 3.5: Virtual Development Environment Architecture

The access point used in this environment is a N600 Wireless Dual Band Gigabit router. The host machine is connected to the access point via Ethernet. With this setup, when one of the client machines wants to communicate with **VM-3**, traffic will be routed from the client to the AP's Wi-Fi interface. Then, the router will forward the packet through its Ethernet interface, reaching the host machine and, consequently, reaching **VM-3**, as its network adapter is bridged to the host's Ethernet interface.

The three virtual machines composing the environment run *Ubuntu Server 22.04* as their operating system. Due to the nature of the goals to be achieved in this phase, which focus on an extremely narrow scope, the machines require very little resources. Regarding networking, **VM-3** is attached to a bridge network on the Ethernet interface of the host machine. As for **VM-1** and **VM-2**, each respective network adapter is attached to a NAT, isolating clients on their own private networks, inaccessible from the outside. All machines can access the internet through the access point. Both **VM-1** and **VM-2** are assigned the same private IP address since they are residing in distinct private networks.

Table 3.1 summarizes the virtual machines' configuration.

	VM-1	VM-2	VM-3
Operating System	Ubuntu Server 22.04	Ubuntu Server 22.04	Ubuntu Server 22.04
Memory (Mb)	1024	1024	1024
Storage (Gb)	10	10	10
CPUs	1	1	1
Network Adapter	NAT	NAT	Bridged (ethernet)
Address	10.0.2.15	10.0.2.15	192.168.10.214

Table 3.1: Development Virtual Machines specification

Headscale Instance Deployment

Headscale provides a configurable open-source implementation of Tailscale's coordination server. At the time of writing, the latest stable Headscale release is *v0.22.3* [2], which is the version of the software referred to in the rest of this document.

The Headscale instance must be made available to all clients. In this environment, that means Headscale is deployed in **VM-3**, since the two other Virtual Machines can communicate with it directly. Hence, in this public server, an Headscale instance was deployed, following the official documentation's guidelines [3]

Headscale includes in its software package a YAML file used to configure coordination server's parameters. First, the **server_url** field, which dictates the endpoint clients should connect to for authentication, points to **VM-3**'s IP address, 192.168.10.214, on port 8080, where Headscale's main orchestrator service is listening. Regarding Tailscale IP assignments, this instance uses the default subnet prefixes, 100.64.0.0/10 for ipv4 and fd7a:115c:a1e0::/48 for ipv6. Registered clients will be assigned IP addresses in these ranges. For the scope of this environment, no other additional configurations are necessary.

With the service running, **VM-3** is now listening for Tailscale clients to connect and start using the protocol. To perform authentication, clients are required to login under a user. By default, Headscale does not create any users automatically. For development purposes, an Headscale user, **dev**, was created in the instance and shall be the user clients will register themselves under.

Client Deployment

Initially, clients can't really establish a direct connection in a traditional way. In fact, neither client is assigned a public address which could be used as a communication's endpoint, nor do they possess any information regarding one another. They can, however, reach the outside Internet, which consequently implies the translation of their private addresses into public ones, a process carried by the adapter attached to the host machine's NAT. In section 2.3.1, Tailscale's support for NAT-Traversal was explored. Thus, a Tailscale client is able to obtain its public IP and port by querying a STUN server and announce it to the control server. With both clients' public addresses stored, when communication is attempted, Tailscale will keep NAT holes open, allowing for a bidirectional flow of packets to be possible, through the public IPs addresses.

[2] Headscale's official releases, hosted in GitHub. https://github.com/juanfont/headscale/releases.
[3] Headscale's online documentation. https://headscale.net/running-headscale-linux/

	VM-1	VM-2
Tailscale Hostname	dev-1	dev-2
Tailscale IP (v4)	100.64.0.1	100.64.0.2
Tailscale IP (v6)	fd7a:115c:a1e0::1	fd7a:115c:a1e0::2

Table 3.2: Clients' Tailscale configuration after authentication

Hence, the Tailscale binaries were installed in each client, using Tailscale's install shell script [4].

At this point, clients are ready to perform authentication in the coordination server using the Tailscale's CLI and start communicating through Tailscale tunnels.

Authentication

Authenticating in the Headscale instance can be done either using a pre-authenticated key, generated by the coordination sever and shared with a client, or by accessing the instance through the browser in the client side. For automation purposes, the authentication keys provide a much more useful mechanism.

With that said, reusable keys were generated in the control server with Headscale's **headscale preauthkeys create**, an utility included in the software's CLI, and shared with its respective clients. The client machines are now able to authenticate by using the **tailscale up** command. Two additional command-line parameters are set, the *login-server*, which points to the address previously defined in Headscale's config's **server_url**, and the *authkey*, where the shared pre-authenticated key is injected.

With the clients authenticated, both devices are respectively assigned a Tailscale IP and hostname. Table 3.2 presents both clients' Tailscale configurations. At this state, a direct connection can be established via Tailscale interfaces.

Communication with ROS

With both clients up and communicating, our last validation in this environment aims to ensure the connections are compatible with the ROS middleware. Therefore, a very simple ROS scenario was deployed in the clients. As the scope for this experiment lies solely on validating communication through Tailscale in a ROS context, clients are only required to run a very basic ROS distribution, hence, a no-GUI package, **ros-noetic-ros-base** [5] was installed in both clients.

The experiment starts by configuring **VM-1** as a ROS Master, with the **ros-core** command which automatically assigns the client as the new master, listening on port 11311, under the **ROS_HOSTNAME** dev-1, matching its Tailscale hostname for convenience. Then, **VM-2** must acknowledge **VM-1** as the ROS master. The **ROS_MASTER_URI** environment variable points to where the ROS Master is listening. So, **VM-2** sets this variable with **VM-1**'s ROS Uniform Resource Identifier (URI), which is composed by **VM-1**'s hostname, dev-1, and the previously established ROS port, 11311.

[4] Tailscale's install script, publicly available online. https://tailscale.com/install.sh

[5] ROS-Base (Bare Bones). Basic ROS packaging, build and communication libraries. No GUI. Pulled from public repositories.

	Coordination Server
Operating System	Ubuntu Server 22.04
Memory (Gb)	16
Storage (Gb)	60
CPUs	4
Address	192.168.1.7

Table 3.3: Pilot environment's coordination server resources

VM-2 successfully configures its ROS ecosystem acknowledging **VM-1** as its master, effectively validating the use of Tailscale tunnels in conjunction with the ROS middleware.

3.3.2 Pilot Environment

The following section presents the configuration and deployment of a solution capable of being used by any client in the campus. As such, and as mentioned before, this implies that the Headscale instance must be available regardless of a client's physical location within UA's premises.

Headscale Deployment

In this solution, the Headscale instance is hosted within IRIS-Lab's network. Table 3.3 presents the server machine's resources. This, however, only allows communications from clients residing in the laboratory's private network. For clients to be able to reach the instance from any location in the campus, port forwarding rules were created on an exposed server in IRIS-Lab's network, available to any client connected to UA's network. This server forwards traffic on ports TCP/8080, for the main Headscale service and the embedded DERP and UDP/3478, for the STUN protocol, to the Headscale server. With this configuration, clients within the campus, which can directly reach the public facing relay machine via HTTP, can perform authentication in the Headscale instance. Figure 3.6 depicts such architecture.

Since the coordination server's host also runs Ubuntu in this environment, deployment followed the process described in the development environment (see Section 3.3.1). Regarding configuration, however, the **server_url** parameter, which points to the endpoint clients should use to authenticate, is now set as the Fully Qualified Domain Name (FQDN) of the proxy server, *iris-lab.ua.pt*.

Initially, Headscale was configured to use Tailscale's DERP server fleet. However, upon testing this scenario, relayed connections implied the redirection of traffic to a DERP server in Germany, resulting in tunnels with an average RTT of 452ms, a value unacceptable for the scope of this dissertation. Moreover, as this solution is meant to be used exclusively by UA's clients, traffic should be contained within the campus' network. Therefore, Tailscale's DERP fleet was discarded, opting for a self-hosted alternative.

Headscale allows the use of an embedded DERP server, which also exposes STUN functionalities for NAT-Traversal. This server can be enabled through Headscale's configuration file, and runs alongside Headscale's main orchestrator service, on the same address and port. As for the STUN endpoints, these are also configured to run in the same address, on UDP port 3478.

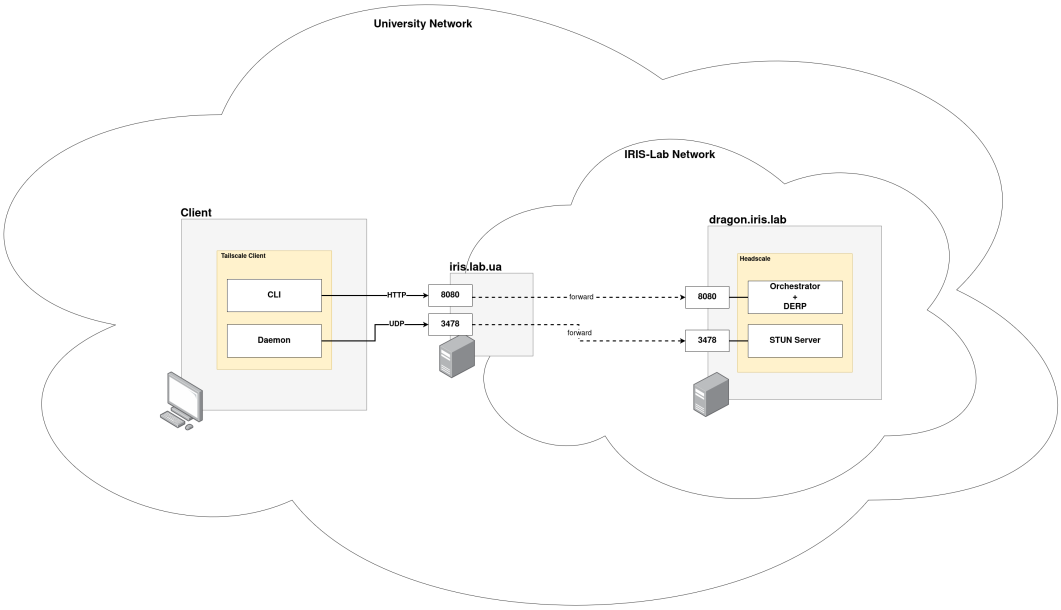


Figure 3.6: Pilot Environment Deployment Diagram

Using the embedded self-hosted DERP allows relayed communication between clients to be much more efficient, with more appealing latencies, discussed further in Chapter 4.

To use the self-hosted DERP server, Headscale is required to be exposed on an Hyper-Text Transfer Protocol Secure (HTTPS) server, hence it needs a valid X509 certificate. Although Headscale's configuration allows a seamless integration with certification tools such as Let's Encrypt (LE) or Caddy, in this scenario, where Headscale's FQDN is not public, such methods aren't as straightforward. Alas, to certify Headscale, self-signed certificates were generated using the **openssl** CLI and added manually to the instance, by pointing to both certificate and key file paths through Headscale's configuration file.

Using self-signed certificates requires additional client certificate trust configurations, covered in the following sections.

Client Deployment

Regarding client configurations, Headscale's certificate needs only to be added to the client machine's trusted list. On Debian distributions, this required moving the certificate to the */usr/local/share/ca-certificates* directory and updating the list with the command **update-ca-certificates**. With the coordination's server certificate added to the trust chain, a peer is able to use the protocol by running the official Tailscale client software.

3.3.3 Automation

This chapter details the development of automation tools to speed up deployment and configuration processes. Two main goals were achieved, related to automatic deployments of the control server and clients. First, automating the deployment of a configured Headscale instance. This implies automating software installation, configuration applying and certificate management. Regarding clients, deployments require Tailscale client installation, adding Headscale's certificate to the trust list and, finally, authenticate in the control server.

All scripts discussed in the following paragraphs are available on a GitHub repository. [6]

Headscale Deployment

Headscale's deployment requires both the installation of the software and its configuration, namely the configuration YAML and the server certificate.

Hence, the **install_headscale.sh** shell script starts by downloading a desired Headscale package, where version and system architecture are specified with command-line arguments. Then, the script fetches the Headscale's configuration YAML, publicly available in the automation repository, and applies it to the instance. Finally, the certificate is generated with the **openssl** CLI and placed, along with the certificate's key, in its respective path (matching what was defined in the the YAML).

Client Deployment

A shell script was also created for the clients. Here, configuring a client requires the installation of the Tailscale client package, running the already covered official install script. Finally, Headscale's server certificate is fetched and added to the system's trust mechanisms.

Running this simple script leaves a machine in a state where, given a pre-shared key, authentication can be performed with the **tailscale up** command.

[6] UA Overlays Automation repository: https://github.com/VascoRegal/ua-overlays-automation

Chapter 4

Experiments and Results

In order to validate the developed solution, several experiments were conducted which provide insights regarding the overlay networks' communication performance, security and usability in the context of ROS applications. Hence, this chapter details the procedures taken for results and metrics gathering, followed by their respective analysis and derived conclusions.

4.1 Network Performance

Network performance is understood as the quality of the communication in the eyes of an overlay node. Measuring network performance requires the collection and visualization of metrics produced while data is being exchanged on the overlay networks. For the scope of our conclusions, four of such metrics were gathered, on connections happening in distinct scenarios.

The metrics we chose for this experiment provide insights on three distinct dimensions, outlined in [4, 32].

Delay

Delay in network performance is a dimension refering to the time packets take to be transmitted from a source to a destination, through its communication medium. Network congestion, insufficient computational resources and routing rules directly influence network delay. To measure the delay associated with a connection, two metrics are considered, Round Trip Time (RTT) and jitter.

RTT times dictate how long a source has to wait to receive a response from its destination, while jitter measures the variation of packet arrival times. High jitter has an impact on the smoothness of a connection, generally having noticeable negative effects on real time communications and media streaming [33].

Bandwidth and Throughput

Bandwidth is a dimension perceived as the maximum amount of information capable of being transmitted through a connection in a given time interval. Throughput, while also measuring the amount of transmitted data over time, represents how much data in a given communication is actually transferred by unit of time. In other words, measuring throughput allows us to understand the rate at which data is transmitted between a source and a destination, while bandwidth measures the network's total capacity. Network throughput can be calculated by executing several packet transmissions between any two nodes and measuring how much information is successfully transmitted over a given time delta.

Losses and Errors

Finally, this dimension tackles how much information is not correctly transmitted and is measured by calculating a communication's packet loss, the fraction of packets sent which don't arrive at a destination. Losing packets implies data retransmission, which contributes negatively to the overall network performance.

4.1.1 Metric Gathering

The metrics to be gathered for network performance analysis are **RTT** and **jitter** regarding delays, TCP **throughput** for information transfer rate and **packet loss** to assess any issues regarding packets not being properly transmitted.

There are numerous open source tools which measure most of these metrics by running configurable network tests. One such example is iperf [1]. Iperf operates by exchanging packets through a client-server model and offers tests capable of collecting both TCP and UDP throughput and bandwidth, network jitter and packet loss. RTT times can be calculated by exchanging a given number of Internet Control Message Protocol (ICMP) packets and measuring communication travel times, achievable with simple **ping** commands.

These processes were automated using a python script [2]. This script starts by connecting to an iperf server and running TCP and UDP iperf tests, outputting its results as Java-Script Object Notation (JSON). Then, to the same host running the iperf server, ICMP packets are exchanged, calculating RTT minimum, maximum, mean and deviation values, which are also appended to the previously produced JSON object. These results are finally exported to a file for later processing. The duration of each iperf test and the total number of ICMP packets exchanged for RTT calculations are both configurable variables.

The result files produced by running this script allows us to build a dataset containing necessary metrics, used to draw the conclusions present further in this chapter.

4.1.2 Performance Experiments

Having defined the relevant metrics to evaluate communication performance through the overlays and development of a script which automates the results gathering processes, experiments are ready to be run between any two overlay nodes. Since communication is possible through Tailscale relayed connections regardless of clients' physical locations, we can collect insights from multiple campus points. This will not only allow us to understand the performance of the solution and how it behaves with geographical distance but also ensure the overlays can be used from anywhere within UA's premises.

For these experiments, an overlay node running an iperf server was deployed in IRIS-Lab, connected to UA's network. Then, with another machine, the experiment script was run,

[1] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool, `https://iperf.fr/`

[2] https://github.com/VascoRegal/ua-overlays-automation/blob/main/tests/network/iperf/run.py

Location	**Building ID**	**Connection Type**
University Library (BIB)	17	Tailscale (relayed)
Eletronics, Telecomunications and Informatics Department (DETI)	4	Tailscale (relayed)
Pedagogical Complex (CP)	23	Tailscale (relayed)
Mathematics Department (DMAT)	11	Tailscale (relayed)
Biology Department (DBIO)	8	Tailscale (relayed)
Economics and Management Department (DEGEIT)	10	Tailscale (relayed)
Exhibition Hall (EXPO)	24	Tailscale (relayed)

Table 4.1: Locations used for network performance analysis

pointing the iperf server as the IRIS-Lab's node's Tailscale IP, from disperse campus' locations. Table 4.1 presents the selected test case buildings.

Regarding test configuration, both TCP and UDP iperf tests were run for 10 minutes each. Regarding RTT calculations, a total of 100 ICMP replies were used.

4.1.3 Results

The metrics collected from each building are presented in Table 4.2.

Building	Round-Trip Time (ms)			Average TCP Throughput (Mbit/s)	Average Network Jitter (ms)
	Min	Mean	Max		
DMAT	8.24	22.37	240.9	45.874	0.859
DEGEIT	7.85	19.23	244.05	49.615	1.119
CP	7.63	19.43	239.96	43.061	0.834
BIB	7.94	12.88	45.17	46.849	0.764
DBIO	0	0	0	0	0
DETI	8.41	20.01	265.88	31.744	11.854
EXPO	7.63	17.06	219.88	33.685	2.843

Table 4.2: Solution's network performance results from various campus' buildings.

4.2 Protocol Overhead

Communicating with the Tailscale protocol implies additional operations to take place on sending and receiving data, which introduces an overhead to the general network performance. In fact, data encryption, security mechanisms and transmission of additional Tailscale protocol packets produce a toll on the regular network behaviour. Moreover, since connections within the university's network are relayed, meaning traffic is being redirected over HTTP, an additional degradation of network performance is also expected. To understand this dimension, this section presents an experiment which provides insights on said protocol overhead, regarding network delay and throughput.

This experiment aims to measure Tailscale's overhead on communication, by interpolating data collected, under very similar conditions, from a connection through a regular Wi-Fi interface against a connection done through Tailscale tunnel.

Such a scenario can be achieved with two clients residing inside IRIS-Lab. With both machines connected to IRIS-Lab's private network, they can communicate directly via private IPs, hence metrics were collected with the script described in Section 4.1.1. Then, the same metric gathering was run for the same scenario, with one of the clients connected to UA's network and the test done via Tailscale IPs.

4.2.1 Results

The two generated datasets allow us to visualize how Tailscale impacts network performance. Figure 4.1 presents the TCP throughputs for the both experiments, while figure 4.2 compares the RTT distributions. Analysing the figures corroborates the idea that communicating through relayed Tailscale connections implies a considerable loss in network performance. To be able to represent this performance depletion numerically, we can calculate the relative change between the two measurements, using the percent log-change formula [34].

The percent log-change is a numerical indicator of the change between two groups of values. In the context of this analysis, the percent log-change represents how much throughput was lost and how RTT times increased, when comparing the internal and tunnel connections. We chose to use the log-change as an indicator instead of directly calculating the percentage change as it outputs a symmetric indicator, abstracting the necessity of choosing one connection as the baseline. The percent log-change is given by the expression:

$$\text{Log-Change (\%)} = \ln\left(\frac{y}{x}\right) \times 100$$

Where **y** and **x** are the two set of values under comparison. Using the average measurements generated by the network tests, throughput loss can be calculated with the expression:

$$\text{Throughput Loss (\%)} = \ln\left(\frac{\overline{\text{TP}}_I}{\overline{\text{TP}}_T}\right) \times 100$$

Where $\overline{\mathbf{TP}}_I$ is the average throughput for the internal connection and $\overline{\mathbf{TP}}_T$ is the average throughput for the Tailscale connection, which results in a throughput loss of **46.23** %

The same procedure can be applied to the measured RTT values:

$$\text{RTT Increase (\%)} = \ln\left(\frac{\overline{\text{RTT}}_I}{\overline{\text{RTT}}_T}\right) \times 100$$

	Average TCP Throughput (Mbit/s)	RTT mean (ms)
Internal Connection	45	10.1
Tailscale Relayed Tunnel	32	19
Loss (%)	21.5	35.1

Table 4.3: Internal and Tailscale connections' performance comparison

Similarly, $\overline{\mathbf{RTT}}_I$ is the average RTT times for the internal connection while $\overline{\mathbf{RTT}}_T$ is the average RTT times for the Tailscale connection. This calculation results in an RTT increase of **21.11** %

Table 4.3 summarizes the collected results. Corroborating the initial performance degradation assumptions related to the relayed tunnel's HTTP encapsulation, these two performance loss indicators give us a qunatification of this solution's overhead.

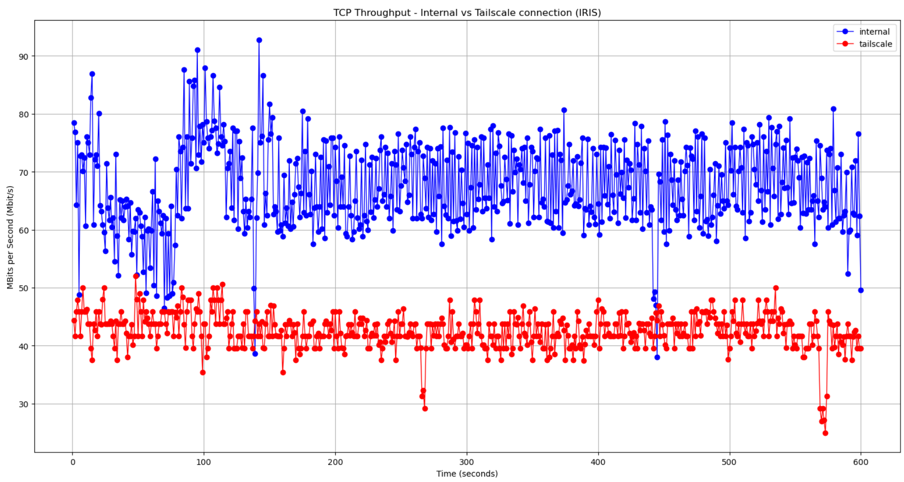


Figure 4.1: Tailscale's toll on throughput, tested on internal and Tailscale connections inside IRIS-Lab

4.3 Security and Confidentiality

As we have seen, Tailscale encrypts and secures communications with WireGuard. This experiment aims to visualize how packets in the overlays are perceived by untrusted entities. A straightforward method to verify this dimension relies on the analysis of network captures. In fact, messages in a Tailscale tunnel can only be understood by the source and destination nodes. Traffic travelling through any other infrastructure is seen as an undecipherable stream of packets.

Hence, this validation operates by generating traffic between two overlay nodes and simultaneously capturing the packets in the destination node's Tailscale interface and in another

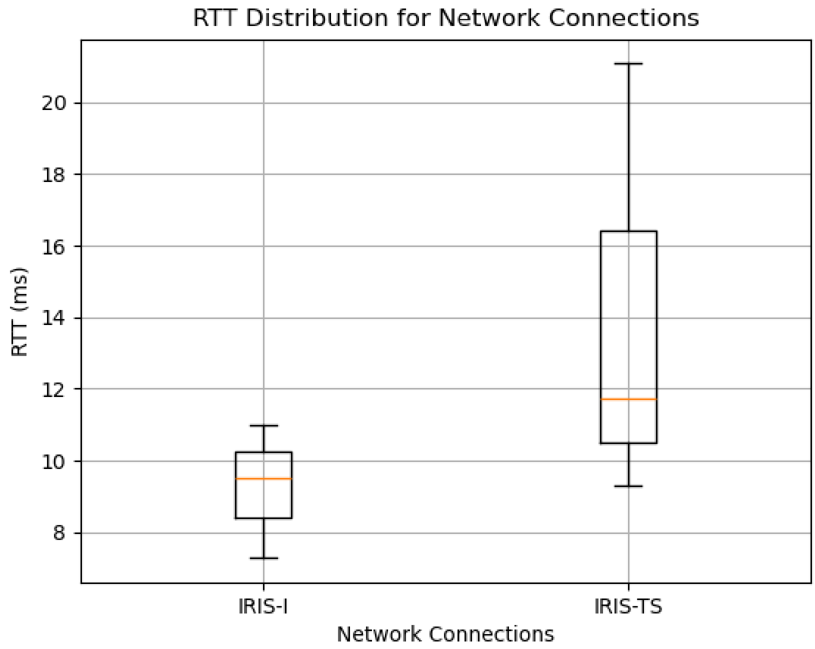


Figure 4.2: Round Trip Time distribution for internal (IRIS-I) and Tailscale (IRIS-TS) connections

---

intermediary network link's interface. On relayed connections, which is the case in this scenario, this intermediary capture is done in the DERP server's interface. We can verify the encryption taking place by correlating packets from both captures using their timestamps as reference.

4.3.1 Capturing and Visualising Traffic

To capture packets on a network interface, **tcpdump** [3], an open source command-line tool, was used. With this tool, we can initiate a capture by using the **tcpdump** command and pointing to a desired interface. Results from a capture can be exported to a file, which then allows a user friendly visualisation using **Wireshark** [4], also an open source tool. Wireshark facilitates analysing and correlating the captures.

4.3.2 Generating Traffic

For the scope of this experiment, a client server scenario was created using two overlay nodes. One of the nodes runs a simple HTTP server, listening on its Tailscale interface, while the other generates traffic by executing requests to this server using **curl** [5]. Both nodes are connected to UA's network, hence the Tailscale connection is relayed.

[3] Tcpdump, `https://www.tcpdump.org/`
[4] Wireshark, `https://www.wireshark.org/`
[5] cURL, `https://curl.se/`

4.3.3 Experiment

Having defined how traffic is generated and captured, the experiment starts by initiating two captures, one on the HTTP server's Tailscale interface and one on the DERP server's Wi-Fi interface. Traffic associated with the generated HTTP requests will travel through both these interfaces, as all Tailscale traffic requires relaying by the DERP server before reaching its respective destination.

With the captures running, HTTP requests were done by the client node, using cURL. After closing the captures, two pcap files are generated, which are loaded in Wireshark and correlated manually by analysing the packets' timestamps.

4.3.4 Results

Figure 4.3 presents both captures side by side in Wireshark. By analysing these results, the HTTP request and response are encapsulated as a TCP stream on the DERP server capture, and its payload encrypted, while the corresponding packets in the client's HTTP server's interface capture shows the same request and response as plain-text. Hence, Tailscale is effectively encrypting tunnel traffic, as only communication end nodes are able to decrypt and understand the information being transmitted.

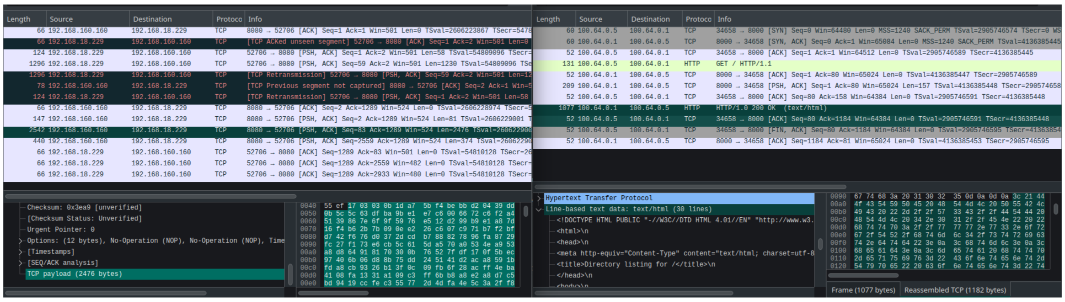


Figure 4.3: TCP Streams from external capture (left) and destination node capture (right)

4.4 Live Camera Feed Application

The next experiment conducted aims to analyse how ROS applications operate when using the overlay networks. For that, a ROS application which publishes a camera feed is launched in a machine connected to IRIS-Lab's network. Then, in another ROS node, the output of the camera is accessed, from outside the laboratory, connected to the campus' network. With this experiment, we are able to compare metrics such as the video's bandwidth and framerate from the source against what a client actually consumes both when connected to the laboratory's internal network and when using the overlays.

4.4.1 Camera and ROS Setup

For this experiment, a two node ROS environment is created where the master node, running inside IRIS-Lab's network, publishes the feed from an external Astra camera to a topic. Then, another node subscribes this topic, being able to consume the camera feed.

Figure 4.4 presents said setup.

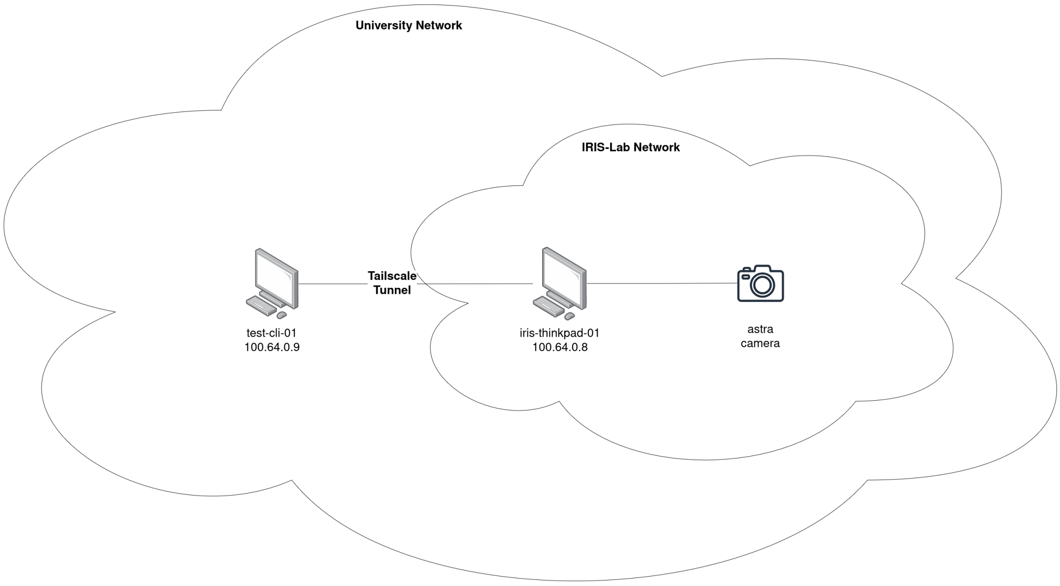


Figure 4.4: Camera setup. The ROS node inside IRIS-lab's network publishes the camera feed.

	Source Camera Feed	**Internal Consumer**	**Tailscale Consumer**
Average Bandwidth (MBytes / s)	27.76	14.53	6.25
Average Frame Rate (frames / s)	30.060	14.582	6.515

Table 4.4: Streaming performance from source and consumer's end on internal and Tailscale connections

4.4.2 Collecting Metrics

The raw, uncompressed camera feed is published to the **/camera/color/image_raw** topic. Image bandwidth and frame rate can be calculated at both ends with the **rostopic bw** and **rostopic hz** commands. These two metrics are used to evaluate the solution's performance in this context.

Metrics were collected on two different scenarios. One with the consumer node also connected to IRIS-Lab's network and communicating through private IPs and one connected to UA's network, accessing the feed via Tailscale interfaces.

4.4.3 Results

Table 4.4.

4.5 Robotic Arm Operation

The last conducted experiment also aims to analyse the performance of ROS applications through the overlays. For this experiment, a robotic arm connected to a ROS Master residing in IRIS-Lab is controlled by another ROS node, connected to UA's network.

4.5.1 ROS Setup

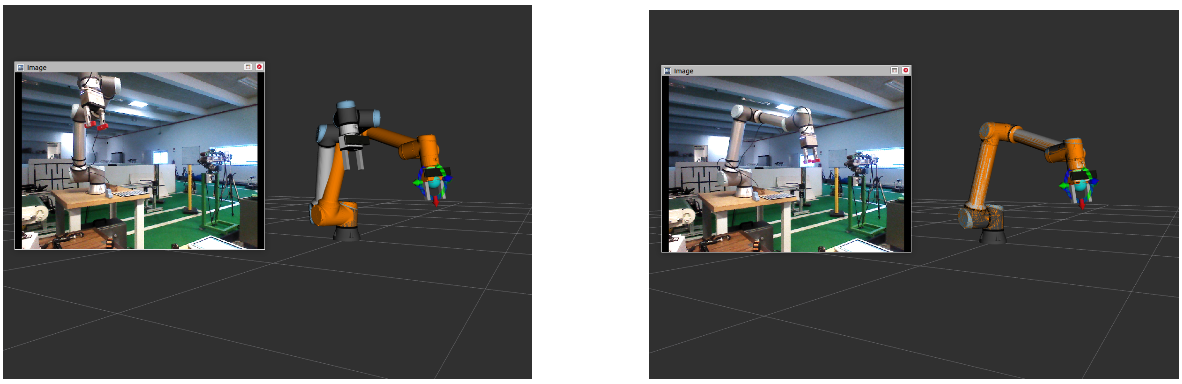
The robotic arm is directly connected to the master node and the ROS application is launched. The camera from the previous experiment is also setup, in order to have a visual feed of the robot movement when controlling it remotely. Figure 4.6 presents this configuration.

4.5.2 Operating the Robotic Arm

Moving the robot is achieved by maniuplating the robot's **joint_states**, a ROS topic which contains the information regarding the arm's position.

To make the robot operation smoother, RViz [6], a ROS 3D visualization tool, was installed in the slave node. RViz provides an interface containing a 3D model of the robot, where movement can be planned by moving the joints on the 3D model. Then, executing such a plan will generate the set of ROS messages responsible for updating the real robot's joints to match the position defined in the model.

Figure 4.5 presents an example of operating the robot through RViz.



(a) Robot's initial state. The orange outline represents the desired postion.

(b) Robot's state after executing the planned movement.

Figure 4.5: Moving the robotic arm using RViz.

4.5.3 Results

Since the messages involved in moving the robot ¡alguma cena de serem pequenas¿, calculated with the **rostopic bw joint_states** command, operating the robot through the overlays performs very similarly to the usual robot operation. In fact, the loss of throughput associated with the overlays is negligible when dealing with messages of such size.

[6]http://wiki.ros.org/rviz

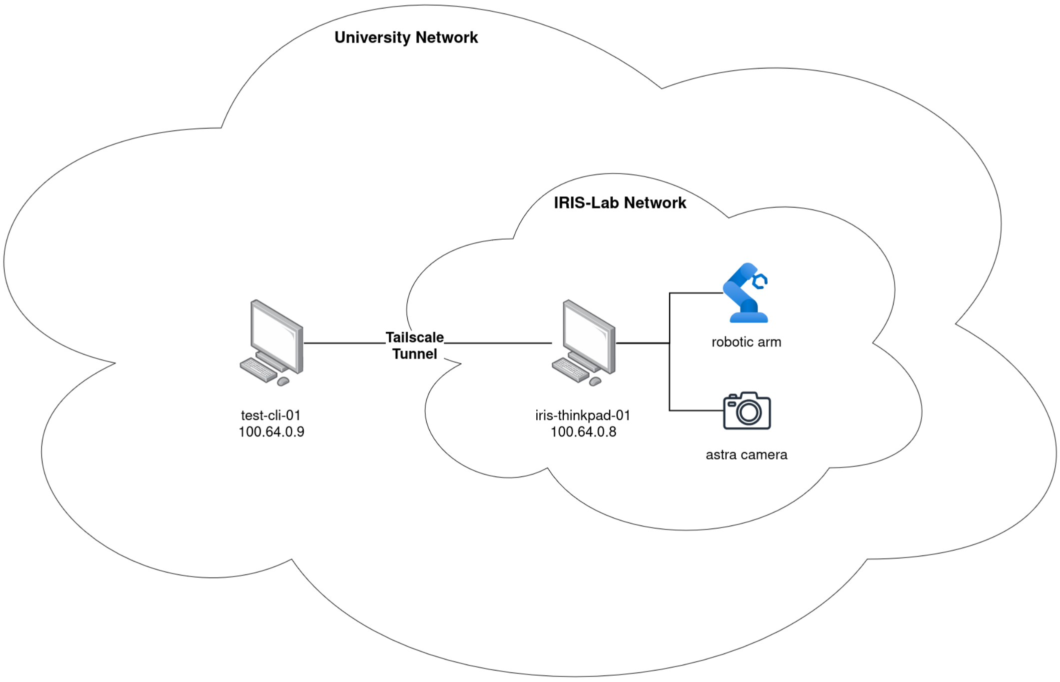


Figure 4.6: Robotic Arm setup. The camera feed captures the robotic arm. The robot is operated through the ROS node residing in the University Network

4.6 Chapter Summary

This chapter presented how this solution was validated and its performance analysed. To collect network performance metrics, **iperf3** was used in a range of different scenarios. First, to gain insights on Tailscale's overhead, network tests were run in the same conditions, through regular interfaces and through Tailscale tunnel. This resulted in a loss of **46.23** % in TCP throughput and an increase in average latencies of **21.11** %. Regarding coverage, which refers to the campus area Tailscale can operate under, all identified key places were able to communicate via Tailscale relayed connections when using UA's network. Moreover, performance results from these experiments showed no particular anomalies, although, and as expected, certain areas underperform.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Concluding, this document presents the details for the implementation of a secure overlay network manager, hosted entirely in a private environment.

After a systematic review of general networking concepts and potential tools and services, WireGuard appeared as the most suitable protocol for encrypted communication.

Although creating stand alone WireGuard overlay networks is possible, it requires managing and configuring individual peers, a process raising scalability and ease of administration issues.

Fortunately, Tailscale addresses such flaws by offering a coordination server capable of orchestrating the creation and configuration of WireGuard peers and respective tunnels, without sacrificing WireGuard's state of the art performance. Additionally, Tailscale employs a set of mechanisms and infrastructure which are able to overcome network constraints that would normally prevent the direct establishment of a WireGuard connection.

To confine this solution to UA's network, Headscale, an open-source self-hosted implementation of Tailscale's coordination server, was used alongside necessary additional services, namely a DERP and a STUN server. Regarding clients, these are required to trust Headscale's server certificate and run Tailscale's open source client software.

Regarding validation, network tests were run through the overlays, using **iperf3**. This allowed the measurement of Tailscale's protocol overhead on communication, which, as expected, resulted in a loss of throughput and increase in RTT. These experiments also provide insights on the overall network performance across multiple campus areas. Finally, the solution was tested when applied to ROS applications, with successful results.

Overall, this project delivers automated procedures to configure and deploy a secure self-hosted overlay network manager requiring minimal client configuration, accomplishing the goals outlined in Section 1.2.

With IRIS-Lab's robots authenticated in the self-hosted Headscale control server and, consequently, configured with Tailscale interfaces, encrypted communication is possible from anywhere within UA's campus through HTTP relayed connections, successfully enabling robot operation outside IRIS-Lab's internal network.

Even though the developments presented in this document meet the goals as expected, serveral aspects of this solution could be improved or should be more thoroughly analysed.

First, regarding deployment automations, although the scripts presented in Chapter **??**

provide a configurable procedure to deploy both an Headscale server and Tailscale clients, this logic could easily be ported to more adequate Infrastructure as Code (IaC) frameworks. In fact, the steps taken in said scripts are easily implemented as, for example, Ansible [1] playbooks. This would allow a team of robots to be defined as an Ansible inventory, creating a true one-touch deployment, not requiring the scripts to be run manually in each individual peer.

An Access Control List (ACL) is a Tailscale functionality capable of adding an extra security layer to overlay networks by defining policies allowing or denying traffic between nodes. These policies can be applied to Tailscale users, tags, or client groups. Ideally, when deploying a team of robots, these should be grouped and traffic restricted to that specific group, creating an additional logical segmentation to the network, furthering the confidentiality and privacy of the solution.

Regarding implementation, such an ACL system could easily be built on top of the previous automation improvements. In other words, a one touch deployment of a team of robots through IaC software would allow the creation of Tailscale groups, associating a team's clients to that group and, finally, applying the ACL.

In Chapter 4, the network tests were run for a duration of 15 minutes each. Evidently, this provides only but a glimpse of the actual network behaviour. To create a more accurate perception of the impact Tailscale has on network throughout the campus, these experiments should be redone using a more ample time frame. This could be achieved either by tweaking the python test script parameters or by setting up monitoring tools such as Grafana [2] integrated with **iperf3**'s metric collections. Using such a monitoring platform would also provide dashboards to help visualize the overall results.

Finally, this solution's scalability should also be properly analysed, by collecting network metrics as more peers are added to the overlays. In fact, the coordination server might introduce a bottleneck, especially regarding the single DERP. As connections outside IRIS-Lab network are all relayed, a single DERP might suffer a degree of resource overloading, as all traffic is being relayed by this service.

[1] Ansible, https://www.ansible.com/

[2] Grafana, https://grafana.com/

Bibliography

[1] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys Tutorials*, 2005.

[2] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole Jr, "Overcast: Reliable multicasting with an overlay network," in *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[3] M. Waldvogel and R. Rinaldi, "Efficient topology-aware overlay network," *ACM SIGCOMM Computer Communication Review*, 2003.

[4] Peterson, Larry L. and Davie, Bruce S., *Computer Networks, Fifth Edition: A Systems Approach.* Morgan Kaufmann Publishers Inc., 2011.

[5] J. T. Harmening, "Chapter 58 - virtual private networks," in *Computer and Information Security Handbook (Third Edition)*, Morgan Kaufmann.

[6] André Zúquete, *Segurança em Redes Informáticas.* FCA, 2013.

[7] T. Berger, "Analysis of current vpn technologies," in *First International Conference on Availability, Reliability and Security (ARES'06)*, 2006.

[8] A. Keränen, C. Holmberg, and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal." RFC 8445, 2018.

[9] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, "Session Traversal Utilities for NAT (STUN)." RFC 8489, 2020.

[10] K. Seo and S. Kent, "Security Architecture for the Internet Protocol." RFC 4301, 2005.

[11] C. Kaufman, P. E. Hoffman, Y. Nir, P. Eronen, and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)." RFC 7296, 2014.

[12] S. Kent, "IP Authentication Header." RFC 4302, 2005.

[13] S. Frankel, K. Kent, R. Lewkowski, A. D. Orebaugh, R. W. Ritchey, and S. R. Sharma, "Guide to ipsec vpns:.," 2005.

[14] T. S. Nam, H. Van Thuc, and N. Van Long, "A High-Throughput Hardware Implementation of NAT Traversal For IPSEC VPN," *International Journal of Communication Networks and Information Security*, 2022.

[15] C. Singh and K. Bansal, "NAT Traversal Capability and Keep-Alive Functionality with IPSec in IKEv2 Implementation," 2012.

[16] J. Yonan, "Openvpn." `https://openvpn.net/`.

[17] J. A. Donenfeld, "Wireguard: next generation kernel network tunnel.," in *NDSS*, 2017.

[18] J. Ćurguz *et al.*, "Vulnerabilities of the ssl/tls protocol," *Computer Science & Information Technology*, 2016.

[19] D. J. Bernstein, "Curve25519: new diffie-hellman speed records," in *Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9*, 2006.

[20] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[21] T. Perrin, "The noise protocol framework," 2018.

[22] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, 2008.

[23] G. Procter, "A security analysis of the composition of chacha20 and poly1305," 2014.

[24] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the wireguard virtual private network protocol," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[25] B. Dowling and K. G. Paterson, "A cryptographic analysis of the wireguard protocol," in *Applied Cryptography and Network Security: 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings 16*, 2018.

[26] S. Mackey, I. Mihov, A. Nosenko, F. Vega, and Y. Cheng, "A performance comparison of WireGuard and OpenVPN," in *Proceedings of the Tenth ACM Conference on data and application security and privacy*, 2020.

[27] L. Osswald, M. Haeberle, and M. Menth, "Performance comparison of vpn solutions," 2020.

[28] A. Pennarun, "How tailscale works." `https://tailscale.com/blog/how-tailscale-works/`, 2020.

[29] C. F. Jennings and F. Audet, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP." RFC 4787, 2007.

[30] J. Font and K. Dalby, "Headscale." `https://headscale.net/`, 2023.

[31] L. J. D. L. Yoonseok Pyo, Hancheol Cho, *ROS Robot Programming (English)*. ROBOTIS, 2017.

[32] A. Hanemann, A. Liakopoulos, M. Molina, and D. M. Swany, "A study on network performance metrics and their composition," *Campus-Wide Information Systems*, 2006.

[33] M. Claypool and J. Tanner, "The effects of jitter on the peceptual quality of video," in *Proceedings of the seventh ACM international conference on Multimedia (Part 2)*, 1999.

[34] L. Törnqvist, P. Vartia, and Y. O. Vartia, "How should relative changes be measured?," *The American Statistician*, 1985.