Universidade de Aveiro
2023

Departamento de
Electrónica, Telecomunicações e Informática,

Vasco Regal Sousa

Multiple Client Wireguard Based Private and Secure Overlay Network

"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, scalibility 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 particular 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 evergrowing interest among any information system. Companies strive to ensure their communications follow principles of integrity and confidentially while minimizing attack vectors that could compromise services and data. With such goals in mind, network topologies are subjected to policies which apply rules and conditions to inbound and outbound traffic. One such mechanism is the use of Virtual Private Network (VPN) .

Traditional VPN services consist in the establishment of a secure, encrypted channel between a client and a network, through an insecure communication medium.

The University of Aveiro (UA)'s Intelligent Robotics and Systems Laboratory (IRIS-Lab) conducts research projects using autonomous mobile robots, which communicate through a Wi-Fi network. Currently, this network is confined to the premises of the IRIS-Lab, preventing the robots from operating in the remaining UA's buildings. Although the UA's Wi-Fi infrastructure covers most of its edifices, which can be used by the robots, due to security mechanisms, this network proves to be highly restraining, not allowing Peer to Peer (P2P) communications through the Robot Operating System (ROS) [21] - the operating system the robots run on - middleware without additional network equipments. Moreover, these constraints keep developers from being able to interact with the robots through their personal machines, which, if otherwise possible, would be of great interest.

1.2 Objectives

The main goal of this dissertation is to implement a private overlay network manager to be used exclusively by UA's clients. The concept of a manager entails both the definition of a network's client universe (which nodes should be allowed to connect to a certain network) and its respective identification and authentication mechanisms.

In the IRIS-Lab scenario, the management platform should provide operations to achieve communication between a team of robots, regardless of their physical location within the campus. Moreover, the authentication and connection to a desired overlay network by the robots must be a seemingless operation, requiring little to no manual configuration.

Finally, all traffic must be encrypted and properly authenticated, to ensure the privacy of the communication.

1.3 Document Structure

This document presents an implementation proposal of such an overlay network manager. Hence, it is structured in two main chapters - State of The Art and Methodology. The former consists in 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 while 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 Encrypted Peer to Peer Communications / VPNs

VPNs have become a mature technology, with widespread usage among 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 a network communication responsible for carrying data between devices. This implies not only the robustness of its authentication methods, encryption suite and protocol security but also of its features regarding concepts such as mobility - how the service behaves when clients change their physical locations and Internet Protocol (IP) adresses - 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 a computation overhead, 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 its features, cryptography and performance. The structure of the following paragraphs is loosely inspired by similar research and publications, namely [1].

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 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 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, specially 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 [12] and STUN [19]. Hence, each VPN service can have its own way of supporting NAT Traversal. Each case is explored seperatly in its own subsection.

2.1.1 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 secuirty mechanisms are applied upon 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 a secure communication to take place [22]. Here, unidirectional means a SA can only be associated to either inbound or outbound traffic, but never to 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) [10] is a negotation protocol which tackles the problems associated with manual SA installation. In fact, IKE allows the negotation 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 less 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 carries 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 [11]. 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 consists in the increment, by the sender, of a sequence integer whenever an outgoing message is processed.

AH, as the name suggests, operates by attaching an 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 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 adding of padding to packets, difficulting cipher analysis techniques. Moreover, it can be made compatible with NAT and employ NAT-traversal techniques [14], [23].

2.1.2 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 negotation 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 instatiates 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, sent to the TUN interface.

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

2.1.3 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 registred 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 originated from such protocol negotiation.

Routing

Peers in a Wireguard communication maintain a data structure containing its 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 aditional field, an internet endpoint, defining the listening address where packets should be sent. If not defined, the incoming packets’ 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 consist in Curve25519 points [3], an implementation of an eliptic-curve-Diffie-Hellman function, characterized by its strong

	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

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 nounce counter, using ChaCha20Poly1305, a Salsa20 variation [4]. The ChaCha cryptographic family offers robust resistance to cryptoanalytic methods [20], 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 - 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 leaks 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 an 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 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 pratical scenario, given two peers, *A* and *B*, with pre-generated keys and internet interfaces, presented on table 2.1, figure 2.1 presents the CLI steps to setup a minimal Wireguard communication, as specified in the official Wireguard documentation.

```
# 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 2.1: Basic Wireguard Communication Between Two Peers

2.1.4 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 compairing Wireguard to its alternatives like OpenVPN and IPsec, presents results in favor of Wireguard in both metrics. This conclusion is backed by more extensive research [13], [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.

2.2 Control Platforms

Although Wireguard proves itself as a robust, performant and maintainable protocol for encrypted communication, it still presents some complexity regarding administration agility and scalibility. New clients added to a standalone Wireguard network imply the manual reconfiguration of every other peer already present, a process with added complexity and 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 pontential 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.

2.2.1 OOR Map Server Implementation

An implementation with said requirements is proposed in [16]. The core architecture of this solution consists in 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 prespective, 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 reconfgiure every device everytime 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. Efectively, 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.

2.2.2 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 by 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 geographical distance between a client to the nearest hub. Also, regarding scalibility 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 an hybrid model. Tailscale's central entity, refered to as a coordination server, functions as a shared repository of peer information, used by clients to retrieve information regarding other nodes and establishing on-demand P2P connections among each other.

This control plane 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, benefitting from the advantages of control plane centralization without bottlenecking its data plane performance.

In pratical 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 in 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 pakcets 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 dont'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 [19] 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 this value to the peer.

There are however, some NAT devices that create a completly 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)) [9].

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 to relay 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, there is however the possibility of carrying 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 inbetween 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.

2.3 University of Aveiro Network

Chapter 3

Methodology

Having outlined the relevant technologies for this solution, Tailscale's implementation appears as the most appealing to produce a configurable overlay network manager with the leverage for configuration automation. Since, as mentioned in the previous section, Tailscale uses Wireguard as its data plane, this protocol also requires considerable attention. The methodology to be taken in this dissertation follows three main phases: devolpment, deployment and automation. Each is described as follows:

First, to better understand the mechanisms of said tools at a pratical level, a development environment, consisting in three LXLE (a lightweight, Ubuntu-based Linux environment) virtual machines will be setup as a sandbox for running experiments. Figure 3.1 depicts the proposed development architecture. In this environment, managed with virtualization software, **VM1** and **VM2** are attached to isolated NAT networks, which means they can't communicate with each other. They can, however, reach the Internet via WiFi through an access point. **VM3** is configured with a bridge network, and connected via Ethernet to the access point. So, **VM1** and **VM2** can reach **VM3** but can't reach other. This effectively creates a scenario very similar to the one being tackled, as our goal is to establish a P2P connection between two machines, when they can't communicate.

Regarding Wireguard, virtual machines can be perceived as protocol peers. As the hosts can communicate between themselves, we have a scenario where Wireguard tunnels can be setup. Although in Tailscale's scope Wireguard is considerably abstracted, this environment allows not only a better understanding of the underlying dataplane at stake, but also to collect metrics associated with network performance, using open-source tools, such as iperf [8]. Tailscale will then be introduced in the environment, with one machine serving as the control plane, or, by Tailscale's terms, the coordination server. Here, devices, users and subnetting configurations will be done. The other two machines will function as Tailscale clients and are orchestrated by the coordination server. Essentially, this first phase serves as an analysis on Tailscale's concepts, configurations and features, which will aid in specifying and understanding the requirements for the Tailscale instance to be deployed in the UA's premises. This environment can also eventually be used for the development of client automation mechanisms, which will be discussed further in the next phases.

Once Tailscale has been researched to its most adequated configuration, we proceed to the next phase, which, analogous to the previous one, consists in the setup of the coordination server in a host within the UA network, followed by the configuration of the clients, either using the robots running ROS or, in a first stage, a sample of test hosts, which can virtually

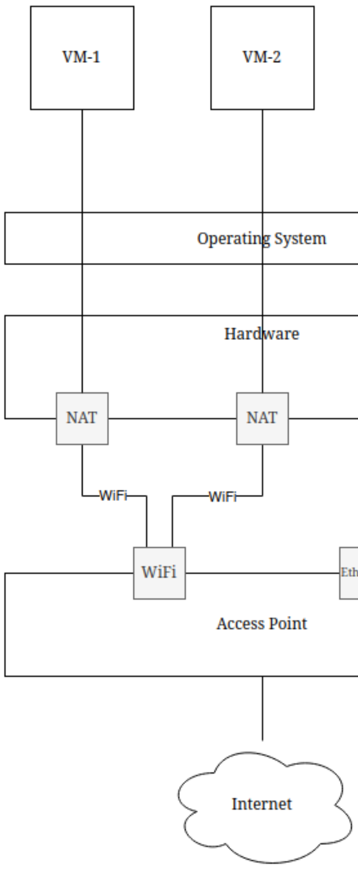
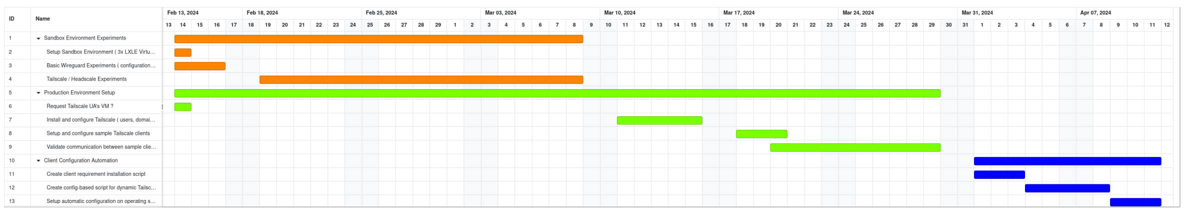


Figure 3.1: Development Enviornment Architecture

be any machine.

The final phase consists in the automation of the processes taken in the previous ones, regarding installation, deployment and configuration of both the coordination server and client-side running on the robots. This automation is achieved with the use of shell scripts and, eventually, open-source automation tools such as Ansible [2]. To configure the clients, a config-based script will be developed. This approach allows the use of the same script for all robots requiring access to an overlay network, with a degree of abstraction regarding operations such as key generations and service discovery. Ideally, this script would only require the configuration of the coordination server's address, the name of the network the robot wants to connect to and, optionally, an endpoint pointing to where the robot can be found. Another topic of great interest would be to ensure the robots automatically connect to their desired network on boot, which can be achieved with the use of, for example, crontabs.

The tasks encompassing the phases described above can be summed up in a Gantt diagram, presented in figure 3.2.



ID	Name
1	Sandbox Environment Experiments
2	Setup Sandbox Environment (3x LXLE Virtu…
3	Basic Wireguard Experiments (configuration…
4	Tailscale / Headscale Experiments
5	Production Environment Setup
6	Request Tailscale UA's VM ?
7	Install and configure Tailscale (users, domai…
8	Setup and configure sample Tailscale clients
9	Validate communication between sample clie…
10	Client Configuration Automation
11	Create client requirement installation script
12	Create config-based script for dynamic Tailsc…
13	Setup automatic configuration on operating s…

Figure 3.2: Development planning proposal

Bibliography

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

[2] Red Hat Ansible. Ansible is simple it automation. `https://www.ansible.com/`.

[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] Vivien Gueant. iperf - the tcp, udp and sctp network bandwidth measurement tool. `https://iperf.fr/`.

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

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

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

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

[13] 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.

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

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

[21] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, 2009.

[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/`.