Department of Information Engineering and Computer Science

Bachelor's Degree in
Information and Communications Engineering

WireGuard VPN Service

Advisor

Prof. Fabrizio Granelli

Thesis co-supervisor

Prof. to be defined

Company tutors

Mattia Zago

Candidate

Mattia Meneghin

Monokee s.r.l.

Academic Year 2022/2023

Contents

1 WireGuard Introduction

WireGuard is a fast and modern VPN that utilizes state-of-the-art cryptography. It aims to be faster, simpler, leaner, and more useful than IPsec, while avoiding the massive headache. It intends to be considerably more performant than OpenVPN, moreover is designed as a general purpose VPN. Initially released for the Linux kernel, it is now cross-platform (Windows, macOS, BSD, iOS, Android) and widely deployable. It is currently under heavy development, but already it might be regarded as the most secure, easiest to use, and simplest VPN solution in the industry.

1.0.1 Fast & High Performance

A combination of extremely high-speed cryptographic primitives and the fact that WireGuard lives inside the Linux kernel means that secure networking can be very high-speed. It is suitable for both small embedded devices like smartphones and fully loaded backbone routers.

1.0.2 Easy-To-Use

WireGuard aims to be as easy to configure and deploy as SSH. A VPN connection is made simply by exchanging very simple public keys – exactly like exchanging SSH keys – and all the rest is transparently handled by WireGuard. It is even capable of roaming between IP addresses, just like Mosh (Mobile Shell). WireGuard has been designed with ease-of-implementation and simplicity in mind. It is meant to be easily implemented in very few lines of code, and easily auditable for security vulnerabilities.

1.0.3 WireGuard Cryptography

WireGuard uses state-of-the-art cryptography, like the Noise protocol framework, Curve25519, ChaCha20, Poly1305, BLAKE2, SipHash24, HKDF, and secure trusted constructions. It makes conservative and reasonable choices and has been reviewed by cryptographers.

1.0.4 Security

WireGuard securely encapsulates IP packets over UDP. You add a WireGuard interface, configure it with your private key and your peers' public keys, and then you send packets across it.

Note All issues of key distribution and pushed configurations are out of scope of WireGuard; these are issues much better left for other layers. In contrast, it more mimics the model of SSH and Mosh; both parties have each other's public keys, and then they're simply able to begin exchanging packets through the interface.

1.1 Cryptography

WireGuard uses state-of-the-art cryptography, like the Noise protocol framework, Curve25519, ChaCha20, Poly1305, BLAKE2, SipHash24, HKDF, and secure trusted constructions. It makes conservative and reasonable choices and has been reviewed by cryptographers.

1.1.1 Noise Protocol Framework

Noise is a framework for building crypto protocols. Noise protocols support mutual and optional authentication, identity hiding, forward secrecy, zero round-trip encryption, and other advanced features.

1.1.2 Curve25519

Curve25519 is a state-of-the-art Diffie-Hellman function suitable for a wide variety of applications. Given a user's 32-byte secret key, Curve25519 computes the user's 32-byte public key. Given the user's 32-byte secret key and another user's 32-byte public key, Curve25519 computes a 32-byte secret shared by the two users. This secret can then be used to authenticate and encrypt messages between the two users.

1.1.3 Poly1305-AES

Poly1305-AES is a state-of-the-art secret-key message-authentication code suitable for a wide variety of applications. Poly1305-AES computes a 16-byte authenticator of a message of any length, using a 16-byte nonce (unique message number) and a 32-byte secret key. Attackers can't modify or forge messages if the message sender transmits an authenticator along with each message and the message receiver checks each authenticator.

1.1.4 BLAKE2

BLAKE2 is a cryptographic hash function faster than MD5, SHA-1, SHA-2, and SHA-3, yet is at least as secure as the latest standard SHA-3. BLAKE2 has been adopted due to its high speed, security, and simplicity. BLAKE2 comes in two flavors:

- BLAKE2b (or just BLAKE2) is optimized for 64-bit platforms (e.g. NEON-enabled ARMs) and produces digests of any size between 1 and 64 bytes
- BLAKE2s is optimized for 8- to 32-bit platforms and produces digests of any size between 1 and 32 bytes

1.2 WireGuard Network Interface

WireGuard works by adding a network interface (or multiple), like eth0 or wlan0, called wg0 (or wg1, wg2, wg3, etc) that acts as a tunnel interface.
WireGuard associates tunnel IP addresses with public keys and remote endpoints.

When the interface sends a packet to a peer, it does the following:

- This packet is meant for 192.168.30.8.
- Get which peer is that.
- Looks for peer ABCDEFGH. (Or if it's not for any configured peer, drop the packet.)
- Encrypt entire IP packet using peer ABCDEFGH's public key.
- The endpoint of peer ABCDEFGH is UDP port 53133 on host 216.58.211.110.
- Send encrypted bytes from step 2 over the Internet to 216.58.211.110:53133 using UDP.

When the interface receives a packet, this happens:

- The packet was from UDP port 7361 on host 98.139.183.24.
- It decrypted and authenticated properly for peer LMNOPQRS.
- Keeps track that peer LMNOPQRS's most recent Internet endpoint is 98.139.183.24:7361 using UDP.
- Once decrypted, the plain-text packet is from 192.168.43.89.
- Check if the peer LMNOPQRS is allowed to be send packets as 192.168.43.89.
- If so, accept the packet on the interface. If not, drop it.

Behind the scenes there is much happening to provide proper privacy, authenticity, and perfect forward secrecy, using state-of-the-art cryptography.

At the heart of WireGuard is a concept called Cryptokey Routing, which works by associating public keys with a list of tunnel IP addresses that are allowed inside the tunnel. Each network interface has a private key and a list of peers. Each peer has a public key. Public keys are short and simple, and are used by peers to authenticate each other. They can be passed around for use in configuration files by any out-of-band method, similar to how one might send their SSH public key to a friend for access to a shell server.

1.3 Cryptokey Routing

At the heart of WireGuard is a concept called Cryptokey Routing, which works by associating public keys with a list of tunnel IP addresses that are allowed inside the tunnel. Each network interface has a private key and a list of peers. Each peer has a public key. Public keys are short and simple, and are used by peers to authenticate each other. They can be passed around for use in configuration files by any out-of-band method, similar to how one might send their SSH public key to a friend for access to a shell server. The Wireguard protocol doesn't differentiate between client and server, it is a peer-to-peer protocol, so we need to authenticate both.

1.3.1 Server Configuration

In the server configuration, each peer (a client) will be able to send packets to the network interface with a source IP matching his corresponding list of allowed IPs. For example, when a packet is received by the server from peer gN65BkIK..., after being decrypted and authenticated, if its source IP is 10.13.13.2 (or 10.13.13.3), then it's allowed onto the interface; otherwise it's dropped.

1.3.2 Server Configuration

In the server configuration, when the network interface wants to send a packet to a peer (a client), it looks at that packet's destination IP and compares it to each peer's list of allowed IPs to see which peer to send it to. For example, if the network interface is asked to send a packet with a destination IP of 10.13.13.2, it will encrypt it using the public key of peer gN65BkIK..., and then send it to that peer's most recent Internet endpoint. Client Configuration

1.3.3 Client Configuration

In the client configuration, its single peer (the server) will be able to send packets to the network interface using the IP defined in the client wg0.conf file. For example, when a packet is received from peer HIgo9xNz..., if it decrypts and authenticates correctly and if the IP matches with the one used in the configuration, then it's allowed onto the interface; otherwise it's dropped.
If you need to send packets with any source IP use AllowedIPs = 0.0.0.0/0 as wildcard
In the client configuration, when the network interface wants to send a packet to its single peer (the server), it will encrypt packets for the single peer using the IP defined in the client wg0.conf file. For example, if the network interface is asked to send a packet using the defined IP, it will encrypt it using the public key of the single peer HIgo9xNz..., and then send it to the single peer's most recent Internet endpoint.

1.3.4 How it works

In other words, when sending packets, the list of allowed IPs behaves as a sort of routing table, and when receiving packets, the list of allowed IPs behaves as a sort of access control list.
This is what we call a Cryptokey Routing Table: the simple association of public keys and allowed IPs.
Any combination of IPv4 and IPv6 can be used, for any of the fields. WireGuard is fully capable of encapsulating one inside the other if necessary.

1.3.5 Easy to manage

Because all packets sent on the WireGuard interface are encrypted and authenticated, and because there is such a tight coupling between the identity of a peer and the allowed IP address of a peer, system administrators do not need complicated firewall extensions, such as in the case of IPsec, but rather they can simply match on "is it from this IP? on this interface?", and be assured that it is a secure and authentic packet. This greatly simplifies network management and access control, and provides a great deal more assurance that your iptables rules are actually doing what you intended for them to do.

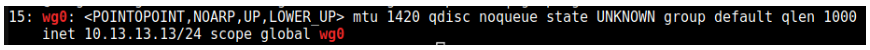
1.4 Built-in Roaming

The client configuration contains an initial endpoint of its single peer (the server), so that it knows where to send encrypted data before it has received encrypted data. The server configuration doesn't have any initial endpoints of its peers (the clients). This is because the server discovers the endpoint of its peers by examining from where correctly authenticated data originates. If the server itself changes its own endpoint, and sends data to the clients, the clients will discover the new server endpoint and update the configuration just the same. Both client and server send encrypted data to the most recent IP endpoint for which they authentically decrypted data. Thus, there is full IP roaming on both ends.

2 WireGuard Configuration

2.1 Configuration

Both Server and client are configured using wg0.conf file.
wg0 will be the name of the network interface:

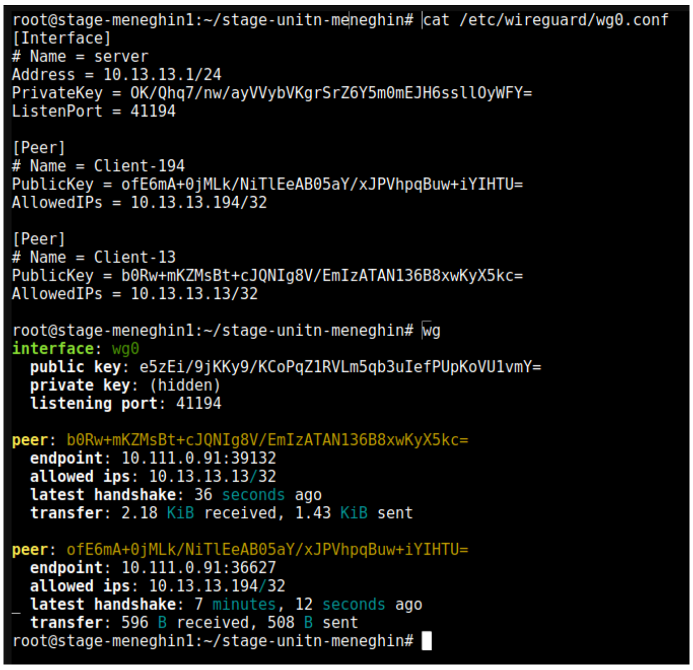


```
15: wg0: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1420 qdisc noqueue state UNKNOWN group default qlen 1000
    inet 10.13.13.13/24 scope global wg0
```

Figure 2.1: Netwrok Interface

2.1.1 Server Configuration File

Path: /etc/wireguard/wg0.conf



```
root@stage-meneghin1:~/stage-unitn-meneghin# cat /etc/wireguard/wg0.conf
[Interface]
# Name = server
Address = 10.13.13.1/24
PrivateKey = OK/Qhq7/nw/ayVVybVKgrSrZ6Y5m0mEJH6ssllOyWFY=
ListenPort = 41194

[Peer]
# Name = Client-194
PublicKey = ofE6mA+0jMLk/NiTlEeAB05aY/xJPVhpqBuw+iYIHTU=
AllowedIPs = 10.13.13.194/32

[Peer]
# Name = Client-13
PublicKey = b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=
AllowedIPs = 10.13.13.13/32

root@stage-meneghin1:~/stage-unitn-meneghin# wg
interface: wg0
  public key: e5zEi/9jKKy9/KCoPqZ1RVLm5qb3uIefPUpKoVU1vmY=
  private key: (hidden)
  listening port: 41194

peer: b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=
  endpoint: 10.111.0.91:39132
  allowed ips: 10.13.13.13/32
  latest handshake: 36 seconds ago
  transfer: 2.18 KiB received, 1.43 KiB sent

peer: ofE6mA+0jMLk/NiTlEeAB05aY/xJPVhpqBuw+iYIHTU=
  endpoint: 10.111.0.91:36627
  allowed ips: 10.13.13.194/32
  latest handshake: 7 minutes, 12 seconds ago
  transfer: 596 B received, 508 B sent
root@stage-meneghin1:~/stage-unitn-meneghin#
```

Figure 2.2: wg0.conf Server File

2.2 Wireguard tools for Nodejs

WireGuard offers a library which includes a class and set of helper functions for working with WireGuard config files in javascript/typescript. Link for the library

2.3 GitHub Repository

Here is the repository

3 Idea of usage

3.1 Handshake of new client configuration

1. client creates a keys pair and sends to the server only the **public key**
2. server checks if the public key received by the client is already existing among the peers
   - if the public key received is already used, then aborts the adding
   - if the public key received is unused, then the server will look for a free IP in the network to assign to the new client request
3. once the server has the public key and the free IP self-established, it adds to its config file the peer

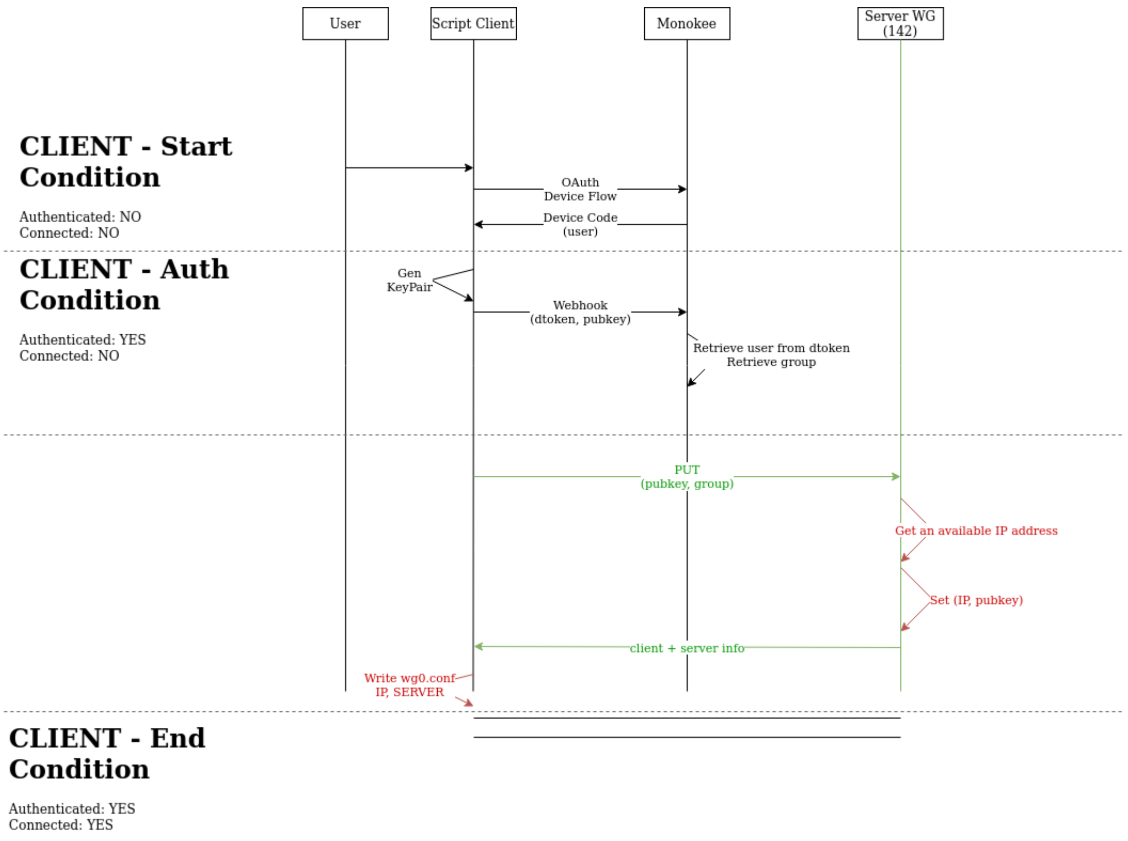
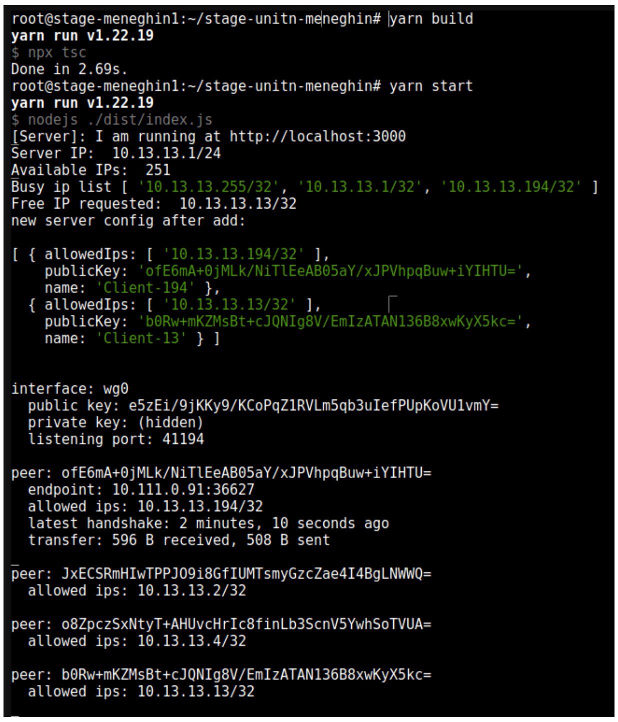
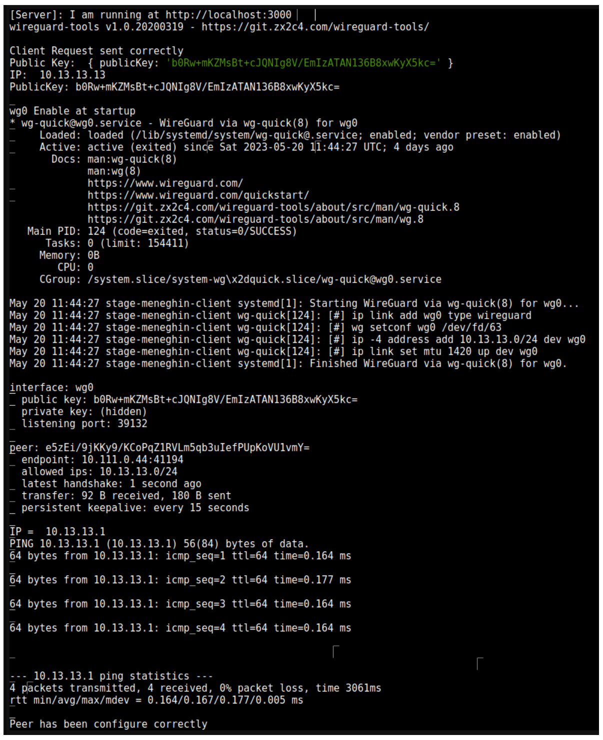


Figure 3.1: Hand Shake Idea



```
root@stage-meneghin1:~/stage-unitn-meneghin# yarn build
yarn run v1.22.19
$ npx tsc
Done in 2.69s.
root@stage-meneghin1:~/stage-unitn-meneghin# yarn start
yarn run v1.22.19
$ nodejs ./dist/index.js
[Server]: I am running at http://localhost:3000
Server IP:  10.13.13.1/24
Available IPs:  251
Busy ip list [ '10.13.13.255/32', '10.13.13.1/32', '10.13.13.194/32' ]
Free IP requested:  10.13.13.13/32
new server config after add:

[ { allowedIps: [ '10.13.13.194/32' ],
    publicKey: 'ofE6mA+0jMLk/NiTlEeAB05aY/xJPVhpqBuw+iYIHTU=',
    name: 'Client-194' },
  { allowedIps: [ '10.13.13.13/32' ],
    publicKey: 'b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=',
    name: 'Client-13' } ]


interface: wg0
  public key: e5zEi/9jKKy9/KCoPqZ1RVLm5qb3uIefPUpKoVU1vmY=
  private key: (hidden)
  listening port: 41194

peer: ofE6mA+0jMLk/NiTlEeAB05aY/xJPVhpqBuw+iYIHTU=
  endpoint: 10.111.0.91:36627
  allowed ips: 10.13.13.194/32
  latest handshake: 2 minutes, 10 seconds ago
  transfer: 596 B received, 508 B sent

peer: JxECSRmHIwTPPJO9i8GfIUMTsmyGzcZae4I4BgLNWWQ=
  allowed ips: 10.13.13.2/32

peer: o8ZpczSxNtyT+AHUvcHrIc8finLb3ScnV5YwhSoTVUA=
  allowed ips: 10.13.13.4/32

peer: b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=
  allowed ips: 10.13.13.13/32
```

Figure 3.2: Server side - Peer Configured



```
[Server]: I am running at http://localhost:3000
wireguard-tools v1.0.20200319 - https://git.zx2c4.com/wireguard-tools/

Client Request sent correctly
Public Key:  { publicKey: 'b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=' }
IP:  10.13.13.13
PublicKey: b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=

wg0 Enable at startup
* wg-quick@wg0.service - WireGuard via wg-quick(8) for wg0
     Loaded: loaded (/lib/systemd/system/wg-quick@.service; enabled; vendor preset: enabled)
     Active: active (exited) since Sat 2023-05-20 11:44:27 UTC; 4 days ago
       Docs: man:wg-quick(8)
             man:wg(8)
             https://www.wireguard.com/
             https://www.wireguard.com/quickstart/
             https://git.zx2c4.com/wireguard-tools/about/src/man/wg-quick.8
             https://git.zx2c4.com/wireguard-tools/about/src/man/wg.8
   Main PID: 124 (code=exited, status=0/SUCCESS)
      Tasks: 0 (limit: 154411)
     Memory: 0B
        CPU: 0
     CGroup: /system.slice/system-wg\x2dquick.slice/wg-quick@wg0.service

May 20 11:44:27 stage-meneghin-client systemd[1]: Starting WireGuard via wg-quick(8) for wg0...
May 20 11:44:27 stage-meneghin-client wg-quick[124]: [#] ip link add wg0 type wireguard
May 20 11:44:27 stage-meneghin-client wg-quick[124]: [#] wg setconf wg0 /dev/fd/63
May 20 11:44:27 stage-meneghin-client wg-quick[124]: [#] ip -4 address add 10.13.13.0/24 dev wg0
May 20 11:44:27 stage-meneghin-client wg-quick[124]: [#] ip link set mtu 1420 up dev wg0
May 20 11:44:27 stage-meneghin-client systemd[1]: Finished WireGuard via wg-quick(8) for wg0.

interface: wg0
  public key: b0Rw+mKZMsBt+cJQNIg8V/EmIzATAN136B8xwKyX5kc=
  private key: (hidden)
  listening port: 39132

peer: e5zEi/9jKKy9/KCoPqZ1RVLm5qb3uIefPUpKoVU1vmY=
  endpoint: 10.111.0.44:41194
  allowed ips: 10.13.13.0/24
  latest handshake: 1 second ago
  transfer: 92 B received, 180 B sent
  persistent keepalive: every 15 seconds

IP =  10.13.13.1
PING 10.13.13.1 (10.13.13.1) 56(84) bytes of data.
64 bytes from 10.13.13.1: icmp_seq=1 ttl=64 time=0.164 ms

64 bytes from 10.13.13.1: icmp_seq=2 ttl=64 time=0.177 ms

64 bytes from 10.13.13.1: icmp_seq=3 ttl=64 time=0.164 ms

64 bytes from 10.13.13.1: icmp_seq=4 ttl=64 time=0.164 ms



--- 10.13.13.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3061ms
rtt min/avg/max/mdev = 0.164/0.167/0.177/0.005 ms

Peer has been configure correctly
```

Figure 3.3: Client side - Peer Configured

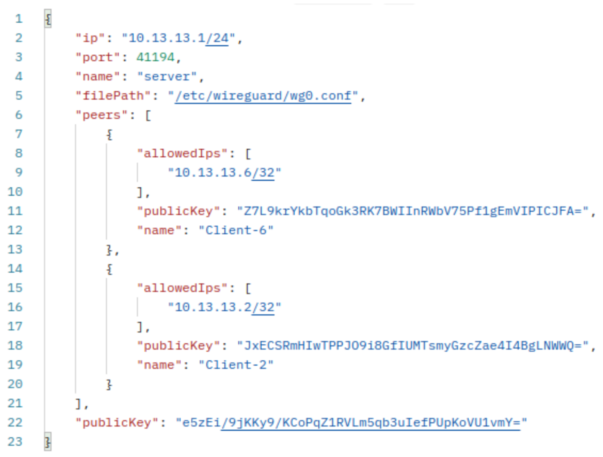
4 APIs

4.1 GET Server Info

GET - /info

Through a passed configuration file, we want to get all the properties of our VPN server, except the server private key.

INPUT - WgConfig server configuration file
RETURN - Object { ip, port, name, filePath, peers[], publicKey }



```json
{
    "ip": "10.13.13.1/24",
    "port": 41194,
    "name": "server",
    "filePath": "/etc/wireguard/wg0.conf",
    "peers": [
        {
            "allowedIps": [
                "10.13.13.6/32"
            ],
            "publicKey": "Z7L9krYkbTqoGk3RK7BWIInRWbV75Pf1gEmVIPICJFA=",
            "name": "Client-6"
        },
        {
            "allowedIps": [
                "10.13.13.2/32"
            ],
            "publicKey": "JxECSRmHIwTPPJO9i8GfIUMTsmyGzcZae4I4BgLNWWQ=",
            "name": "Client-2"
        }
    ],
    "publicKey": "e5zEi/9jKKy9/KCoPqZ1RVLm5qb3uIefPUpKoVU1vmY="
}
```

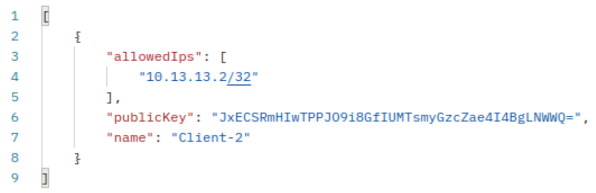
Figure 4.1: API - GET Server Info

4.2 GET Server Peers

GET - /peers

Through a passed configuration file we want to get all the server peers. (i.e. the clients of our VPN server).

INPUT - WgConfig server configuration file
RETURN - List with all the server peers



```json
[
    {
        "allowedIps": [
            "10.13.13.2/32"
        ],
        "publicKey": "JxECSRmHIwTPPJO9i8GfIUMTsmyGzcZae4I4BgLNWWQ=",
        "name": "Client-2"
    }
]
```

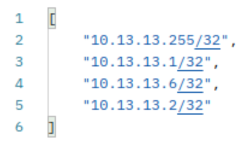
Figure 4.2: API - GET Server Info

4.3 GET All Busy IPs

GET - /all_ip

Through a passed configuration file we want to get all the busy IPs assigned to the clients of our VPN server.

INPUT - WgConfig server configuration file
RETURN - List with all the busy IPs



```
[
    "10.13.13.255/32",
    "10.13.13.1/32",
    "10.13.13.6/32",
    "10.13.13.2/32"
]
```

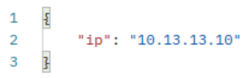
Figure 4.3: API - GET All busy IPs

4.4 GET Free IP

GET - /free_ip

Through a passed configuration file we want to get a free IP to be assigned to a new client of our VPN server.

INPUT - WgConfig server configuration file
RETURN - Object containing a single free IP in the server network



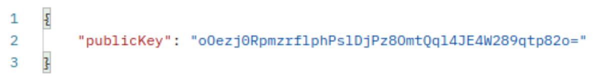
```
{
    "ip": "10.13.13.10"
}
```

Figure 4.4: API - GET Free IP

4.5 PUT client request

PUT - /request
First step of the client configuration hand-shake. The client will generate a keys pair and save it into a temporary file in */etc/wireguard/temp/*, then returns its public key.

INPUT - Null
RETURN - Object public Key



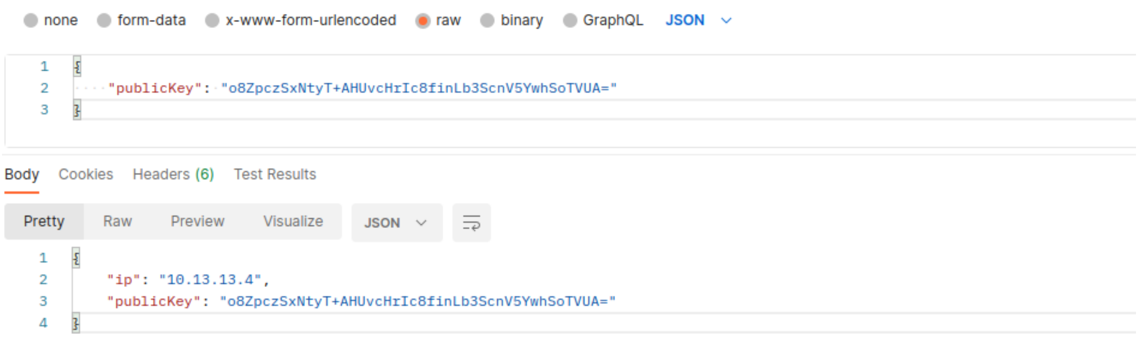
```
{
    "publicKey": "oOezj0RpmzrflphPslDjPz8OmtQql4JE4W289qtp82o="
}
```

Figure 4.5: API - PUT client request

4.6 PUT Peer On Server

PUT - /server/
Through a passed public key (Object), we want to add the client, assigning it a free IP available. It writes the rows in the server configuration file and adds the route.

INPUT - Object client public key
RETURN - Object IP and client public key



```
{
    "publicKey": "o8ZpczSxNtyT+AHUvcHrIc8finLb3ScnV5YwhSoTVUA="
}
```

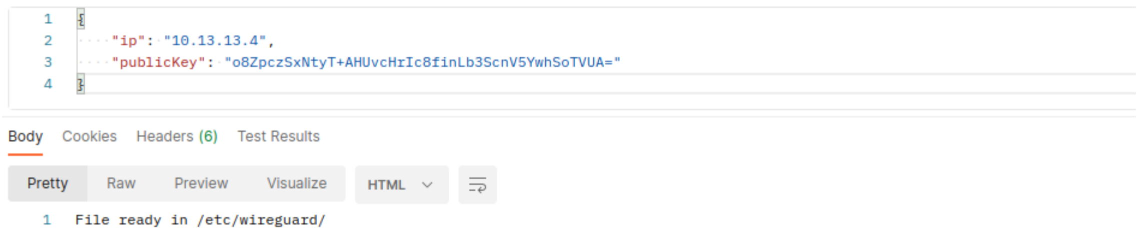
```
{
    "ip": "10.13.13.4",
    "publicKey": "o8ZpczSxNtyT+AHUvcHrIc8finLb3ScnV5YwhSoTVUA="
}
```

Figure 4.6: API - PUT Peer On Server

4.7 PUT client create

PUT - /create
With the given IP and key previously received, it generates the client config file, enable the interface at startup and activate the interface. Then it will ping the server in order to check the status.

INPUT - Object IP and client public key
RETURN - Void



```
{
    "ip": "10.13.13.4",
    "publicKey": "o8ZpczSxNtyT+AHUvcHrIc8finLb3ScnV5YwhSoTVUA="
}
```

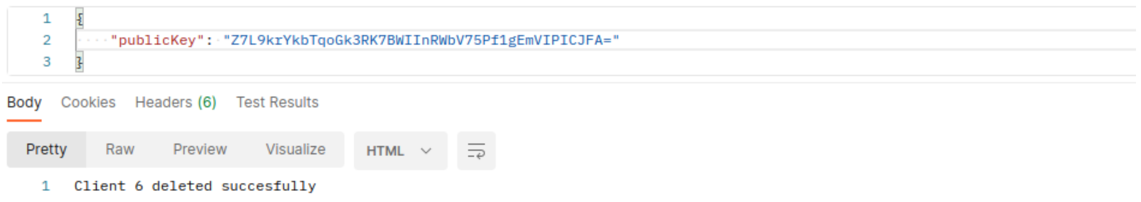
```
File ready in /etc/wireguard/
```

Figure 4.7: API - PUT client create

4.8 DELETE client-Peers from server

DELETE - /server/
Through a passed public key (String) we want to delete the correspondent client. It deletes the rows in the server configuration file and removes the route.

INPUT - Object IP and client public key
RETURN - Void



```
{
    "publicKey": "Z7L9krYkbTqoGk3RK7BWIInRWbV75Pf1gEmVIPICJFA="
}
```

```
Client 6 deleted succesfully
```

Figure 4.8: API - DELETE client-Peers from server