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

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 virtual or logic channel, built on top of another network. Although there are already numerous services and protocols implementing this mechanic, scalability and administration agility are among the most desired characteristics of such a network topology. Hence, this document presents a centralized solution for the creation and control of secure overlay networks for multiple nodes - from client management to operation auditing, based on Wireguard, an open-source protocol for encrypted communication. In the University of Aveiro, namely the autonomous robot ecosystem residing in the IRIS lab, supporting such a networking architecture would prove to be particularly interesting, both for development and project organization.

# 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. 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 constraints to inbound and outbound traffic.

Such is the case in the University of Aveiro (UA), where the network, although covering most of its edifices, enforces several constraining mechanisms that prevent, for example, the establishment of direct Peer to Peer (P2P) between two clients.

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 in the remaining of UA's locations.

Overcoming these limitations would be extremly 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 UA's clients. 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 management platform should provide operations to achieve a secure, private communication between a group of robots connected to UA's network, regardless of their physical location within the campus. 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 the 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 is suitable for IRIS-Lab's projects requirements.

## 1.3 Document Structure

This document presents an implementation of such an overlay network manager. Hence, it is structured in two main chapters, the state of the art and the methodology. The former describes an exploration of the background and current state of the art, providing an analysis not only of potential tools, protocols, and frameworks suitable for the scope of the dissertation but also of published research conducted covering similar topics and scenarios. The latter establishes the work methodology to be taken for the development and results gathering process.

# Chapter 2

# State of the Art

"Observation is a dying art"

— Stanley Kubrick

## 2.1 Overlay Networks

In the last few decades, the Internet has been subjected to an exponential growth, both in the number of users and connected devices. To answer the increasing demand and support aspects such as mobility and scalibility, Internet applications have diverged from classic distributed systems to more complex network topologies, creating an extremly heterogeneous environment. In such a non-patternized landscape, P2P overlay networks have emerged as a topic of growing interest, as conducted research on the matter attempts to create networking solutions capable of addressing the adversities imposed by the modern day Internet [13]. This section explores the fundamental principles of overlay networks and how its abstraction layer is able to produce a topology with the potential to adapt to the limitations of today's Internet.

By definition, an overlay network is a logical network implemented on top of the links of another, already established, underlay network [19]. In other words, nodes (also called peers) in an overlay network (which also exist in the underlay network) implement its 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 P2P logical links that form the overlay network. Figure 2.1 illustrates this concept.

Overlay networks can be applied to applications to achieve numerous functionalities, such as finding alternate routes for unicast applications, explored by Resilient Overlay Networks (RON) [1], which measures peers' average latencies in order to discover routes optimizing network end-to-end reliability and performance. This idea is also the principle of Software-Defined WAN (SD-WAN) services. Another common example of the use of overlay networks are object sharing applications such as BitTorrent, where the overlay network is formed by the nodes which possessing parts of the desired file. When a peer want to download a file, it establishes P2P connections to the other overlay peers and retrieves the part of the data until the file is complete.

While decentralized by design, some topologies applying overlay concepts have a degree of centralization, generally serving information regarding active peers in the network. Although a central point can cause bottlenecks, on-demand peer information is able to make a system

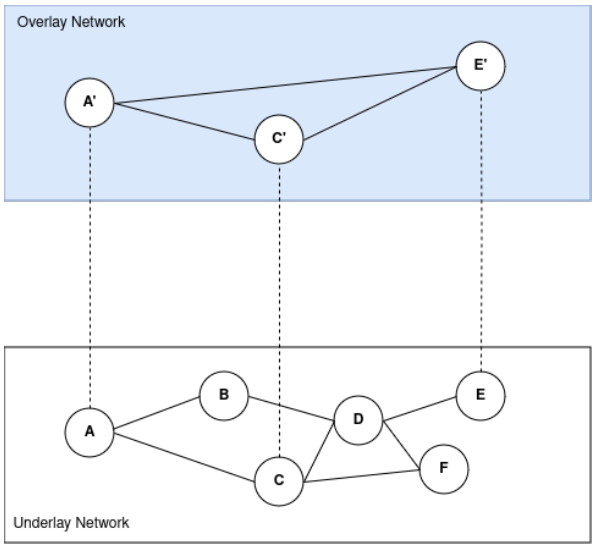


Figure 2.1: Concept of a very basic overlay network. Nodes A, C and E create logical links with each other, forming an overlay network

easier to scale and reconfigure. BitTorrent is an example of this, where a new peer queries a central server, called a tracker, retrieving the addresses and identification of the peers it should connect to. Such centralization can, however, be discarded by implementing a peer finding algorithm which relies solely on the other peers already in the network.

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 traffic, which is the case when dealing with devices in the underlay network, like routers or switches. Routing traffic through a node allows any application-defined manipulation to be applied to the datagrams, namely cryptographic operations. This idea is explored in further sections, as it serves as the backbone in several encrypted P2P communication protocols.

Conclusively, the use of overlay networks provides an efficient and reliable way to create a logical structure between a set of nodes residing in an otherwise unstructured topology. Additionally, as overlay nodes are essentially end systems, there's great leverage for applications to freely manipulate and route traffic. By relying on P2P communications, overlay networks' decentralized nature provides a very scalable and reconfigurable design.

## 2.2 Virtual Private Networks and Encrypted Peer to Peer Protocols

As mentioned in the previous section, the Internet's general topology proves to be highly unpredictable and unstructured. Effectively, traffic traveling through the Internet is forwarded across multiple devices and links, which an user generally can't control. This reality raises questions regarding privacy and confidentiality of a communication, as sensitive data could easily be compromised. To secure packets and ensure only trusted entities can access the data, mechanisms have been employed to encrypt and authenticate datagrams. One such mechanism is the use of Virtual Private Networks (VPNs). The following section first presents

an analysis on how VPNs offer a security layer to communications, followed by an exploration of prominent implementations of such solutions, regarding not only their cryptographic and authentication methods but also protocol performance and reliability when faced with network constraints.

VPNs have become a mature technology. With such a range of products offering VPN capabilities, this section aims to analyze some of its most notable providers, focusing on the processes involved on their respective data planes, which refers to the subsection of network communications responsible for carrying data between devices. This implies not only the robustness of its authentication methods, encryption suite and protocol security but also its features regarding concepts such as mobility - how the service behaves when clients change their physical locations and Internet Protocol (IP) addresses - and overcoming constraints associated with networks using Network Address Translator (NAT) mechanisms and secured with firewall rules. Finally, since operations taken in the data plane are necessarily associated with computational overheads, namely traffic encryption and session management, overall performance is also perceived as a valued dimension.

## 2.3 Virtual Private Networks and Encrypted Peer to Peer Protocols

VPNs have become a mature technology, with widespread usage on the Internet. With such a range of products offering VPN capabilities, this section aims to analyze some of its most notable providers, focusing on the processes involved on their respective data planes, which refers to the subsection of network communications responsible for carrying data between devices. This implies not only the robustness of its authentication methods, encryption suite and protocol security but also its features regarding concepts such as mobility - how the service behaves when clients change their physical locations and IP addresses - and overcoming constraints associated with networks using NAT mechanisms and secured with firewall rules. Finally, since operations taken in the data plane are necessarily associated with computational overheads, namely traffic encryption and session management, overall performance is also perceived as a valued dimension.

VPNs can be classified according to their topology in two main categories: client-to-site and site-to-site. A client-to-site VPN is characterized by connections from a single user (client) to a private network (site), while site-to-site VPNs offer a secured connection between two private networks. Thus, in site-to-site networks, users are not required to individually configure VPN clients. The tunnel in this type of VPN is made available to the entire network.

For the scope of the scenario at hand, where robots (the clients) require access to a private network, there's an emphasis on client-to-site use cases.

This section aims to explore some of the most popular and widely used VPN protocols, regarding features, cryptography and performance. The structure of some of the following paragraphs is loosely inspired by similar research and publications, namely [2].

### The problem with NAT

NAT is a networking mechanism responsible for translating IP addresses in private networks into 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 con-

straint, 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 fact, it is very likely that devices on 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 [11] and STUN [20]. Hence, each VPN service can have its own way of supporting NAT Traversal. Each case is explored separately in its own subsection.

### 2.3.1 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 [22]. 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) [9] 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 [22], 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, in the context of VPNs, 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 [10]. 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 [7].

#### 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, difficulting cipher analysis techniques. Moreover, it can be made compatible with NAT and employ NAT-traversal techniques [14], [23].

### 2.3.2 OpenVPN

OpenVPN [24] 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.

### 2.3.3 Wireguard

Wireguard [5] is an open-source 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, increasing security robustness by avoiding SSL/TLS vulnerabilities commonly originating from such protocol negotiation.

#### Routing

Peers in a Wireguard communication maintain a data structure containing their own identification (both the 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 set of allowed source ips.

This structure is queried both for outgoing and incoming packets. To encrypt packets to be sent, the structure is consulted, and, based on the destination address, the desired peer's public key is retrieved. As for receiving data, after decryption (with the peer's own 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.

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 instead.

#### Cipher Suite

As aforementioned, Wireguard offers a single cipher suite for encryption and authentication mechanisms in its ecosystem. The peers' pre-shared keys are Curve25519 points [3], 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.

Regarding payload data cryptography, a Wireguard message's plain text is encrypted with the sender's public key and a nonce counter, using ChaCha20Poly1305, a Salsa20 variation [4]. The ChaCha cryptographic family offers robust resistance to cryptoanalytic methods [21], without sacrificing its state-of-the-art performance.

Finally, before any encrypted message exchange actually happens, Wireguard enforces a 1-Round Trip Time (RTT) handshake for symmetric key exchange (one for sending, and one for receiving). The messages involved in this handshake process follow a variation of the Noise [18] protocol, which is essentially a state machine controlled by a set of variables maintained by each party in the process.

#### Security

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

With such a scope in mind, 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, an operation requiring considerable CPU usage, 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 message, allowing the requested handshake to proceed later.

#### Basic Wireguard Configuration

Connecting two peers in a Wireguard communication can be done with minimal configuration. In fact, 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 (where it can be currently found). After both peers configure each other, the tunnel is established and packets can be transmitted through the Wireguard interface. In 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.

### 2.3.4 Performance Comparison

The concept of performance in VPN applications entails both protocol overhead on communication throughput and bandwidth usage minimization. These dimensions can be empirically measured, by calculating communication latency / ping time and throughput. The performance claims on [5], where, when comparing Wireguard to its alternatives like OpenVPN and IPsec, present results in favor of Wireguard in both metrics. This conclusion is

| | 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

backed by more extensive research [12], [15], where communication is tested in a wide range of different environments and CPU architectures.

Wireguard, due to its kernel implementation (compared to, for example, OpenVPN's user space implementation) and efficient multi-threading usage, contribute greatly to such performance benchmarks. Moreover, its relatively small codebase (around 4000 lines) creates a very auditable, maintainable VPN protocol.

## 2.4 Control Platforms

Although Wireguard proves itself to be a robust, performant and maintainable protocol for encrypted communication, it still presents some complexity regarding administration agility and scalability. New clients added to a standalone Wireguard network imply the manual reconfiguration of every other peer already present, a process with added complexity that is prone to errors, as more nodes join the system. With this in mind, this section explores applications and implementations of control 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.

First, it is mandatory to define what a control platform is. The main goal should be to overcome the limitations previously mentioned, by supporting:

- A centralized server storing peers' identification (public key and tunnel IP address).
- Establishment of secure channels between peers and such a centralized server.
- On-demand retrieving of information regarding any peer in its network domain.

### 2.4.1 OOR Map Server Implementation

An implementation with said requirements is proposed in [16]. The core architecture of this solution is composed by a centralized Open Overlay Router (OOR) Map Server, containing peer identification data, which provides devices with on-demand information regarding any other peer in the network to setup a direct connection. From a client perspective, a peer wanting to communicate with another should first establish a secure Wireguard connection to this server and request a connection with a destination node. The server, with the source IP and public key of the requesting client, redirects this data to the destination node, reaching a state where both peers contain all necessary information to begin the Wireguard tunnel.

This prototype successfully tackles one of the main limitations of Wireguard, offering a mechanism capable of dynamically configuring peers, without the need to reconfigure every device every time a new client joins the network. Also, it reduces routing table complexity, as peers are not required to keep all other peers' information locally. However, the addition of such a centralized entity also introduces a new attack vector. Effectively, if the private key of the central server, crucial in creating the first secure channel between a peer and the server, is compromised, a man-in-the-middle attack could be mounted, since an attacker could impersonate the centralized server.

Regarding performance, there is, as expected, an overhead compared to native OOR benchmarks, as requests to OOR Map Server are themselves conducted through a Wireguard channel.

### 2.4.2 Tailscale

Tailscale is a VPN service operating with a golang user-space Wireguard variant as its data plane [17]. Traditional VPN services operate under a hub-and-spoke architecture, a model composed of one or more VPN Gateways - devices accepting incoming connections from client nodes and forwarding the traffic to their final destination. Hub-and-spoke architectures carry some limitations. First, it implies increased latency associated with the geographical distance between a client and the nearest hub. Also, regarding scalability and dynamic configuration, adding new clients to the network requires the distribution of its keys to all hubs. With these constraints in mind, Tailscale offers a hybrid model. Tailscale's central entity, referred to as a coordination server, functions as a shared repository of peer information, used by clients to retrieve information regarding other nodes and establish on-demand P2P connections among each other.

This control plane approach differs from traditional hub-and-spoke since the coordination server carries nearly no trafic - it only serves encryption keys and peer information. Tailscale's architecture provides the best of both worlds, benefiting from the advantages of control plane centralization without bottlenecking its data plane performance.

In practical terms, a Tailscale client will store, on 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 Wireguard interface and start communicating with any other node in its domain.

### Overcoming network constraints

Tailscale also successfully supports procedures to overcome the problems described in the introduction of this section. 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 (essentially assuming that, if outbound traffic flowed to a destination, the source expects to receive an answer from that same destination), Tailscale keeps, in its coordination server, the *ip:port* 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 [20] and can solve this problem. For a peer to discover and store in the coordination server its own public *ip:port*, it first sends a packet to a STUN server. Upon receiving this packet, the STUN server can see which source address was used (the address NAT translated to) and replies with this value to the peer.

There are, however, some NAT devices that create a completely different public address mapping to each different destination a machine communicates with, which hinders the above address discovery process. Such devices are classified as Endpoint-Dependent Mapping (EDM) (in opposition to Endpoint-Independent Mapping (EIM)) [8].

Networks employing EDM devices and/or really strict firewall rules, such as blocking outgoing UDP entirely, 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 Hyper-Text Transfer Protocol (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 packets. These relay servers are distributed geographically. However, there is the possibility of an increase in latency and loss of bandwidth, which isn't terrible, as the alternative is not being able to establish connections at all.

As such, Tailscale's design provides a set of directives and infrastructure that work together to ensure Wireguard tunnels can be set up between any two peers, regardless of what policies the network between them employs.

### Headscale

While Tailscale's client is open source, its control server isn't. There is, however, an open-source, self-hosted alternative to Tailscale's control server, Headscale. Headscale [6] provides a narrow-scope implementation (with a single Tailscale private network) of the aforementioned control 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.

## 2.5 University of Aveiro Network

Due to security and privacy concerns, the specification of the 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 derived from observing its behavior.

First, the network is highly segmented, where clients are grouped according to their roles. In fact, 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. Then, the present NAT mechanisms in the network are unknown, as are their mappings' Time To Live (TTL), which is an important variable discussed in the previous sections. Finally, the network contains a segment for public services, which can be accessed from anywhere on the Internet. This public point of communication will allow clients from within the campus to connect to the control platform, serving in this domain.

# Chapter 3

# Methodology

Having outlined the relevant technologies for this dissertation's context, this chapter aims to present a work proposal for the solution implementation. The approach to be taken is segmented in three main stages: Prototype Development, Production Deployment and, finally, Automation.

## 3.1 Prototype Development

This first phase aims to build a small-scale prototype in a virtual development environment. The prototype should, using Headscale and Tailscale, be able to establish a P2P connection between two sample clients, which couldn't communicate beforehand. Hence, this phase's main goals are to (i) create a virtual environment which simulates the scenario being tackled, (ii) establish P2P connections between clients behind private NAT networks and (iii) validate the communication through the Robot Operating System (ROS) middleware. Obviously, regarding goal (i), the simulation of the networking conditions can't be entirely accuracte, since, as referenced in section 2.5, details regarding UA's network mechanisms are vastly unknown. It is possible, however, to create an analogous situation, where clients can't form P2P commmunications with each other but can all communicate with an external, public server. Moreover, as stated in previous sections, Tailscale's protocol effictevly deals with most network costraints 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.

This virtual environment must be composed of three entities: a public server and two clients in their own private networks. As mentioned in section 2.4.2, Headscale is an open-source implementation of Tailscale's control server. Thus, the control server in this environment is a self-hosted Headscale instance deployed in the public server. Both clients, which can reach the public server, but not each other, should then auhtenticate in this control server and configure their Tailscale interfaces and addresses, allowing for a direct communication to take place.

Such an enviornment can easily be established in a single host, using virtualization software, with the use of three linux virtual machines running on minimum resources. To achieve the networking requirements, as stated in goal (i), the client machines should be attached to individual private NAT Networks and connected via Wi-Fi to an AP, while the server machine should be attached to a bridge on the host's ethernet interface. Figure 3.1 depicts this architecture.

Finally, to validate goal (iii), a simple ROS application should be deployed and tested in the clients, in which communication is done through the Tailscale interfaces.

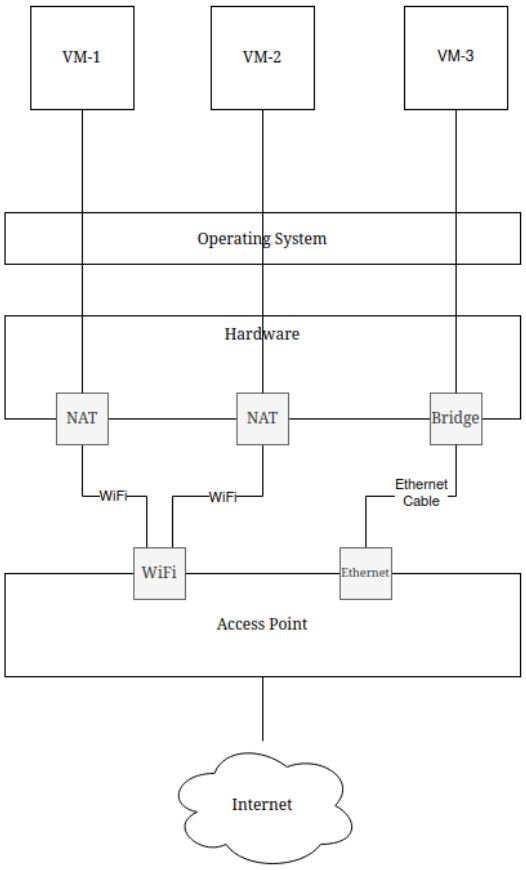


Figure 3.1: Development Enviornment Architecture

## 3.2 Deployment

After achieving and configuring the prototype previously developed, the next phase focuses on the deployment of the solution within UA's premises. As already mentioned, the Headscale instance should be hosted in a server residing in UA's public domain segment. This allows the control server to be reachable to any client within the campus. Here, the configuration should be based on the research done during the previous phase, with few eventual changes to ports and/or addresses. Regarding clients, ideally, each team of robots should have its own user, which individual robots will use for authentication. ¡TODO: COMO SERA O PROCESSO DE GERAR KEYS???¿.

Hence, this phase aims to (i) deploy an Headscale server in UA's public network domain, (ii) configure robots to act as Tailscale clients and (iii) establish communication within a team of robots spread out throughout the campus.

## 3.3 Automation

Automation of the processes taken in the previous phases, to make configuration with little to no manual intervention

## 3.4 Work Plan

The tasks encompassing the phases described above can be summed up in a Gantt diagram, presented in figure 3.2. The tasks to be carried out are as follows:

- **Development Environment Design and Requirement Analysis** - Setup of the development environment and minimum requirement analysis (identification of packets, accounts creation and domain definitions)
- **Prototype Development** - Implementation, in the development environment, of a prototype fulfilling the requirements. The end goal is to establish a P2P connection between the two machines that can't communicate.
- **Deployment** - Deployment of the control server in UA's public services domain and configuration of clients in the robots.
- **Validation** - Validation of the deployed solution, ensuring encrypted communication between nodes in different geographical locations within the campus.
- **Automation** - Development of config-based scripts automating client and server configurations.
- **Final Document Writing** - Writing of the final document, presenting the development process and providing analysis of respective results.

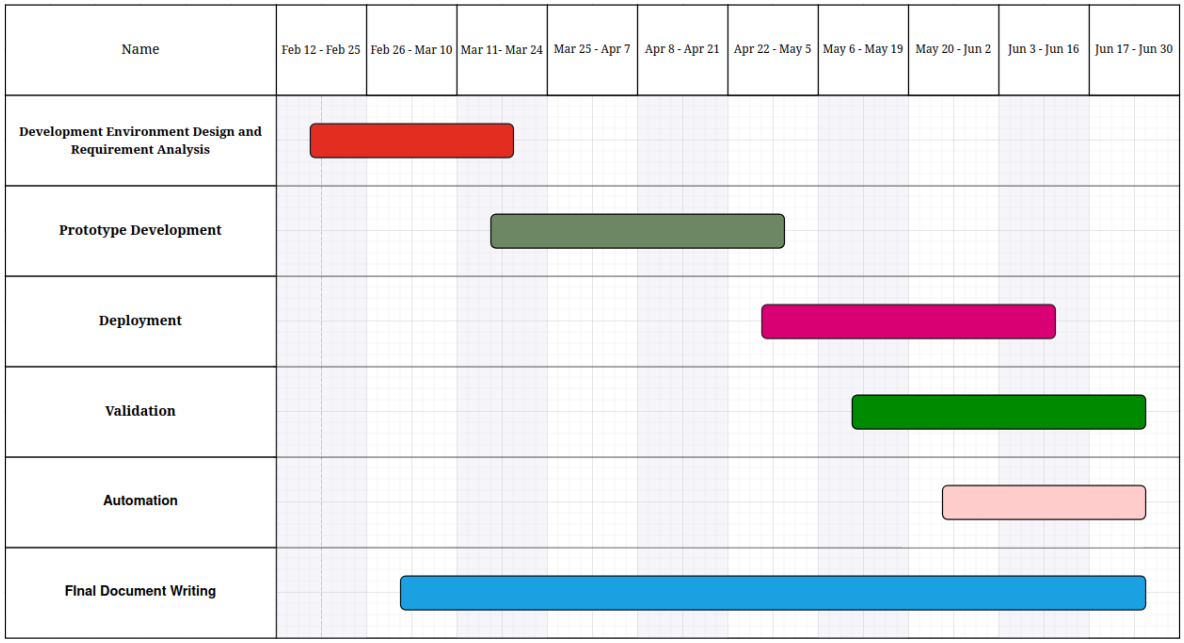


Figure 3.2: Development planning proposal

# Chapter 4

# Prototype Development

This chapter details the configuration of the development environment, as proposed in figure 3.1, and its use for the implementation of a prototype. This environment is managed with Oracle Virtual Box. Having defined the goals for this experiment in the aforestated work porposal, this prototype requires the deployment of the Headscale control server, which is hosted in **VM-3**, followed by the configuration, and respective authentication of both clients, **VM-1** and **VM-2**, using the Tailscale Command Line Interface (CLI). With the clients configured with Tailscale addresses, these can be used to run tests on ROS applications.

## 4.1 Development Environment

#### Virtual Machines

The three virtual machines composing the environment use *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 extremly 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 each client on its own private network, unaccessible from the outside. All machines can access the internet through the access point, explored in the following subsection.

Both **VM-1** and **VM-2** are assigned the same private IP address since they are residing in distinct private networks. For convenience, the Virtual Machines are also configured to allow Secure Shell (SSH) connections, which means port 22 is open on all machines. Moreover, for machines **VM-1** and **VM-2**, which are confined to their private networks, this was achieved with port forwarding rules, forwarding **VM-1**'s port 22 to the host machine's port 2222 and **VM-2**'s port 22 to the host machine's port 2223.

Table 4.1 summarizes said specification.

#### Access Point

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**, the traffic will be routed from the client to the AP's Wi-Fi interface. Then the router will forward the packet through

| | 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 4.1: Development Virtual Machines specification

| Service | Listen Address | Port | Description |
|---|---|---|---|
| Control Server | 192.168.10.214 | 8080 | Main service implementing the control plane |
| Metrics | 127.0.0.1 | 9090 | Exposes the /metrics endpoints, for monitoring |

Table 4.2: Services running in the Headscale instance

its ethernet interface, reaching the host machine and, consequently, reaching **VM-3**, as its network adapter is bridged to the host's ethernet interface, as described above.

## 4.2 Headscale Instance Deployment

Headscale provides an highly configurable open-source implementation of a control server, allowing the configuration of DERP relays and STUN servers, which must also be hosted in this server. At the time of writing, the latest stable Headscale release is *v0.22.3* [1], which is the version of the software refered to in the rest of this chapter.

The instance was configured to run with an embedded DERP server in a sample region, which additionally provides STUN functionalities to make NAT traversal possible within the environment. These configurations were achieved through Headscale's configuration file, a yaml provided in the package. With the service running, **VM-3** is now listening for Tailscale clients to connect and start using the protocol. For development purposes, an Headscale user, **dev**, was registred in the instance and shall be the user the clients will register themselves with. Finally, regarding Tailscale IP assignments, the 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.

Table 4.2 presents the available services and their respective listening configuration produced by the running Headscale instance. Besides the metrics and gRPC services, which are not required by clients to use the Tailscale protocol, are listening privately, on the server's localhost interface. The remaining services are exposed in the defined ports, reachable by the clients.

## 4.3 Client Configuration

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 tehy possess any information regarding one another. They can, however, reach the

[1] Headscale's official releases, hosted in GitHub. https://github.com/juanfont/headscale/releases.

| | 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 4.3: Clients Tailscale configuration after authentication

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. ¡TODO: explicar teoria de porque é que o tailscale vai funcionar aqui¿

Using Tailscale's CLI, a client is able to authenticate in the Headscale instance previously deployed, which in turn configures its Tailscale interface with a respective Tailscale IP, in the range previously configured in the control server. This will allow a state where P2P WireGuard tunnels can be established freely between the registered clients.

Hence, the Tailscale binaries were installed in each client, using Tailscale's install shell script [2].

At this point, clients are ready to perform authentication in the control server and start communicating, a process described in the next section.

## 4.4 Authentication

Authenticating in the Headscale instance can be done either using a pre-authenticated key, generated by the control 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 presenting itself as the option more adequate for this solution.

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. Clients are now able to authenticate with Tailscale's CLI, by using the **tailscale up** command. Two additional command-line parameters are set, the *login-server*, which points to the address Headscale is listening and the *authkey*, where the previously shared pre-authenticated key is injected.

With the clients authenticated, both devices are respectively assigned a Tailscale IP and hostname, used to communicate through the overlay network. Table 4.3 presents the clients' Tailscale configurations at this state.

## 4.5 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** [3] was

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

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

installed in both clients.

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

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

## 4.6 Chapter Summary

This chapter details a successful implementation of a minimal configuration prototype which simulates, in an analogous environment, the interactions between a self-hosted Headscale control server and its clients. In fact, the configured environment presented a situation where two nodes, due to their network topology, couldn't immediately establish a P2P connection, but could both reach a third node. Headscale was then introduced in the public node as a self-hosted Tailscale control server, which, after authentication operations via pre-shared keys, made it possible to connect the client nodes through the Tailscale interfaces. Finally, this tunnel was tested when used along with ROS, operating as expected. The procedure taken in this phase meets the goals outlined in section 3.1.

# Chapter 5

# Production Deployment

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

In this solution, the Headscale instance is hosted within IRIS-Lab's network. This, however, only allows communications from clients residing in the IRIS-Lab's 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, accessible within UA's network, which redirects traffic on ports 8080 (the main Headscale service and the embedded DERP) and 3478 (STUN interface) to the Headscale server. With this configuration, clients within the campus, which can directly reach the public facing proxy machine, can perform authentication. Figure 5.1 presents such architecture.

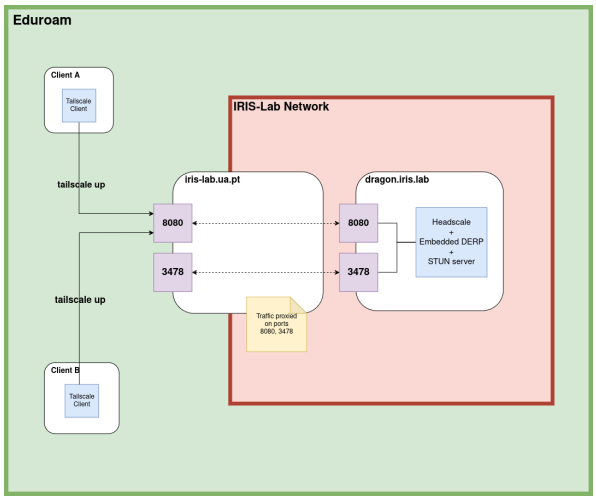


Figure 5.1: Production Deployment Architecture

Since Headscale's host is also an Ubuntu linux environment, installation of this service followed the process described in the development environment (see Section 4.2). Regarding configuration, however, the **server_url** parameter, which points to the endpoint clients should use to authenticate, is now the FQDN of the proxy server, *iris-lab.ua.pt*.

At this point, clients connected to UA's network are able to successfully perform authenticaton and, consequently, start establishing P2P WireGuard tunnels.

### 5.0.1 Self-Hosting DERP Infrastructure

Initially, Headscale was configured to use Tailscale's DERP server fleet, however, upon testing this scenario using the two development virtual machines as clients, the established communication is relayed, which means DERP servers are being used to redirect traffic over HTTP, as no other process was able to establish a P2P User Datagram Protocol (UDP) connection. Using Tailscale's DERPs implied the redirection of traffic to a DERP server in Germany, resulting in tunnels with an average latency of 452ms, a value unacceptable for the scope of this dissertation. Moreover, as this solution is meant to be used exclusevly by UA's clients, Tailscale's DERP fleet was discarded, opting for a self-hosted alternative, as packets should have no necessity of leaving UA's network.

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 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.

Using the embedded self-hosted DERP allows communication between clients to be much more efficient, with an average latency of 1ms, for the VM clients. Moreover, as clients can now perform NAT traversal due to the now accessible local STUN endpoints, this scenario can now establish direct, non relayed, connections.

## 5.1 Sample Clients

### 5.1.1 Configuring Clients

# Chapter 6

# Automation

# Chapter 7

# Validation and Results

# Chapter 8

# Conclusion

# Bibliography


[1] Andersen, D.G. and Balakrishnan, H. and Kaashoek, M.F. and Morris, R. The case for resilient overlay networks. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, 2001.

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

[3] Daniel 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.

[4] Daniel J Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, 2008.

[5] Jason A Donenfeld. Wireguard: next generation kernel network tunnel. In *NDSS*, 2017.

[6] Juan Font and Kritoffer Dalby. Headscale. `https://headscale.net/`, 2023.

[7] Sheila Frankel, Karen Kent, Ryan Lewkowski, Angela D Orebaugh, Ronald W Ritchey, and Steven R Sharma. Guide to ipsec vpns:. 2005.

[8] Cullen Fluffy Jennings and Francois Audet. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787, 2007.

[9] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 7296, 2014.

[10] Stephen Kent. IP Authentication Header. RFC 4302, 2005.

[11] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445, 2018.

[12] Steven Mackey, Ivan Mihov, Alex Nosenko, Francisco Vega, and Yuan Cheng. A performance comparison of WireGuard and OpenVPN. In *Proceedings of the Tenth ACM Conference on data and application security and privacy*, 2020.

[13] Apostolos Malatras. State-of-the-art survey on p2p overlay networks in pervasive computing environments. *Journal of Network and Computer Applications*, 2015.

[14] Tran Sy Nam, Hoang Van Thuc, and Nguyen Van Long. A High-Throughput Hardware Implementation of NAT Traversal For IPSEC VPN. *International Journal of Communication Networks and Information Security*, 2022.

[15] Lukas Osswald, Marco Haeberle, and Michael Menth. Performance comparison of vpn solutions. 2020.

[16] Jordi Paillisse, Alejandro Barcia, Albert Lopez, Alberto Rodriguez-Natal, Fabio Maino, and Albert Cabellos. A control plane for wireguard. In *2021 International Conference on Computer Communications and Networks (ICCCN)*, 2021.

[17] Avery Pennarun. How tailscale works. `https://tailscale.com/blog/how-tailscale-works/`, 2020.

[18] Trevor Perrin. The noise protocol framework. 2018.

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

[20] Marc Petit-Huguenin, Gonzalo Salgueiro, Jonathan Rosenberg, Dan Wing, Rohan Mahy, and Philip Matthews. Session Traversal Utilities for NAT (STUN). RFC 8489, 2020.

[21] Gordon Procter. A security analysis of the composition of chacha20 and poly1305. 2014.

[22] Karen Seo and Stephen Kent. Security Architecture for the Internet Protocol. RFC 4301, 2005.

[23] Chaman Singh and KL Bansal. NAT Traversal Capability and Keep-Alive Functionality with IPSec in IKEv2 Implementation, 2012.

[24] James Yonan. Openvpn. `https://openvpn.net/`.