UNIVERSITY OF AMSTERDAM

PROJECT REPORT

Peer-to-peer VPN solution for eduVPN using WireGuard

February 10, 2024

Students:
Marijn Valks
15080129

Robin Slot
12654264

Institution:
SURF, GÉANT and DeiC

Institution supervisor:
Rogier Spoor, Jeroen Wijenbergh
and François Kooman

Course:
Security and Network Engineering

Abstract

eduVPN is a VPN solution for education and research institutions, offered by GÉANT and co-developed by SURF and DeiC. It supports OpenVPN and WireGuard. Communication between two peers on the VPN is always relayed through the central server, following the hub-and-spoke model. This introduces unwanted latency, and increases resource usage for the VPN server. In this research project, we will investigate ways of introducing peer-to-peer connections into eduVPN's WireGuard system. Peer-to-peer connections are especially useful in combination with user-defined networks functionality, where an eduVPN user can create their own private networks with custom network prefixes and static IP addresses for their devices.

1 Introduction

With more people working and collaborating remotely, the need for secure access to data and easy collaboration has increased significantly. Without a physical network connecting development servers and workstations, a virtual network is needed to securely tunnel traffic over the internet.

The current eduVPN service provides VPN tunnels using OpenVPN or WireGuard. However, in this system the clients connect to a centralized server. Even when connecting to a device close by, traffic has to be relayed via the centralized VPN server. Not only does this unnecessarily increase latency and decreases possible throughput, such a server is costly to operate and is a single point of failure. The client-server VPN model is traditionally more designed to facilitate access to resources within one organization and is therefore less ideal for facilitating collaboration between people from different organizations and institutions. In recent years, the demand for collaboration between people from different organizations, even internationally, has increased. A peer-to-peer VPN model would provide better scaling options by connecting clients directly, making them independent of the performance of central VPN servers.

Peer-to-peer VPN technology would enable users to create mesh networks where every peer is directly connected. Only in the rare case where a peer-to-peer connection cannot be made, a relay server can be used. We will research methods for establishing connections between peers behind firewalls and NAT, examining their feasibility for integration into eduVPN. This includes investigating different NAT types and evaluating the possibility of employing hole punching to create peer-to-peer connections between them. The technique of hole punching, which involves opening network ports on NAT devices, is studied for reliable integration with WireGuard. In cases where hole punching is not feasible, a central relay server is employed to seamlessly integrate into the existing eduVPN architecture.

1.1 Structure

The paper is structured as follows: In section 3, we discuss earlier relevant work. Section 4 covers essential background information about relevant topics that are crucial for understanding our research. The methodology section 5 highlights our methodology, including an explanation of different NAT traversal techniques. The research results, along with a description of the technical workings of the newly developed solution, are examined in section 6, and the findings are discussed in section 7. Finally, section 8 presents the conclusion of the findings and in section 9, some topics for future work are discussed.

2 Research question

2.1 Research Question

In the paper, we will answer the following research question: **How to design a peer-to-peer VPN solution for the current eduVPN infrastructure, using WireGuard?** To answer this question, we define the following sub-questions:

- How can WireGuard be made to establish connections using UDP hole punching, without modifying WireGuard itself?
- How can WireGuard be made to establish connections via a relay server, without modifying WireGuard itself?
- What is the feasibility of each method of receiving UDP datagrams, as discussed in the abstract?

2.2 Scope

The scope of the project will be limited in different aspects:

- The research is limited to the WireGuard VPN protocol. Altough the availability of various alternatives like Tinc, n2n, Tailscale/Headscale, Zerotier, and Nebula, the project will exclusively rely on the WireGuard VPN protocol. The primary reason for this limitation is that WireGuard is deeply integrated into eduVPN, and altering this integration would require a substantial effort. Additionally, Aquina, an earlier intern, conducted a comprehensive study on the integration of WireGuard into the eduVPN architecture [1]. Lastly, the effectiveness of the WireGuard architecture has been proven, considering an alternative VPN architecture would only be justified if the advantages significantly outweigh the disadvantages.

- Client application development is beyond the project's scope; The eduVPN team is responsible for developing user clients for various supported operating systems. This encompasses client authentication, logging, and configuration polling through the eduVPN portal. During the project's finalization phase, the modified client configuration will be transferred to SURF's responsible team.

- WireGuard only supports UDP. The team behind eduVPN is experimenting with solutions of encapsulating WireGuard traffic in TCP segments[1], to support networks where UDP is blocked. However, the current state of WireGuard in eduVPN depends on UDP. In this research project, networks where UDP is blocked will not be considered.

3 Related work

Küthe conducted a comparison of various mesh VPN solutions, including Tinc, n2n, Tailscale/Headscale, ZeroTier and Nebula [2]. Although the comparison provided valuable insights into the solutions and their possibilities, none of them seamlessly integrates with the current eduVPN architecture.

Tailscale presented a solution [3] for establishing a peer-to-peer network using WireGuard and NAT-traversal in combination with a central coordination server. Headscale[2] on the other hand developed an open-source, self-hosted implementation of the Tailscale coordination server. Although these solutions do use WireGuard as their authentication and encryption protocol, they require a completely new eduVPN architecture (conflicting with the project scope). Despite this, they do provide some valuable insights into their approach using WireGuard as a protocol and using NAT-traversal [4] for building peer-to-peer mesh networks.

WireGuard, as a communication protocol for providing lightweight encrypted VPN networks, is widely adopted and researched in various projects. However, these projects use the WireGuard protocol as part of a large codebase. eduVPN on the other hand aims to use WireGuard without modification, instead only providing a simple management daemon. This is advantageous for maintainability and robustness.

Aquina, an earlier research intern at SURF [1] investigated eduVPN itself and how WireGuard could be integrated. At that time, eduVPN only used OpenVPN to provide their service. The paper explored the possibilities of enhancing eduVPN using WireGuard, but it didn't mention anything about a peer-to-peer solution.

4 Background

This section covers the necessary background information to understand the rest of the paper.

[1]`https://codeberg.org/eduVPN/proxyguard`
[2]`https://github.com/juanfont/headscale`

4.1 Current eduVPN architecture

eduVPN currently employs a combination of OpenVPN and WireGuard. As OpenVPN is considered more of a legacy environment, it will not be further investigated during this research. The reason for it to be considered legacy is the simplicity that comes with WireGuard, resulting in a more lightweight solution compared to OpenVPN, making it faster and reducing security risks. With a much smaller codebase, there is potentially a lower chance of security implications.

The eduVPN hub-and-spoke model allows for the routing of all traffic through the VPN or selectively routing specific traffic, supporting both IPv4 and IPv6. The client itself is available for Windows, macOS, Linux, iOS, and Android, each having its own codebase. eduVPN users can either pull a WireGuard configuration from the web portal or use an eduVPN client to connect. For a more comprehensive list of features and access to the complete eduVPN codebase, please refer to the eduVPN documentation [5].

4.2 WireGuard architecture

WireGuard is an open-source communication protocol designed to create lightweight encrypted VPN networks. It requires minimal user configuration and therefore is easy to implement. Each peer in the network is identified by a key pair. Users are responsible for exchanging their public key with other peers and configuring peer endpoints with the correct address. WireGuard utilizes advanced cryptographic algorithms, including `ChaCha20` for symmetric encryption, `Poly1305` hash function for authentication, `Curve25519` for the Diffie–Hellman key agreement protocol, and `SipHash24` & `BLAKE2` for private key management[6]. This combined with WireGuard's great performance and minimal codebase, makes WireGuard an ideal protocol for peer-to-peer solutions.

To establish a WireGuard connection between two peers, one must be reachable via a known address and port, while the other client may be roaming. If a client is behind NAT, WireGuard can periodically send keepalive messages to ensure mappings are persisted by stateful firewalls and NAT devices.

4.3 Different NAT types

When constructing a peer-to-peer solution, a thorough understanding of various types of NATs is crucial. RFC 3489 (STUN) categorizes four distinct NAT variations [7]:

- **Full cone NAT**, also referred to as 1:1 NAT or static NAT, consistently maps the same internal IP address and port for all outgoing requests to a corresponding external IP address and port. This ensures that all incoming traffic from external hosts can use this mapped IP address and port to forward the traffic to an internal host.

- **Address-restricted cone NAT** is restricted to a specific fixed IP, as indicated by the name. It requires an internal host to send a packet from its IP address and port to a fixed external IP address with a corresponding port before permitting any incoming packets from that same fixed external host.

- **Port-restricted cone NAT** operates on the same principle as address-restricted cone NAT, with the additional requirement of responses being from the same source port. Any request from an internal IP and port is mapped to a unique external IP address and port. Incoming packets are only permitted if sent from an external host port that previously received a packet on that specific port.

- **Symmetric NAT** uniquely maps the internal IP address and port to an external IP address and port for each distinct destination. This means that the packet's source IP and port are determined by the destination IP. Even if the same internal IP address and source send out a packet, a different mapping will be used for each new destination address. Symmetric NAT is incompatible with UDP hole punching [7].

RFC 4787 categorizes similar variations of NAT:

- **Endpoint-Independent Mapping (EIM)** is for devices that choose external source ports and addresses independently of the destination address (i.e., endpoint). In RFC 3489 terms, symmetric NAT is an example of a NAT that is *not* EIM, it is endpoint dependent.
- **Address-Dependent Mapping (ADM)** is comparable to address-restricted cone NAT (if EIM). ADM NAT can be EIM-NAT or not EIM-NAT.
- **Address and Port-Dependent Mapping (APDM)** is comparable to port-restricted cone NAT (if EIM). APDM NAT can be EIM-NAT or not EIM-NAT.

In this research paper, full cone NAT, address-restricted cone NAT, and port-restricted cone NAT are referred to as Endpoint-Independent Mapping NAT (EIM-NAT), adhering to the naming convention observed in [8, §2.9].

RFC 4787 states that NAT devices must use endpoint-independent mappings, i.e. not operate like symmetric NAT. The authors state that symmetric NAT has no security advantages, and has the large disadvantage of preventing peer-to-peer communication using UDP hole punching [9, § 4.1]. Still, there are many symmetric NAT devices operating in the field. There is limited data on the number of users behind symmetric NAT, where hole punching is not possible. In [10], the frequency of symmetric NAT is estimated to be around 8%, meaning that in 0.6% of cases the relay server would be required, assuming UDP is permitted. Halkes and Pouwelse [11] estimates that around 85% of networks use EIM and around 11% non-EIM. In other cases, UDP was blocked entirely.

4.4 UDP hole punching

Peer-to-peer communication relies on a direct interconnected connection between two peers. This necessitates an open (in this case) UDP port for sending packets between the nodes. UDP hole punching is a technique that could be employed to open up these network ports. Hole punching can be used in combination with EIM-NAT, but in the scenario of symmetric NAT, hole punching is not technically feasible [7].

UDP hole punching operates by having two parties send UDP packets to the other party. To do this, both parties must know the external IP address and external source port that correspond to their UDP requests. As depicted in figure 1, peer A and B use a server on the open internet for this purpose, most commonly a STUN server. Peer A and B send a message to the STUN server from an arbitrary source port. The STUN server replies with the peer's external addresses and source ports. After exchanging the external address information between peers, peer A can send a UDP datagram to B's external address, and B can do the same. Even though the endpoint for the second datagram is different, a NAT device using EIM-NAT will use the same external source port for the request. Peer A's NAT device now has an entry accepting responses from B, and Peer B's NAT device will accept datagrams from A.

4.5 STUN & TURN

A STUN (Session Traversal Utilities for NAT) server is a simple, lightweight server that returns the requester's external IP address and port in a response body. By issuing multiple specific requests to a TURN server, the existence of NAT and its type can be determined. This information provided by STUN can be used to perform UDP hole punching for establishing peer-to-peer connectivity. When UDP hole punching is not possible, like with symmetric NAT, a TURN (Traversal Using Relay NAT) server can be used as a relay or proxy.

5 Methodology

This section will discuss the approach and various techniques used to realize a peer-to-peer solution for various NAT scenarios. First, the various techniques to traverse NATs are ex-

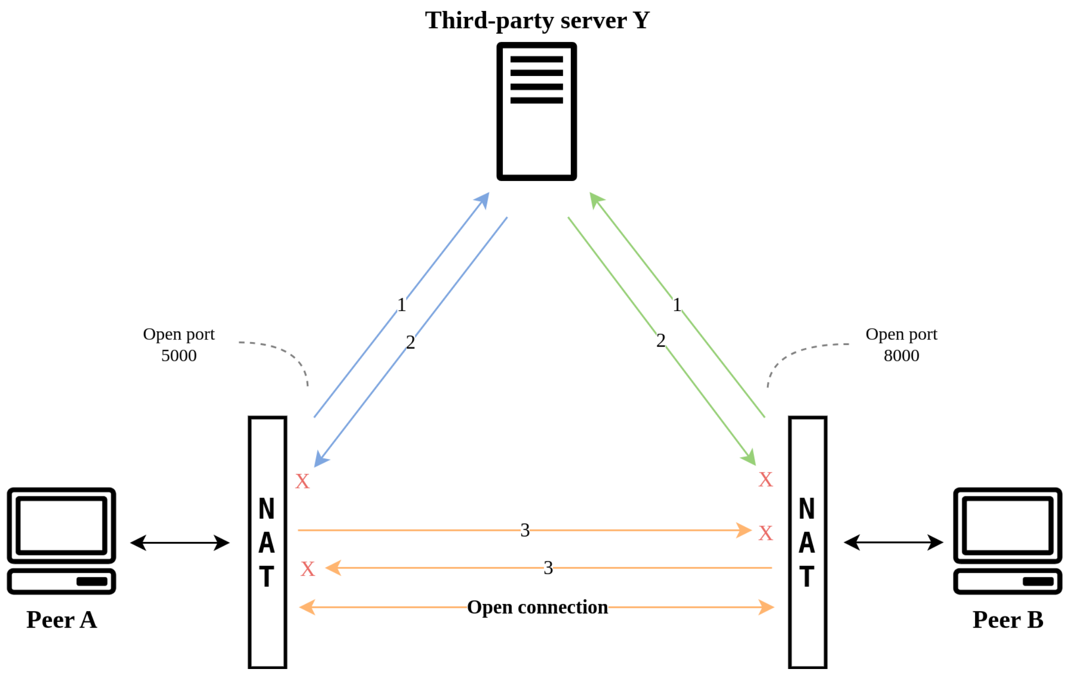

Figure 1: UDP Holepunching

plained, followed by an overview of the test methodology. In the end, some design principles regarding the project are explained.

5.1 Techniques used to traverse NATs

In section 4.3, we discussed various NAT types. EIM-NAT necessitates a distinct NAT traversal approach compared to symmetric NAT. Besides these two approaches, there are other approaches that can facilitate peer-to-peer connections through firewalls or NAT routers[8]:

- If the device has a direct connection to the internet without a firewall (IPv4 or IPv6), a direct connection can be made.
- If the device is behind a firewall, but has an IPv6 Globally Reachable Address (GUA) or external IPv4 address, UDP hole punching can be used to allow traffic through the firewall.
- If the device is behind NAT, UDP hole punching can be used to forward traffic to the device, as long as the NAT router does not use Symmetric NAT. UDP hole punching works with many types of NAT, as long as it is endpoint independent (IEM-NAT).
- If the device is in a network where UPnP port forwarding is allowed, it can request the NAT router to forward traffic to itself.
- In other cases, a proxy server must be used to relay traffic to the device. The device creates an outgoing connection to the relay server.

5.2 Testing methodoloy

An important aspect of our research is determining the NAT mode for all possible scenarios, as this forms the starting point of our investigation. The new solution must be developed with the various NAT scenarios in mind. For example, if many practical scenarios use Symmetric NAT, a different project approach must be adopted compared to the case where most scenarios use EIM NAT.

To test this, a STUN client is used. `Pystun3`[3] serves as both a library for programmers and a command line STUN client written in Python. `Pystun3` implements the STUN protocol and returns the NAT type, external source IP address, and external source port used to connect to either a public STUN server or a self-defined STUN server as a response. The discovered IP address and port could be used to perform UDP hole punching; i.e. to create a firewall or NAT table entry as explained in section 4.4. The table entry allows incoming traffic from a specific IP address and port to enter the network for a device-dependent time period. RFC 4787[9] recommends a duration of 5 minutes or more for UDP NAT mappings, but this may differ in practice. Most vendors use a lower duration as the default configuration. This testing principle is used to test whether direct peer-to-peer connections are genuinely possible for EIM-NATs, using the following test flow:

1. Run ‘pystun3’ on both peer A and peer B to request NAT type, IP address, and port.
2. Use this information to send a UDP packet from peer A to peer B and vice versa.
3. This creates a UDP table entry for the external IP address and port of peer A to peer B and vice versa.
4. Within the time range of the UDP table entry, attempt to establish a WireGuard connection to the other peer’s IP address and port.

This testing flow using `pystun3` is employed to determine the following practical scenarios:

- A client behind a router with Symmetric NAT
- A client behind double port restricted NAT
- A client behind a incoming firewall without NAT
- A client with an open internet connection
- A client behind different combination of everything described above

5.3 Preserving the integrity of the WireGuard protocol

One of the project goals is to preserve the integrity of the WireGuard protocol. The main reason for this is that it keeps the integration with the client simple and straightforward. In addition to various implementation benefits, it also simplifies various management tasks of the eduVPN client. By using the original WireGuard codebase without making modifications for a particular use case, the responsibility for addressing security vulnerabilities in the WireGuard codebase remains with the contributors of WireGuard.

Additionally, on Linux it means that the existing WireGuard implementation can be leveraged via NetworkManager. By shifting the responsibility of managing network interfaces, the root access requirement is eliminated.

5.4 NetworkManager and rootless operation

Setting up a WireGuard tunnel on Linux would require root access. To keep attack surface to a minimum, this should be avoided. NetworkManager is a system service found in most Linux desktop installations, that allows regular users to manage network connections. It can be controlled via D-Bus, a middleware message bus for the Linux desktop. This D-Bus API can be used from a Python application using PyGObject[4]. The proof of concept should have an operation mode using NetworkManager, to confirm it is possible to dynamically add and remove peers using NetworkManager.

Root access is still required for UDP hole punching as RAW sockets are being used. RAW sockets have direct access to lower-layer protocols, providing more capabilities than regular

[3]https://github.com/talkiq/pystun3
[4]https://pygobject.readthedocs.io/en/latest/

sockets but requiring root privileges. However, this part of the code can be extracted to a separate binary, so it can be run as root while the rest of the application is run as a normal user. This code is only responsible for sending a single UDP datagram.

5.5 User-defined network approach

We considered two approaches of building a peer-to-peer VPN solution. One would be to adapt the existing network via a central server, to build additional tunnels between peers on that network. There would be some algorithm to decide when to 'upgrade' connections to a peer-to-peer connection. We could not think of a robust way of implementing these connection upgrades and downgrades. In addition, there was concern whether such a system would have use cases, or whether it was a solution in search of a problem.

The alternative chosen approach, as suggested by F. Kooman (the original architect of eduVPN), involves using user-defined networks to build fully peer-to-peer mesh networks separate from the standard VPN solution. Users would be able to create *user-defined networks* with custom private subnets. They would be able to assign static IP addresses to devices on their network. This user-defined network would function like a virtual LAN, where all devices are reachable but can be physically located anywhere.

Due to the potential overlap of IP ranges, the central server can't participate in these VPN networks. As a result, the central server cannot serve as a fallback option when peer-to-peer connections are not possible. A custom relay server must be built to relay traffic between these isolated peers.

6 Results

6.1 Different scenarios

In section 5.1, various techniques for traversing NATs are discussed, all of which have been thoroughly researched and tested in practice. UDP hole punching is effective for scenarios where a device is connected to the internet with Endpoint-Independent Mapping NAT (EIM-NAT).

In the final program, the management server also functions as a STUN server. For our proof of concept, only very basic STUN functionality was needed, so it was easier and simpler to implement it ourselves without using `pystun3`.

Table 1 shows possible network configurations for both peers, and whether a peer-to-peer connection is possible. In all cases, at least a slower connection is possible through the relay server. In the vast majority of cases, a peer-to-peer connection is possible. As a comparison, table 2 shows when connections are possible with the standard WireGuard client.

In very restricted networks where UDP is blocked, `proxyguard` [7] can be used to encapsulate WireGuard's UDP datagrams in a TCP connection. This was not implemented in the proof of concept, but it should be considered in future work.

	Direct[5]	FW[6]	EIM-NAT	Symmetric NAT	UDP blocked
Direct	P2P	P2P	P2P	P2P	Relay + `proxyguard`[7]
No NAT w/ FW		P2P	P2P	P2P	Relay + `proxyguard`[7]
EIM-NAT			P2P	P2P	Relay + `proxyguard`[7]
Symmetric NAT				Relay	Relay + `proxyguard`[7]
UDP blocked					Relay + `proxyguard`[7]

Table 1: Possible connections using our solution.

[5] Also applies in situations where NAT is used but bypassed using manual port forwarding
[6] Firewall for incoming connections
[7] TCP proxy for WireGuard: `https://codeberg.org/eduVPN/proxyguard`
[8] PersistentKeepalive required on one end

	Direct[5]	FW[6]	EIM-NAT	Symmetric NAT	UDP blocked
Direct	Yes	Yes[8]	Yes[8]	Yes[8]	No
No NAT w/ FW		No	No	No	No
EIM-NAT			No	No	No
Symmetric NAT				No	No
UDP blocked					No

Table 2: Possible peer-to-peer connections using standard WireGuard.

6.2 Feasibility of UPnP

Automated port forwarding via UPnP, NAT-PMP, or PCP would provide a solution for peer-to-peer connectivity in difficult NAT situations, like symmetric NAT on both ends of a connection. However, when testing UPnP in the wild using MiniUPnP[9] we found that consumer routers have a strict throughput limit on connections via ports opened using UPnP. The TP-Link Archer C7 had a limit of 100Mbit/s. This already negates one advantage of peer-to-peer connections; higher throughput. Other devices, such as modem routers provided by the Dutch residential ISP Ziggo, have a limit of only 1Mbit/s, completely invalidating UPnP as an option for establishing high-throughput VPN tunnels. These limits were reported by the routers. We confirmed that they were accurate by opening a port via UPnP and testing throughput using `iperf3`.

6.3 eduVPN general working

The eduVPN peer-to-peer solution has two operating modes, either a direct peer-to-peer connection or a connection through a relay in cases where a direct peer-to-peer connection is not feasible. As mentioned in section 5.3, the goal was to maintain the integrity of the WireGuard protocol. This has the result that in both cases, the UDP packet used by the WireGuard protocol remains identical. Figure 2 visualizes the two operating modes: on the left side, a peer-to-peer connection is made between two clients by using hole punching and on the right side, a relay server is used to connect two clients.

Further details about both operating modes are provided in sections 6.6 and 6.7 respectively.

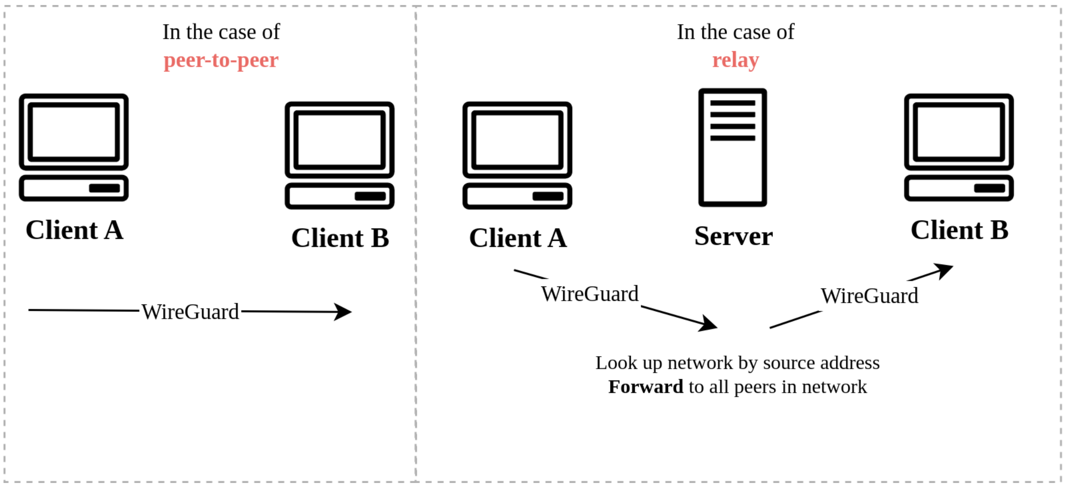

Figure 2: WireGuard messages, comparing peer-to-peer (left) scenario and relaying (right)

[9]`http://miniupnp.free.fr/`

6.4 Separation of management connection and data stream

Apart from the regular WireGuard data stream, the new eduVPN solution features a dedicated management connection between each client and the management server. This management connection helps solve the scaling issue of WireGuard itself. WireGuard relies on configuration files or commands for configuring and updating peers. However, there is no direct support for managing these peers as required in the eduVPN use case. Since eduVPN is designed as a user-friendly VPN solution, users are not expected to handle their own WireGuard configurations. Therefore, the new solution must provide a mechanism to add or remove peers or inform other peers to request a configuration update. This is achieved by providing an additional management connection alongside the regular WireGuard datastream.

6.5 Connection process

The management connection establishes a continuous link between all clients and the management server, relying on a TCP session. Figure 3 illustrates the initial connection flow and the message payloads between Client A and the server. The connection begins with the client generating a WireGuard key pair. The first two messages of the handshake between the client and server use UDP. The client sends a magic packet to the known server address and port. The server receives this and responds with an `AddressResponse` packet, based on the same principle as STUN, which retrieves the external source IP address and port of Client A used to connect to the server. At this point, Client A knows its external IP address and port. It will create a WireGuard interface using the same source port as the port used for the initial UDP communication. That means that the external source port will be also be the same, now known to the client via the `AddressResponse`.

Simultaneously, the client opens a TCP connection to the management channel, in a new thread. The client sends the first message `PeerHello` to the server and keeps the management socket active, awaiting replies. The server actively listens for `PeerHello` messages. When such a message is received, it responds with a `PeerList` message, containing all currently known peers in the network. It will also send the same `PeerHello` message to all other clients already connected to the server, so they can add the newly connected peer.

When the client receives `PeerList`, it calls a function to update its peers. This function initiates the hole-punching process using a raw UDP socket, as explained in section 4.4. After the hole-punching process, the clients add the peers to their WireGuard interface. If peers use EIM-NAT, a peer-to-peer connection starts working, if not it falls back to the relay scenario.

6.6 eduVPN peer-to-peer hole punching working

After the client successfully connects to the central server and receives the `PeerList` with the `PeerInfo`, it starts a new thread for every `PeerInfo` using a setup peer connection function. This function initiates the hole-punching process for all peers simultaneously. The management connection uses the same address and port as the WireGuard connection. Due to the ongoing management connection, hole-punching is blocked resulting in an 'address is already in use' error. RAW sockets can provide a solution for that as they have direct access to lower-layer protocols and simply do not check for double addresses. The RAW sockets send a UDP datagram packet to the host IP address and port of the other peer, using its own source address with the same source port as used earlier. This results in a NAT table entry for the given address and port.

6.7 eduVPN peer-to-peer relay working

The fallback from hole punching to relay uses an update peer function. This function changes the peer endpoint from the address of the other peer to the relay server address.

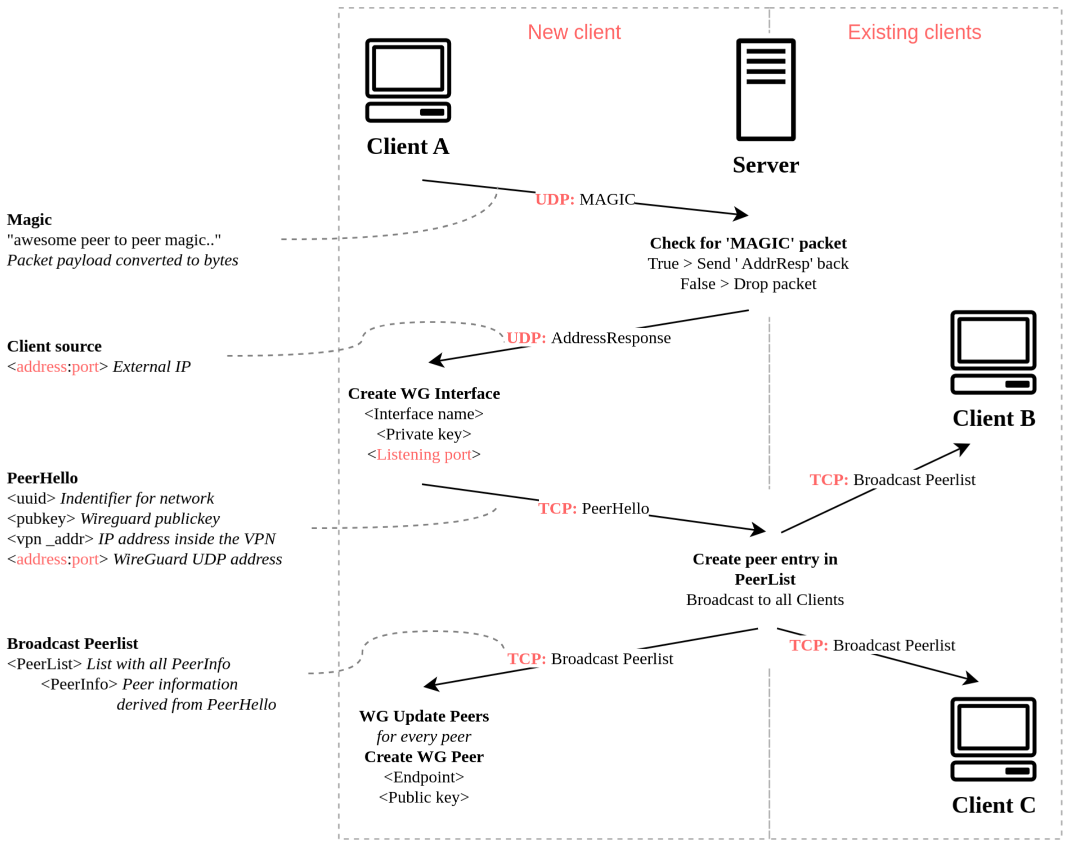

Figure 3: New client joining a network

The relay server on the other hand actively listens for WireGuard data packets on its configured listening port. When the server receives a data packet, it checks the sender's address and looks to see if it is stored in the `Network_By_Address` dictionary. If so, it transmits the packet to all addresses within the same network group; otherwise, it discards the packet. Therefore, apart from the endpoint address there is no client-side configuration difference between a peer-to-peer connection or a relay connection.

6.8 Determination of peer-to-peer or relay

As described in section 6.1, our solution is feasible for every different NAT scenario. However, in some cases a direct peer-to-peer connection is not possible, and a relay server must be used. The client employs a principle to determine whether a direct peer-to-peer solution is possible or if it must use the relay as a fallback. It accomplishes this by following the steps described below:

1. Perform UDP hole punching
2. Set up a WireGuard peer-to-peer connection with `PersistentKeepalive=1`
3. Monitor the RX bytes (received bytes) for the newly set up peer
    (a) If RX bytes increases the peer-to-peer connection must be successful, set `PersistentKeepalive=25`
    (b) If RX bytes does not increase the peer-to-peer connection failed, set endpoint to relay server and set `PersistentKeepalive=25`

After the client determines that a peer-to-peer connection is not possible, it switches to relay mode and changes its WireGuard peer endpoint to the address of the relay server. By doing this, the peer will no longer send messages to the other peer's address but to the relay server. This interface switch occurs on both clients through their management connections. This management connection is also used to adjust the `PersistentKeepalive` from 1 second to 25 seconds. This is done in both scenarios: whether a peer-to-peer connection is feasible or if the switch to the relay server is implemented.

6.9 IPv6

A WireGuard interface can only listen on one port. Hence, on dual stack IPv4 and IPv6 hosts, hole punching must done using the same local source port for IPv4 and IPv6. If NAT is used for both IPv4 and IPv6, two external source ports will be obtained. This is no problem, the management server distributes IPv4 and IPv6 IP addresses and port numbers. A connecting client can choose between the two. Only between IPv4-only hosts and IPv6-only hosts, peer to peer communication is not possible. This is not a limitation of our proof of concept, but of networking in general.

Figure 3 shows in which cases a peer to peer connection is possible. 'P2P' means a peer-to-peer connection is attempted, but it may still fall back to relay mode for reasons explained in section 6.1.

	IPv4	IPv4+IPv6	IPv6
IPv4	P2P	P2P	Relay
IPv4+IPv6		P2P	P2P
IPv6			P2P

Table 3: Operation modes using different address families.

6.10 Performance improvement

The performance improvement of the new eduVPN solution compared to the old solution is challenging to measure. The peer-to-peer architecture generally provides a faster connection to other endpoints than the old central server VPN architecture. Performance is a combination of bandwidth and latency, where bandwidth doesn't improve using peer-to-peer connections as it is mostly based on the client and internet service provider network bandwidth. The peer-to-peer connection probably uses the same internet connection as the regular central server model. Bandwidth could only be improved if the central server model had client restrictions or bottlenecks in its VPN server. When multiple clients use the client-server VPN model, they share the CPU resources of the central VPN server. It could be the case that the CPU of this server does not have the capabilities to provide enough resources to every individual client. In such cases, a peer-to-peer connection could improve performance as it doesn't rely on a central server with bandwidth restrictions or bottlenecks.

Discussing improvements in latency is as intricate as discussing enhancements in bandwidth. By removing the central server from the traditional solution, the total number of hops[10] from client to client reduces, as the peer-to-peer solution connects directly. However, the performance depends on the specific situation. For example, if two clients connect through a peer-to-peer connection and are geographically close to each other, the connection could be much faster compared to if the same clients connected through a central VPN server geographically located far away from the clients. The question is whether these kinds of scenarios happen frequently in practice. In most cases, clients won't have the ability to connect to each other with fewer internet hops, thus not benefiting from the peer-to-peer model as they continue to use the regular internet route through their internet service provider's network.

[10]A hop is network counting term for when a packet is passed from one network segment to the next segment.

So, there are scenarios where clients could benefit from a peer-to-peer connection compared to a regular central server model, but it is challenging to measure.

7 Discussion

7.1 Performance and availability

During this research and development of the Proof of Concept (PoC), some problems and possible improvements have arisen. The first one is the performance of the relay server, as it is a performance-critical component of the VPN system. The current prototype in Python only manages to relay traffic with very poor throughput of <100 megabits per second, as measured with `iperf3` with default settings. To be suitable for use in a production environment, the relay server must be implemented in a programming language with high networking performance like C, Go or Rust. Go would make the most sense, as the current eduVPN management daemon is also written in Go.

Another downside of the current PoC is that in the case of a relay packet, it simply forwards the packet to all connected peers. When a client sends a packet through the relay server to another client, it gets broadcasted by the relay server to all currently connected peers. Apart from the confidentiality downsides, it is also very inefficient for large networks.

The PoC still relies on one server that functions as a central management server and relay (TURN) server, making it a single point of failure. Most of these discussing points are taken into consideration for future works9 or are simply considered out of scope for the PoC as it is not production-proof yet.

7.2 Security

The second point of discussion is the lack of security in the PoC. Although various techniques are built into the PoC, it is not considered secure. There is no client authentication; if the management/ relay server is active, anyone who obtains the client source code could form a peer if they know the server endpoint address. Another security risk of the current PoC is IP spoofing. The relay server checks the source address of a data packet before relaying it to the designated peers, but it isn't protected against IP spoofing. If a relay packet with a spoofed IP address is sent to the relay server, it gets forwarded without any additional checking mechanisms. This poses a security risk as the relay server could be used to send malicious traffic to connected peers.

7.3 Roaming

As discussed in section 6.8, the PoC decides when a connection is setup whether a peer-to-peer connection can be established. However, once a connection is working, it does not monitor the connection status. Usually, changing networks means the TCP management channel socket breaks. When this socket is closed, the server automatically removes the corresponding peer and broadcasts a new peer list to other peers in the network. When a VPN client starts a new management channel connection, its address information is sent to all other peers again and a new network connection is started.

However, in the case where for whatever reason WireGuard's UDP data stream breaks, but the TCP management socket stays alive, the VPN connection breaks. This can happen for example if a device loses connectivity for a few minutes. Firewalls and NAT devices usually remember idle sessions longer in the case of TCP than for UDP. TCP session timeout is required to be at least 2 hours and 4 minutes [12, §5], while for UDP the requirement is no less than 2 minutes [9, §4.3]).

8 Conclusion

To enable eduVPN to offer peer-to-peer VPN functionality, various network scenarios in which potential eduVPN users might want to establish peer-to-peer connections were investigated. A technique for enabling a peer-to-peer connection must be determined for each specific scenario. In situations where clients have access to a public IPv4 or IPv6 address, a direct connection can be established. A firewall on incoming traffic, if present, can be bypassed using UDP hole punching. However, for more challenging and simultaneously, more common scenarios involving NAT devices, various techniques are being researched to traverse these NAT devices.

One such technique is to use hole punching to open network ports. Hole punching utilizes a STUN server to create table entries in network NAT devices. The functionality required by the STUN protocol to create NAT table entries is incorporated into a dedicated central server. This central server not only functions as a STUN server but also serves as a relay(TURN) server and management server. The relay server becomes essential when a direct peer-to-peer connection is not feasible, which occurs only when both clients are behind a Symmetric NAT device. In any other combination of various EIM NAT types, or even Symmetric NAT on one client, a direct peer-to-peer connection is possible.

The new solution incorporates a mechanism to first assess the feasibility of a peer-to-peer connection, resorting to the fallback relay server if necessary. This relay server forwards data packets using RAW sockets from one client to the other. The orchestration of both the peer-to-peer connection and the relay connection is managed through a dedicated management connection. All eduVPN clients connect to the management server through a custom handshake protocol, which is employed to dynamically add, remove, or update peers for the client.

Although UPnP is a technique for opening network ports, it is deemed impractical due to bandwidth limitations on UPnP connections in most consumer routers. For the last network scenario where UDP is completely blocked, the relay server could be utilized in combination with `proxyguard`. However, this option was not thoroughly researched as it was excluded from the project scope.

To conclude, WireGuard can be used to develop a peer-to-peer VPN solution for various practical scenarios, utilizing hole punching for direct peer-to-peer connections or a relay (TURN) server in cases where hole punching is not feasible. The developed solution could be integrated into the eduVPN architecture without modifying the WireGuard protocol itself, thus maintaining the integrity of the protocol.

9 Future Work

9.1 TCP Proxy

WireGuard VPN tunnels use UDP traffic. For networks that block UDP, eduVPN currently falls back to OpenVPN over TCP. However, they are developing a TCP proxy for WireGuard; `proxyguard`. Instead of connecting to a peer directly, WireGuard connects to `proxyguard` which encapsulates UDP datagrams as TCP segments and forwards it to a `proxyguard` instance on the other end.

Our mesh network can use `proxyguard` in a similar way. WireGuard would connect to `proxyguard`, which would connect to the relay server over TCP. The relay server would need to be modified to relay TCP traffic alongside UDP traffic.

9.2 Separation of relay server and management server

The current proof of concept has a single server program that runs a UDP relay and mock management server. This is because the relay server needs information from the management server; the network and peers that correspond to a UDP source address and port.

However, it could also obtain this information from the management server via an API over the network.

eduVPN already has a management server, in which the functions of our proof of concept management server could be integrated. The relay server should be built as a separate server component. This separation is important because it could allow the relay server to be programmed in a high performance programming language. Since the relay server is a simple independent program, many instances of the relay server can be hosted as a distributed system, only occasionally retrieving data from the main management server. The relay servers require lots of network throughput, the management servers don't.

9.3 Integration with eduVPN client

The proof of concept is built separately from the official eduVPN solution. The next step of the project will be to integrate it into an eduVPN release. eduVPN is currently running on version 3[11], and as the proof of concept is using completely new techniques, integrating it would likely result in a new major version. Before this can take place, the last remaining points discussed in section 7 discussion and section 9 future works must be resolved or, at least, taken into consideration.

[11]https://codeberg.org/eduVPN/vpn-user-portal/src/branch/v3/CHANGES.md

References

[1] Nick Aquina. "WireGuard in eduVPN". In: (Jan. 12, 2021). URL: https://www.tuxed.net/fkooman/files/eduVPN-WireGuard.pdf.

[2] Marek Küthe. *Comparison of self-meshing VPNs*. Dec. 13, 2023. URL: https://mk16.de/blog/comparison-of-self-meshing-vpns (visited on 01/08/2024).

[3] Avery Pennarun. "How Tailscale works". In: *Tailscale Blog* (Mar. 20, 2020). URL: https://tailscale.com/blog/how-tailscale-works (visited on 01/10/2024).

[4] David Anderson. *How NAT traversal works*. Aug. 21, 2020. URL: https://tailscale.com/blog/how-nat-traversal-works.

[5] *Documentation eduVPN*. URL: https://docs.eduvpn.org (visited on 12/06/2023).

[6] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnel*. June 1, 2020. URL: https://www.wireguard.com/papers/wireguard.pdf (visited on 01/10/2024).

[7] Jonathan Rosenberg et al. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. RFC 3489. Mar. 2003. DOI: 10.17487/RFC3489. URL: https://www.rfc-editor.org/info/rfc3489.

[8] Bryan Ford, Dan Kegel, and Pyda Srisuresh. *State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)*. RFC 5128. Mar. 2008. DOI: 10.17487/RFC5128. URL: https://www.rfc-editor.org/info/rfc5128.

[9] Cullen Fluffy Jennings and Francois Audet. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. RFC 4787. Jan. 2007. DOI: 10.17487/RFC4787. URL: https://www.rfc-editor.org/info/rfc4787.

[10] Patrick Sattler. "NAT Analyzer Results". In: (Nov. 30, 2011). URL: https://web.archive.org/web/20200213115759/http://nattest.net.in.tum.de/results.php (visited on 02/13/2020).

[11] Gertjan Halkes and Johan Pouwelse. "UDP NAT and Firewall Puncturing in the Wild". In: *NETWORKING 2011*. Ed. by Jordi Domingo-Pascual et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–12. ISBN: 978-3-642-20798-3.

[12] Bryan Ford et al. *NAT Behavioral Requirements for TCP*. RFC 5382. Oct. 2008. DOI: 10.17487/RFC5382. URL: https://www.rfc-editor.org/info/rfc5382.

Appendix

The code can also be found on GitHub[12]

A README.md

A.1 Peer to Peer WireGuard

Example of a VPN client and server that allows creating a peer to peer mesh network using WireGuard, without modifications to WireGuard itself. Uses UDP hole punching or a relay server. Fully supports IPv4 and IPv6 inside the tunnel, for peer to peer connections, for the relay server and the management server.

A.2 Requirements

Install `wireguard-tools` and a recent version of Python 3. No Python dependencies are required.

Additional requirements to use NetworkManager: `python3-gi` (Debian) or `python3-gobject` (Fedora).

B Server

Create the configuration file `server_config.json`:

```
{
    "server_port": 3000,
    "log_level": "INFO"
}
```

`log_level` can be changed to `DEBUG` for increased log output. The server runs a TCP and UDP server; ensure incoming traffic is allowed for both protocols.

B.1 Client

Create the configuration file `client_config.json`:

```
{
    "uuid": "b20b3973-6dcd-43be-a097-e80126ae6532",
    "address4": "10.200.0.1",
    "address6": "fdf0:a1e8:32b1:200::1",
    "server_host": "localhost",
    "server_port": 3000,
    "log_level": "INFO",
    "network_manager": true
}
```

- Every peer in a mesh network needs to be configured with the same UUID. A UUID can be generated using `uuidgen` or `python3 -m uuid`.
- Each peer should use a unique IPv4 and IPv6 address. The IPv4 address will be part of a /24 network, and the IPv6 address part of a /64 network. The IPv4 address should usually be in `10.0.0.0/8` or `172.16.0.0/12`, `192.168.0.0/16`. The IPv6 address should usually be chosen from the `fd00::/8` range, in the `fdss:ssss:ssss:nnnn::/64` format; `s` is a randomly chosen global ID, and `n` the network ID.

[12]https://github.com/Derkades/os3-wg-p2p/tree/main

- The client must be started with elevated privileges.
- `log_level` can be changed to `DEBUG` for increased log output.
- `network_manager` can be set to `true` to use NetworkManager via DBus or `false` to run `ip` / `wg` commands.

C client.py

```python
import json
import logging
import os
import select
import sys
import time
from ipaddress import IPv6Address
from pathlib import Path
from socket import SHUT_RDWR, SOCK_DGRAM, SOCK_STREAM, getaddrinfo, socket
from threading import Event, Thread
from typing import Optional
import random

import messages
from messages import MAGIC, AddressResponse, PeerHello, PeerList
from wg import WGManager, get_wireguard

log = logging.getLogger('client')


def mgmt_thread(mgmt_sock: socket, config, pubkey: str, addr4: tuple[str, int
    ], addr6: tuple[str, int], wg: WGManager):
    # Send hello to server via management channel
    hello = PeerHello(config['uuid'],
                      pubkey,
                      config['address4'],
                      config['address6'],
                      addr4,
                      addr6)
    log.debug('sending hello: %s', hello)
    mgmt_sock.send(messages.pack(hello))

    while True:
        data = mgmt_sock.recv(16384)
        if data == b'':
            break

        peer_list: PeerList = messages.unpack(data)
        log.debug('received %s peers from server', len(peer_list.peers))
        log.debug("peer list: %s", peer_list)
        wg.update_peers(peer_list.peers)

    log.debug('mgmt thread exits')


def get_addr_info(server_host: str, server_port: int, source_port: int) ->
    tuple[tuple[str, int], tuple[str, int]]:
    """
    Send UDP packet to server to discover external address and port.
    Also opens up NAT/firewall to receive UDP from relay server to WireGuard
    """
    addr4: Optional[tuple[str, int]] = None
    addr6: Optional[tuple[str, int]] = None

    for address in getaddrinfo(server_host, server_port, type=SOCK_DGRAM):
        s_family, s_type, _s_proto, _s_canonname, s_addr = address

        with socket(s_family, s_type) as sock:
            log.debug('connecting to %s with source port %s', s_addr, source_port)
            sock.bind(('', source_port))
            sock.connect(s_addr)
            sock.send(MAGIC)
            readable, _writable, _exceptional = select.select([sock], [], [], 2)
            if not readable:
                log.debug('no response from server')
                continue
            data = readable[0].recv(AddressResponse.SIZE)
            resp = AddressResponse.unpack(data)
            ipv4_mapped = IPv6Address(resp.host).ipv4_mapped
            if ipv4_mapped:
                addr4 = (str(ipv4_mapped), resp.port)
            else:
                addr6 = (resp.host, resp.port)

    return addr4, addr6


def main():
    config = json.loads(Path('client_config.json').read_text(encoding='utf-8'))
    logging.basicConfig()
    logging.getLogger().setLevel(config['log_level'])

    privkey = WGManager.gen_privkey()
    pubkey = WGManager.gen_pubkey(privkey)
    relay_endpoint = f"{config['server_host']}:{config['server_port']}"

    log.debug('wireguard public key: %s', pubkey)

    log.info('retrieving address information')

    listen_port = random.randint(2**15, 2**16)
    addr4, addr6 = get_addr_info(config['server_host'], config['server_port'], listen_port)

    if not addr4 and not addr6:
        log.error('could not discover external address')
        sys.exit(1)

    log.info('address info IPv4: %s', addr4)
    log.info('address info IPv6: %s', addr6)

    addresses = getaddrinfo(config['server_host'], config['server_port'], type=SOCK_STREAM)
    for s_family, s_type, _s_proto, _s_canonname, s_addr in addresses:
        # Establish connection for management channel
        mgmt_sock = socket(s_family, s_type)
        mgmt_sock.connect(s_addr)
        log.info('connected to management server: [%s]:%s', s_addr[0], s_addr[1])
        break
    else:
        log.error('cannot resolve server host: %s', config['server_host'])
        sys.exit(1)

    if_name = 'wg_p2p_' + os.urandom(2).hex()

    wg = get_wireguard(config['network_manager'], if_name, privkey, pubkey,
                       listen_port, addr4 is not None, addr6 is not None,
                       config['address4'], config['address6'],
                       relay_endpoint)

    def interface_up():
        Thread(target=mgmt_thread, args=(mgmt_sock, config, pubkey, addr4, addr6, wg)).start()

    log.info('creating WireGuard interface')
    wg.create_interface(interface_up)

    try:
        while True:
            time.sleep(1)
    except (KeyboardInterrupt, SystemExit):
        log.info('exiting')
        log.debug('close socket')
        mgmt_sock.shutdown(SHUT_RDWR)
        mgmt_sock.close()
        log.debug('remove interface')
        event = Event()
        wg.remove_interface(event.set)
        log.debug('waiting for remove_interface event')
        event.wait()


if __name__ == '__main__':
    main()
```

D server.py

```python
import json
import logging
import queue
import select
from socket import IPPROTO_IPV6, IPV6_V6ONLY, SHUT_RD, SO_REUSEADDR,
    SOCK_STREAM, SOL_IP, SOL_SOCKET, socket, SOCK_DGRAM, AF_INET, AF_INET6
import time
from dataclasses import dataclass
from multiprocessing.pool import ThreadPool
from pathlib import Path
from threading import Thread

import messages
from messages import MAGIC, AddressResponse, PeerHello, PeerInfo, PeerList

log = logging.getLogger('server')

@dataclass
class Peer:
    """Device (a WireGuard interface) in a network"""
    sock: socket
    addr4: tuple[str, int]
    addr6: tuple[str, int]
    pubkey: str
    vpn_addr4: str
    vpn_addr6: str


@dataclass
class Network:
    """Network of peers"""
    uuid: str
    peers: list[Peer]


NETWORK_BY_UUID: dict[str, Network] = {}
NETWORK_BY_ADDR: dict[tuple[str, int], Network] = {}  # for relay only
SOCKETS: set[socket] = set()
POOL = ThreadPool(32)


class Server:
    inputs: list[socket] = []
    outputs: list[socket] = []
    queues: dict[socket, queue.Queue] = {}
    sock_to_peer: dict[socket, tuple[Network, Peer]] = {}

    def send(self, sock: socket, data: bytes):
        self.queues[sock].put(data)
        self.outputs.append(sock)

    def close(self, sock: socket):
        log.debug('closing socket')
        self.inputs.remove(sock)
        if sock in self.outputs:
            self.outputs.remove(sock)
        sock.close()
        self.remove_peer(sock)

    def remove_peer(self, sock: socket):
        if sock in self.sock_to_peer:
            net, peer = self.sock_to_peer[sock]
            log.info('removing disconnected peer: %s', peer)
            net.peers.remove(peer)
            del self.sock_to_peer[sock]
            self.broadcast_peers(net.peers)
        else:
            log.debug('socket closed without disconnecting peer')

    def broadcast_peers(self, peers: list[Peer]):
        log.info('broadcast updated peer list to %s peers', len(peers))
        peer_list = PeerList([PeerInfo(peer.addr4, peer.addr6, peer.pubkey, peer.vpn_addr4, peer.vpn_addr6) for peer in peers])
        peer_list_bytes = messages.pack(peer_list)
        def send(peer: Peer):
            self.send(peer.sock, peer_list_bytes)
        POOL.map(send, peers)

    def handle_peer_hello(self, data, sock):
        hello: PeerHello = messages.unpack(data)
        if not hello:
            log.warning('ignoring invalid message from client')
            return

        log.debug('received hello: %s', hello)

        new_peer = Peer(sock, hello.addr4, hello.addr6, hello.pubkey, hello.vpn_addr4, hello.vpn_addr6)
        log.debug('new peer: %s', new_peer)

        if hello.uuid in NETWORK_BY_UUID:
            log.info('joining peer %s onto existing network %s', hello.pubkey, hello.uuid)
            net = NETWORK_BY_UUID[hello.uuid]
            net.peers.append(new_peer)
        else:
            log.info('registered new network %s for peer %s', hello.uuid, hello.pubkey)
            net = Network(hello.uuid, [new_peer])
            NETWORK_BY_UUID[hello.uuid] = net

        NETWORK_BY_ADDR[hello.addr4] = net
        NETWORK_BY_ADDR[hello.addr6] = net
        self.sock_to_peer[sock] = (net, new_peer)

        self.broadcast_peers(net.peers)

    def start(self, config):
        with socket(AF_INET6, SOCK_STREAM) as server:
            server.setblocking(0)
            server.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
            # Bind to IPv6 only
            # Dual stack mode accepts IPv4 connections using IPv4-mapped IPv6 address
            server.bind(('::', config['server_port']))
            server.listen()
            log.info('management server listening on [%s]:%s', server.getsockname()[0], server.getsockname()[1])
            SOCKETS.add(server)
            self.inputs.append(server)

            while self.inputs:
                readable, writable, exceptional = select.select(self.inputs, self.outputs, self.inputs)

                for sock in readable:
                    if sock is server:
                        client_sock, client_addr = sock.accept()
                        log.debug('new client connected from %s', client_addr)
                        self.inputs.append(client_sock)
                        self.queues[client_sock] = queue.Queue()
                        continue

                    data = sock.recv(16384)
                    if data:
                        self.handle_peer_hello(data, sock)
                    else:
                        self.close(sock)

                for sock in writable:
                    try:
                        next_msg = self.queues[sock].get_nowait()
                    except queue.Empty:
                        self.outputs.remove(sock)
                    else:
                        sock.send(next_msg)

                for sock in exceptional:
                    self.close(sock)

        log.debug('mgmt exit')


class Relay:
    def start(self, config):
        with socket(AF_INET6, SOCK_DGRAM) as sock:
            SOCKETS.add(sock)
            # Bind to IPv6 only
            # Dual stack mode accepts IPv4 connections using IPv4-mapped IPv6 address
            sock.bind(('::', config['server_port']))
            log.info('relay server listening on [%s]:%s', sock.getsockname()[0], sock.getsockname()[1])
            while True:
                data, addr = sock.recvfrom(1024)
                if data == b'':
                    log.debug('udp socket is dead')
                    break

                if data == MAGIC:
                    log.info('sending address response to %s', addr)
                    sock.sendto(AddressResponse(addr[0], addr[1]).pack(), addr)
                    continue

                if addr in NETWORK_BY_ADDR:
                    net = NETWORK_BY_ADDR[addr]
                    # Relay to all peers in the network. This is very inefficient for
                    # larger networks. A proper solution would be to run a separate
                    # UDP relay for every peer on a dedicated port. Then, outbound
                    # traffic from WireGuard would go to a different relay depending
                    # on the desired actual peer.
                    for peer in net.peers:
                        # at least don't relay back to peer
                        if addr != peer.addr4 and addr != peer.addr6:
                            log.debug('relay %s -> %s', addr, peer.wg_addr)
                            sock.sendto(data, peer.wg_addr)
                    continue

                log.warning('received unknown data from %s', addr)


def main():
    config = json.loads(Path('client_config.json').read_text(encoding='utf-8')
    )
    logging.basicConfig()
    logging.getLogger().setLevel(config['log_level'])

    relay = Relay()
    server = Server()

    Thread(target=relay.start, args=(config,)).start()
    Thread(target=server.start, args=(config,)).start()

    try:
        while True:
            time.sleep(1)
    except (KeyboardInterrupt, SystemExit):
        log.info('shutting down sockets')
        for sock in SOCKETS:
            try:
                sock.shutdown(SHUT_RD)
            except OSError:
                pass
            sock.close()


if __name__ == '__main__':
    main()
```

E messages.py

```
import gzip
import json
import struct
from abc import ABC
from dataclasses import asdict, dataclass
from gzip import BadGzipFile
from ipaddress import IPv6Address
from json import JSONDecodeError
from typing import Optional

MAGIC = b'awesome peer to peer magic to distinguish packet as different from
    wiregurad traffic'


@dataclass
class AddressResponse:
    SIZE = 18
    FORMAT = '!16sH'
    host: str # IPv6 address or IPv4-mapped IPv6 address (16 bytes)
    port: int # port number (2 bytes)

    def pack(self) -> bytes:
        return struct.pack(self.FORMAT, IPv6Address(self.host).packed, self.
    port)

    @classmethod
    def unpack(cls, packed: bytes) -> 'AddressResponse':
        host, port = struct.unpack(cls.FORMAT, packed)
        return cls(str(IPv6Address(host)), port)
```

```python


class Message(ABC):
    pass


@dataclass
class PeerHello(Message):
    uuid: str # uuid
    pubkey: str # wireguard pubkey
    vpn_addr4: str # IPv4 address inside the VPN
    vpn_addr6: str # IPv6 address inside the VPN
    addr4: Optional[tuple[str, int]] # for wireguard udp
    addr6: Optional[tuple[str, int]] # for wireguard udp


@dataclass
class PeerInfo:
    addr4: Optional[tuple[str, int]] # for wireguard udp
    addr6: Optional[tuple[str, int]] # for wireguard udp
    pubkey: str # wireguard public key
    vpn_addr4: str # IPv4 address inside the VPN
    vpn_addr6: str # IPv6 address inside the VPN


@dataclass
class PeerList(Message):
    peers: list[PeerInfo]


# quick and dirty message packing: gzipped json


def pack(msg: Message):
    return gzip.compress(json.dumps({'type': type(msg).__name__, **asdict(msg)
    }).encode())


def unpack(data) -> Optional[Message]:
    try:
        obj = json.loads(gzip.decompress(data).decode())
    except (BadGzipFile, JSONDecodeError):
        return None
    type = obj['type']
    del obj['type']
    if type == 'PeerHello':
        # rewrite list to tuple
        for name in ['addr4', 'addr6']:
            if obj[name] is not None:
                obj[name] = tuple(obj[name])
        return PeerHello(**obj)
    elif type == 'PeerList':
        return PeerList([PeerInfo(**peer) for peer in obj['peers']])
    raise ValueError(type)
```

F wg.py

```python
import logging
import os
import socket
import subprocess
import tempfile
import time
import uuid
from abc import ABC, abstractmethod
from dataclasses import dataclass
from pathlib import Path
from threading import Thread
from typing import Optional
```

```python

import udp
from messages import PeerInfo

log = logging.getLogger(__name__)

def create_tempfile(content: bytes, suffix: Optional[str] = None) -> str:
    fd, temp_path = tempfile.mkstemp(suffix=suffix)
    with os.fdopen(fd, 'wb') as temp_file:
        temp_file.write(content)
    return temp_path


def run(command: list[str],
        check: bool = True,
        stdin: Optional[bytes] = None,
        capture_output: bool = False) -> Optional[bytes]:
    log.debug('running command: %s', ' '.join(command))
    result = subprocess.run(command, check=check, capture_output=
    capture_output, input=stdin)
    return result.stdout.decode() if capture_output else None

@dataclass
class WGManager(ABC):
    if_name: str
    privkey: str
    pubkey: str
    listen_port: int
    ipv6: bool
    addr4: str
    addr6: str
    relay_endpoint: str
    mtu: int = 1380

    @abstractmethod
    def create_interface(self, callback) -> None:
        pass

    @abstractmethod
    def remove_interface(self, callback) -> None:
        pass

    @abstractmethod
    def list_peers(self) -> list[str]:
        pass

    @abstractmethod
    def add_peer(self, pubkey: str, endpoint: str, keepalive: int, allowed_ips
    : list[str]) -> None:
        pass

    @abstractmethod
    def remove_peer(self, pubkey: str) -> None:
        pass

    @abstractmethod
    def update_peer(self, pubkey: str, endpoint: str, keepalive: int) -> None:
        pass

    @abstractmethod
    def peer_rx(self, pubkey: str) -> int:
        pass

    def update_peers(self, peers: list[PeerInfo]):
        current_pubkeys = self.list_peers()
        log.debug('current peers: %s', current_pubkeys)

        for peer in peers:
            if peer.pubkey == self.pubkey or peer.pubkey in current_pubkeys:
                continue

            # If multiple peers are added, they need to be added at the same time, because
            # UDP hole punching and relay fallback are time-sensitive.
            Thread(target=self.set_up_peer_connection, args=(peer,)).start()

        # Remove local peers that are no longer known by the server
        active_pubkeys = {peer.pubkey for peer in peers}
        for pubkey in current_pubkeys:
            if pubkey not in active_pubkeys:
                log.info('removing peer: %s', pubkey)
                self.remove_peer(pubkey)

    def _find_source_ip(self, dest_ip: str) -> str:
        # There must be a better way...
        output = run(['ip', 'route', 'get', dest_ip], capture_output=True)
        dev = False
        for part in output.split():
            if dev:
                iface = part.strip()
                break
            dev = part.strip() == 'dev'

        output = run(['ip', 'addr', 'show', iface], capture_output=True)
        inet = False
        for part in output.split():
            if inet:
                return part.strip().split('/')[0]
            inet = part.strip() == 'inet'

    def set_up_peer_connection(self, peer: PeerInfo):
        peer_addr = None
        if self.ipv6 and peer.addr6:
            # can only connect to IPv6-only host directly
            peer_addr = peer.addr6
        elif not self.ipv6 and peer.addr4:
            # can connect to IPv4-only or dual stack host
            peer_addr = peer.addr4

        allowed_ips = [f'{peer.vpn_addr4}/32', f'{peer.vpn_addr6}/128']

        if not peer_addr:
            log.info('peer %s uses different address family, must use relay', peer.pubkey)
            self.add_peer(peer.pubkey, self.relay_endpoint, 25, allowed_ips)
            return

        log.info('trying p2p connection to peer: %s %s', peer.pubkey, peer_addr)

        # UDP hole punching
        try:
            source_ip = self._find_source_ip(peer_addr[0])
            source = (source_ip, self.listen_port)
            udp.send(b'', source, peer_addr)
        except PermissionError:
            log.warning('no permission to send raw udp for hole punching, are you root?')
        # Add peer with low keepalive
        endpoint = f'{peer_addr[0]}:{peer_addr[1]}'

        self.add_peer(peer.pubkey, endpoint, 1, allowed_ips)

        # Monitor RX bytes. The other end has also set persistent-keepalive=1, so we should see our
        # received bytes increase with 32 bytes every second
        rx_bytes = self.peer_rx(peer.pubkey)
        time.sleep(7)
        new_rx_bytes = self.peer_rx(peer.pubkey)
        log.debug('rx from %s to %s', rx_bytes, new_rx_bytes)
```

```python
        if new_rx_bytes > rx_bytes:
            log.info('p2p connection appears to be working')
            # Keepalive can now be increased to 25 seconds
            self.update_peer(peer.pubkey, endpoint, 25)
            return

        # Even if the two ends of a peer to peer connection decide differently on whether the
        # connection is working, they will still end up both using the same method, because
        # WireGuard updates its endpoint when it receives data from a different source address.

        log.info('p2p connection to %s failed, falling back to relay server', peer.pubkey)
        # Set endpoint to relay server, also increase keepalive
        self.update_peer(peer.pubkey, self.relay_endpoint, 25)

    @staticmethod
    def gen_privkey():
        return run(['wg', 'genkey'], capture_output=True)[:-1]

    @staticmethod
    def gen_pubkey(privkey: str):
        return run(['wg', 'pubkey'], stdin=privkey.encode(), capture_output=True)[:-1]

# Documentation: https://github.com/Derkades/os3-wg-p2p/issues/4#issuecomment-1909762430
class NMWGManager(WGManager):
    nm_uuid: Optional[str] = None
    nm: Optional['NM.Client'] = None
    glib_loop: Optional['GLib.MainLoop'] = None

    def _get_connection(self):
        return self.nm.get_connection_by_uuid(self.nm_uuid)

    def _get_wireguard_setting(self):
        con = self._get_connection()
        for setting in con.get_settings():
            if isinstance(setting, NM.SettingWireGuard):
                return setting
        return None

    def _get_device(self):
        return self.nm.get_device_by_iface(self.if_name)

    def create_interface(self, callback):
        self.glib_loop = GLib.MainLoop()
        Thread(target=self.glib_loop.run).start()

        self.nm_uuid = str(uuid.uuid4())

        GLib.idle_add(lambda: self._create_interface(callback))

    def _create_interface(self, callback):
        s_con = NM.SettingConnection.new()
        s_con.set_property(NM.SETTING_CONNECTION_TYPE, 'wireguard')
        s_con.set_property(NM.SETTING_CONNECTION_INTERFACE_NAME, self.if_name)
        s_con.set_property(NM.SETTING_CONNECTION_ID, self.if_name)
        s_con.set_property(NM.SETTING_CONNECTION_UUID, self.nm_uuid)

        s_ip4 = NM.SettingIP4Config.new()
        s_ip4.set_property(NM.SETTING_IP_CONFIG_METHOD, 'manual')
        s_ip4.add_address(NM.IPAddress(socket.AF_INET, self.addr4, 24))

        s_wg = NM.SettingWireGuard.new()
        s_wg.set_property(NM.SETTING_WIREGUARD_LISTEN_PORT, self.listen_port)
        s_wg.set_property(NM.SETTING_WIREGUARD_PRIVATE_KEY, self.privkey)
        s_wg.set_property(NM.SETTING_WIREGUARD_MTU, self.mtu)

        profile = NM.SimpleConnection.new()
        for s in (s_con, s_ip4, s_wg):
            profile.add_setting(s)

        self.nm = NM.Client.new(None)

        def add_callback(nm2, result):
            log.debug('add_callback')
            nm2.add_connection_finish(result)
            callback()

        log.debug('add_async')
        self.nm.add_connection_async(connection=profile, save_to_disk=False,
    callback=add_callback)

    def remove_interface(self, callback):
        def delete_callback(a, res):
            log.debug('delete_finish')
            a.delete_finish(res)
            self.glib_loop.quit()
            callback()

        def delete():
            log.debug('delete connection')
            con = self._get_connection()
            con.delete_async(callback=delete_callback)

        def disconnect_callback(a, res):
            log.debug('disconnect_finish')
            a.disconnect_finish(res)
            delete()

        def disconnect():
            device = self._get_device()
            if device:
                log.debug('disconnect_async')
                device.disconnect_async(callback=disconnect_callback)
            else:
                log.warning('device does not exist, already disconnected?')
                delete()

        GLib.idle_add(disconnect)

    def list_peers(self):
        s_wg = self._get_wireguard_setting()
        return [s_wg.get_peer(i).get_public_key() for i in range(s_wg.
    get_peers_len())]

    def update(self):
        def reapply_callback(a, res):
            log.debug('reapply_callback')
            a.reapply_finish(res)
            log.debug('peers after apply: %s', self.list_peers())

        def reapply():
            log.debug('reapply_async')
            log.debug('peers before apply: %s', self.list_peers())
            con = self._get_connection()
            self._get_device().reapply_async(con, 0, 0, callback=
    reapply_callback)

        def commit_callback(a, res):
            log.debug('commit_callback')
            a.commit_changes_finish(res)
            reapply()

        def commit():
            log.debug('peers before commit_changes: %s', self.list_peers())
            log.debug('commit_changes_async')
```

```python
            con = self._get_connection()
            con.commit_changes_async(save_to_disk=False, callback=commit_callback)

        GLib.idle_add(commit)

    def add_peer(self, pubkey: str, endpoint: str, keepalive: int, allowed_ips: list[str]) -> None:
        peer = NM.WireGuardPeer.new()
        peer.set_endpoint(endpoint, allow_invalid=False)
        peer.set_public_key(pubkey, accept_invalid=False)
        peer.set_persistent_keepalive(keepalive)
        for ip in allowed_ips:
            peer.append_allowed_ip(ip.strip(), accept_invalid=False)

        s_wg = self._get_wireguard_setting()
        s_wg.append_peer(peer)
        self.update()

    def remove_peer(self, pubkey: str) -> None:
        s_wg = self._get_wireguard_setting()
        pp_peer, pp_idx = s_wg.get_peer_by_public_key(pubkey)
        if pp_peer:
            s_wg.remove_peer(pp_idx)
            self.update()
        else:
            log.warning('peer %s does not exist', pubkey)
            log.debug('peers: %s', self.list_peers())

    def update_peer(self, pubkey: str, endpoint: str, keepalive: int) -> None:
        s_wg = self._get_wireguard_setting()
        pp_peer, pp_idx = s_wg.get_peer_by_public_key(pubkey)
        if pp_peer:
            peer = pp_peer.new_clone(True)
            peer.set_endpoint(endpoint, allow_invalid=False)
            peer.set_persistent_keepalive(keepalive)
            s_wg.set_peer(peer, pp_idx)
            self.update()
        else:
            log.warning('peer %s does not exist', pubkey)
            log.debug('peers: %s', self.list_peers())

    def peer_rx(self, pubkey: str) -> int:
        log.warning('cannot determine per-peer rx bytes using network manager')
        log.warning('returning interface rx bytes, unreliable when multiple peers are active')
        try:
            path = Path('/sys/class/net') / self.if_name / 'statistics' / 'rx_bytes'
            return int(path.read_text(encoding='utf-8'))
        except FileNotFoundError:
            log.warning('cannot read rx_bytes')
            return 0


class WGToolsWGManager(WGManager):
    def create_interface(self, callback):
        privkey_path = create_tempfile(self.privkey.encode())
        run(['ip', 'link', 'add', self.if_name, 'type', 'wireguard'])
        run(['wg', 'set', self.if_name, 'private-key', privkey_path, 'listen-port', str(self.listen_port)])
        run(['ip', 'address', 'add', self.addr4 + '/24', 'dev', self.if_name])
        run(['ip', 'address', 'add', self.addr6 + '/64', 'dev', self.if_name])
        run(['ip', 'link', 'set', 'mtu', str(self.mtu), 'up', 'dev', self.if_name])
        os.unlink(privkey_path)
        callback()

    def remove_interface(self, callback):
        run(['ip', 'link', 'del', self.if_name])
        callback()

    def list_peers(self) -> list[str]:
        return run(['wg', 'show', self.if_name, 'peers'], capture_output=True)
    .splitlines()

    def add_peer(self, pubkey: str, endpoint: str, keepalive: int, allowed_ips
    : list[str]) -> None:
        run(['wg', 'set', self.if_name,
             'peer', pubkey,
             'endpoint', endpoint,
             'persistent-keepalive', str(keepalive),
             'allowed-ips', ', '.join(allowed_ips)])

    def remove_peer(self, pubkey: str) -> None:
        run(['wg', 'set', self.if_name, 'peer', pubkey, 'remove'])

    def update_peer(self, pubkey: str, endpoint: str, keepalive: int) -> None:
        run(['wg', 'set', self.if_name, 'peer', pubkey, 'endpoint', endpoint,
    'persistent-keepalive', str(keepalive)])

    def peer_rx(self, pubkey: str) -> int:
        output = run(['wg', 'show', self.if_name, 'transfer'], capture_output=
    True)
        for line in output.splitlines():
            cols = line.split()
            if cols[0].strip() == pubkey:
                return int(cols[1].strip())
        log.warning('could not determine rx bytes for %s', pubkey)
        return 0


def get_wireguard(use_nm, *args) -> WGManager:
    if use_nm:
        import gi
        gi.require_version("NM", "1.0")
        from gi.repository import NM as NM2
        from gi.repository import GLib as GLib2
        global NM, GLib
        NM = NM2
        GLib = GLib2
        return NMWGManager(*args)
    else:
        return WGToolsWGManager(*args)
```

G udp.py

```python
import socket
import struct
from ipaddress import IPv4Address
import logging

log = logging.getLogger('udp')

# TODO: IPv6
# TODO: Run as helper program so main program doesn't need root access

# UDP header (from RFC 768):
#   0      7 8     15 16    23 24    31
#   +--------+--------+--------+--------+
#   |     Source      |   Destination   |
#   |      Port       |      Port       |
#   +--------+--------+--------+--------+
#   |                 |                 |
#   |     Length      |    Checksum     |
#   +--------+--------+--------+--------+
#   |
#   |          data octets ...
#   +---------------- ...
#
# Pseudo header for checksum:
#    0       7 8     15 16    23 24    31
#   +--------+--------+--------+--------+
#   |          source address           |
#   +--------+--------+--------+--------+
#   |        destination address        |
#   +--------+--------+--------+--------+
#   |  zero  |protocol|   UDP length    |
#   +--------+--------+--------+--------+

def send(data, source_addr: tuple[str, int], dest_addr: tuple[str, int]):
    if ':' in dest_addr[0]:
        log.error('IPv6 is not supported, cannot send to %s', dest_addr)
        return

    log.debug('sending UDP from %s to %s', source_addr, dest_addr)

    data_len = len(data)
    udp_length = 8 + data_len
    checksum = 0
    pseudo_header = struct.pack('!4s4sBBH',
                                IPv4Address(source_addr[0]).packed, IPv4Address(dest_addr[0]).packed,
                                0, socket.IPPROTO_UDP, udp_length)
    udp_header = struct.pack('!HHHH', source_addr[1], dest_addr[1], udp_length, 0)
    checksum = _checksum_func(pseudo_header + udp_header + data)
    udp_header = struct.pack('!HHHH', source_addr[1], dest_addr[1], udp_length, checksum)
    with socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_UDP) as sock:
        sock.sendto(udp_header + data, dest_addr)


def _checksum_func(data):
    checksum = 0
    data_len = len(data)
    if (data_len % 2):
        data_len += 1
        data += struct.pack('!B', 0)

    for i in range(0, data_len, 2):
        w = (data[i] << 8) + (data[i + 1])
        checksum += w

    checksum = (checksum >> 16) + (checksum & 0xFFFF)
    checksum = ~checksum & 0xFFFF
    return checksum
```