Politecnico di Torino

NAT Traversal Laboratory

Network and Cloud Security

Claudia Sanna, Carlo Federico Vescovo

May 31, 2025

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

You are free:

- **to Share**: to copy, distribute and transmit the work
- **to Remix**: to adapt the work

Under the following conditions:

- **Attribution**: you must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Noncommercial**: you may not use this work for commercial purposes.
- **Share Alike**: if you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

More information on the Creative Commons website.

Contents

1 Introduction

Nowadays networks have to face two crucial challenges: reaching devices behind NATs and ensuring a secure connection among all nodes: NAT Traversal Techniques and VPNs are the fundamental tools to solve this issues.
The aim of this laboratory experience is to understand how NATs work and what techniques can be effectively employed to traverse them. Moreover, we will see how Wireguard creates secure tunnels between hosts and how can the Rust implementation of libp2p, a modular network framework for peer-to-peer connections, assist us with developing a peer-to-peer application. No prior knowledge of Wireguard, neither libp2p or Rust is required.
Moreover, some network tools (e.g. Wireshark) will be used to analyse the packets exchanged between hosts and some network devices (e.g. MikroTik and various Linux hosts) will be configured inside a GNS3 environment to create simplified simulations of real world scenatios. At the end of this experience, you will be able to configure Wireguard VPN tunnels, understand (and, if you are interested, write) some basic libp2p code in Rust and debug common networking issues that might arise when working with NATted networks. Overall, from a theoretical standpoint, you will have gained a deeper understanding of how the various categories of NAT work, how they affects the packets exchanged between peers and how to traverse them.
As far as the structure of this laboratory is concerned, as already mentioned, it consists of sequential, simplified, real world scenarios, starting from a simple Wireguard network to a more complex solution aimed at communicating using direct connections demonstrated by a basic chat application. In the following section, some theoretical background common to all scenarios will be provided, whereas in sections 4 and 5 each scenario will first be analysed from a technical point of view, giving more insights in the specific tools employed, and later be implemented step-by-step.

2 Theoretical background on NATs

Before proceeding with the implementation of our NAT Traversal techniques, some theoretical background on network address translation is needed. Moreover, some fundamental tools, like Wireshark and GNS3, of which some basic knowledge is taken for granted, will be briefly presented in the following Chapter.

2.1 Introduction

Network Address Translation (NAT) is a technique used to modify the IP address and/or port number of packets as they pass through a router or firewall. NAT is commonly used to allow multiple devices on a private network to share a single public IP address. This is particularly important when working on IPv4 networks, since the top level IPv4 address pool has been completely exhausted in 2011 [3] and, in particular, the Europe, Middle East and Central Asia pool managed by RIPE has run out of addresses in 2019 [18]: as there are more devices connected to the Internet than available addresses, multiple hosts will have to share the same public address. Two-way communication can still be ensured by employing ports as another discerning factor: this approach, however, as explained in the next paragraphs, comes with multiple drawbacks. Moreover, keep in mind that ports are a TCP/UDP concept (L4, Transport Layer), while the Internet Protocol sits at layer 3 in the network stack. [14]

2.2 NAT Types

There are two macro-categories of NATs:

- Basic NAT (one-to-one), mainly employed to interconnect IP networks with incompatible addresses. This type of NAT is mentioned here for completeness but it will not be covered in this laboratory experience as it is less common and its traversal is trivial.
- One-to-many NAT, which maps multiple private hosts, each one having a private IP address, to one - sometimes public - external IP address. The aim is to have a singular public IP address for multiple devices in order to tackle IPv4 address space exhaustion. For TCP segments and UDP datagrams, as outgoing traffic passes throughout the NAT device (usually, a router), intended to reach external destinations, the device replaces the private source IP address in each packet header with its own public IP address and the source port with another one, later used to correctly forward incoming packets. Indeed, as incoming traffic reaches the NAT device, it replaces back the original source IP address and port in the destination fields. Therefore, the device performs a kind of connection tracking. This process is sometimes more strictly named network address and port translation (NAPT).

In turn, there are many types of One-to-many NATs:

- Endpoint-Independent NAT, more widely known as **Full Cone NAT**: Once an internal address (iAddr:iPort) is mapped to an external address (eAddr:ePort), any packets from iAddr:iPort are sent through eAddr:ePort. Any external host can send packets to iAddr:iPort by sending packets to eAddr:ePort. This the simplest type of NAPT.

- Address-Dependent NAT, also called **Restricted Cone NAT**: as far as outgoing traffic is concerned, the behaviour is the same as Full Cone NAT. However, incoming traffic gets forwarded to the internal hosts only if iAddr:iPort has previously sent a packet to hAddr:any. Any means the port number doesn't matter.
- Address and Port-Dependent NAT, also known as **Port Restricted Cone NAT**: same as Restricted Cone NAT but the port numbers matter too.
- Address and Port-Dependent NAT, commonly referred to as **Symmetric NAT**: The combination of one internal IP address and a destination IP address and port is mapped to a single unique external source IP address and port; if the same internal host sends a packet even with the same source address and port but to a different destination, a different mapping is used. Only an external host that receives a packet from an internal host can send a packet back.

2.3 The false sense of security provided by NAT

One of the main issues with NATs is that they introduce an additional layer of complexity in IPv4 networks, requiring applications to use TCP/UDP (as NATs make extensive use of ports), preventing in some cases direct communication between hosts and causing performance issues, since packets must be modified by the routers performing the translation. Some network administrators and many users think that NATs provide enough security by itself but this is not true: NAT creates a form of obscurity, by hiding internal network organization, but does substitute a proper firewall configuration nor a comprehensive security strategy [19]. NAT should be used for NAT needs, not for security, as it provides very little for the effort involved as, in general, NAT only translates addresses, while firewall filtering rules block unsolicited or malicious traffic. Therefore, a carefully planned firewall configuration, with or without NAT, provides more fine-grained control, resulting in better security and simpler management [1]. Moreover, NAT can be bypassed if internal hosts start the communication with malicious nodes: in fact, NAT traversal techniques take advantage of this behaviour to achieve direct communication between hosts behind NATs.

2.4 NAT Traversal

2.4.1 Port Forwarding and UPnP

When hosts belonging to different LANs try to communicate, they cannot do it directly as they are not assigned publicly routable IP addresses. One way to solve this problem is to use port forwarding and address incoming packets to the router public IP address, letting the router forward the traffic to the target machine. However, this is impractical, as it requires user configuration, it is not dynamic and it is not always feasible, especially on mobile networks, where ISPs deploy their own NATs. An incomplete, and rather weak, as far as security is concerned, solution, has been developed over the last 20 years as part of the Universal Plug and Play protocol, commonly referred to as UPnP. It alllows devices that need peer to peer communication to autodiscover networking devices, like routers and firewalls, and automatically configure port mappings using the UPnP-IGD protocol. However, form a security standpoint, UPnP in general and UPnP-IGD in particular have been found vulnerable to multiple attacks: since UPnP does not implement any authentication techniques [15] and some of its implementations by netwoking devices have been poorly designed [16]. Modern alternatives to

UPnP-IGD have been developed over time, like NAT-PMP (Apple) and its successor PCP, but have seen limited adoption by IT vendors, especially on consumer devices.

2.4.2 Other techniques

In addition to simple, port-forwarding-based techniques, which often require manual configuration and support from networking devices vendors and ISPs, there are other techniques that can help us traverse NATs. Most of these are based on tricks that exploit the way in which NATs work, so for different kinds of NATs the traversal success rate might vary. In fact, if both nodes are located behind symmetric NATs, a direct connection cannot be set up, always requiring some kind of relay server.

Hole punching

Hole punching is a NAT traversal technique that lets non-public nodes (i.e. behind firewall / NAT) create a direct connection with each other: in any case, the presence of a relay node is required. This node has to be reachable from both peers but it is no longer needed after a direct connection is established. The working principle is the following: Node A and B connect to relay node R, which observes and stores their own addresses and port information, which are the ones they are connecting from; R then relays each node's information to the other one. This data will be used by each node to establish a direct communication with the other one, as they will simply have to address packets to each other using the appropriate observed address and port.
The communication then happens in this way:

1. both nodes send the first packet that passes through their respective routers[1]
2. router adds a 5-tuple (source IP address, source port, destination IP address, destination port, and transport protocol) to their router state table, a data structure containing routes to particular network destinations.
3. the packets sent by each node 'punch holes' into their respective router's firewall
4. both packets arrive at the respective end router
5. once A's packet arrives to B's router, the latter checks its state table and reads the 5-tuple added by node B
6. The router forwards the packet through the 'punched hole' to B and the same occurs to the other side with B's packet.

This mechanism requires synchronisation between A and B. Moreover, if the peers stop talking to each other, the routers will forget about them and close the holes they punched.
Hole punching has been formalized and standardized as a set of methods and protocols in RFC 3489 [21], later superseeded by RFC 5389 [20], and it is called STUN (Session Traversal Utilities for NAT[2]). For further information, read both documents.
STUN is now employed as a tool by ICE, WebRTC and SIP and it inspired, through ICE, the design and development of libp2p, which we wil analyze in Section 5.2.

[1] Here, router does both network address translation and routing
[2] This is the current meaning of the acronym (after RFC 5389); previously, in RFC 3489, STUN stood for Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators

Relayed connection

One of the greatest advantages of hole punching is that the relay server is needed only when creating the direct connection between peers but it does not need to forward any packet during the conversation among nodes. However, when working behind symmetric NATs, hole punching cannot be employed, as the NAT box will now allow packets directed to the punched hole only from the original target we contacted to open it, making it useless when trying to talk with another peer. Indeed, if peers are behind symmetric NATs, the only possible solution consist of relaying everything through the relay server. In this way, though, network bandwidth usage (and costs) will increase, making the communication less efficient.
An example standardization of this solution has been formalized and standardized in RFC 8656 [17] and is called TURN (Traversal Using Relays around NAT).
Because of the bandwidth implications of using TURN-like solutions, protocols supporting both hole punching and relayed connections should always try to perform STUN (or similar techniques) first, and only if they fail retry with TURN.

3 Tools

3.1 GNS3

For this laborartory, since we want to simulate various network scenarios, we chose to use GNS3, a network simulation tool that streamlines the design and implementation of topologies of any complexity by letting us connect simulated devices (both as VMs and as Docker containers) to each other. It supports simulating networks with firewalls, routers, switches and generic virtual machines. GNS3 can also be used to connect virtual devices to real devices, allowing us to create hybrid networks.

3.1.1 Templates

Devices in GNS3 are generally called templates and can be both created manually, depending on the underlying technology that runs them (e.g. Docker, QEMU, VirtualBox, ...) and automatically as ready-to-use appliances provided by the GNS3 community on GNS3 servers. This laboratory employs both GNS3 provided appliances and Docker containers, with Docker images created and uploaded to Docker Hub by the authors of this work. In the next paragraphs, a short guide will explain how to import what we need to complete our activity. You can skip it for now and come back to it when performing the activities listed in Sections 4 and 5

3.1.2 How to import a QEMU image from pre-existing templates

To create a new device from a QEMU image using an existing GNS3 appliance, carefully follow the steps reported here.

1. File → New Template (Figure 3.1)
2. 'Install an Appliance from the GNS3 server (recommended)' and click on 'Next >' (Figure 3.2)
3. Choose the device category and the specific device you are interested in, which will be both specified for each device we will use during this laboratory. Then click on 'Install'. (Figure 3.3)
4. 'Install the Appliance on the GNS3 VM (recommended)' and click on 'Next >' (Figure 3.4)
5. Keep the current, default selection for QEMU binary and on 'Next >' (Figure 3.5)
6. Open the drop-down at the version you want to install, which will be specified for every device we will import in this laboratory. Then click on the image name and then on the 'Download' button. (Figure 3.5).
7. Extract the image from zip file and then click on 'Import'. The status will become 'Ready to Install', so you can now select the image and click on 'Next >' (Figure 3.6).
8. In the newly opened window, click 'Yes' and then 'Finish'.

3.1.3 How to import a Docker Image

As already mentioned, during this laboratory we will employ Docker Images to perform some activities, which can be imported by following the steps reported below.

1. Go to the bar at the top and open the menu 'Edit' and then click on 'Preferences' (Figure 3.7)
2. A window as in Figure 3.8 will appear, go to 'Docker' and then to 'Docker container templates'.
3. Click on 'New' to create a new image, select 'Run this Docker container on the GNS3 VM' and then 'Next >' (Figure 3.9).
4. Select 'New image' and insert the name (Figure 3.10), which for each image will be provided, and then click on 'Next >'.
5. In the next window you can set the name of the device as you prefer.

All the containers we will use have the following configurations:

- Adapters: 1
- Start command: keep it empty
- Console type: telnet
- Environment: keep it empty

Then click on 'Finish': we will find our new device in 'Browse all devices' in the sidebar.

DHCP

Unless otherwise specified, in the following scenarios we will use DHCP autoconfiguration for the networking interfaces of our Docker containers. To enable DHCP autoconfiguration, after dragging the device in the GNS3 environment, right click on it, go to 'Configure', then click on 'Edit' next to 'Network Configuration'. We have to decomment the following lines:

```
auto eth0
iface eth0 inet dhcp
```

3.2 Wireshark

Wireshark is the most popular free and open source, cross-platform network protocol analyser which lets us capture and analyse the packets exchanged between the peers. Wireshark can be used to inspect the packets at different layers of the OSI model, including the application layer, transport layer and network layer. It is able to detect hundreds of protocols, perform live capture and offline analysis, capture compressed traffic and decompress it on the fly and it can also be used to filter packets based on different criteria, such as IP address, port number and protocol. Wireshark lets the user put network interface controllers into promiscuous mode in order to allow them to see all the traffic visible on that interface including unicast traffic not sent to that network interface controller's MAC address.[28]

3.2.1 WireGuard fields in Wireshark

To better understand WireGuard protocol, we can capture traffic with Wireshark and explore it using some filters containing the field names in Table 3.1. The protocol name used to identify Wireguard messages in Wireshark is wg, so fields will have the form wg.<field_name> and the standard port used in Wireguard configurations is 51820 (over UDP), so the corresponding filter is `udp port 51820`. [27]

Field name	Description
wg.timestamp.value	Timestamp
wg.encrypted_packet	Encrypted Packet
wg.encrypted_static	Encrypted Static
wg.encrypted_timestamp	Encrypted Timestamp
wg.ephemeral	Ephemeral
wg.handshake_ok	Handshake decryption successful
wg.receiver	Receiver
wg.receiver_pubkey	Receiver Static Public Key
wg.sender	Sender

Table 3.1

3.3 MikroTik routers

This experiment will utilize MikroTik RB4011 (virtual) routers for our testing. This appliance was selected for numerous reasons, top of which is its ease of use and popularity among the hobbyist and small professional networking community. The RB4011 is relatively affordable compared to enterprise-grade equipment, making it the better choice for academic and test environments where budgets are a concern. Despite all the low cost, the device is extremely configurable and full-featured, supporting advanced routing protocols, firewall rules, NAT configurations, and traffic management schemes. This makes the device particularly valuable for testing complex network topologies and behaviours without the overhead or limitations of less flexible consumer-grade devices. A key point in the selection was the authors' prior experience and technical background with MikroTik's RouterOS system, ensuring that the focus is maintained on the network scenarios and not diverted into trying to learn a new platform. Also, that MikroTik is a Latvian European company provides an added regional taste to our choice. This is in line with larger goals of promoting European technology suppliers and lowering the historically U.S.-dominated universe of networking equipment.
Together, cost-effectiveness, richness of features, user familiarity, and local production make the MikroTik RB4011 an ideal platform for the experimental setup described in this lab.

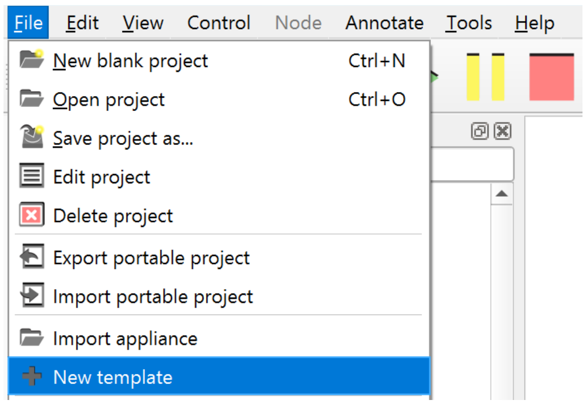
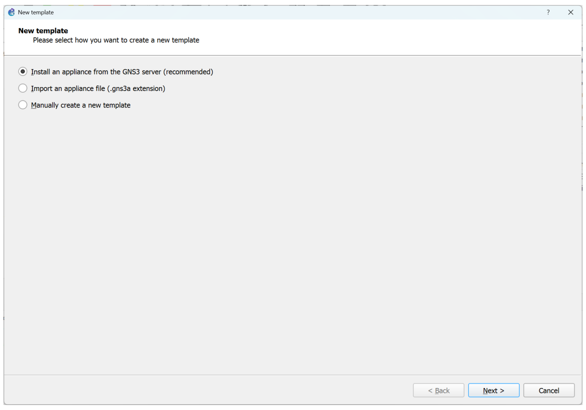


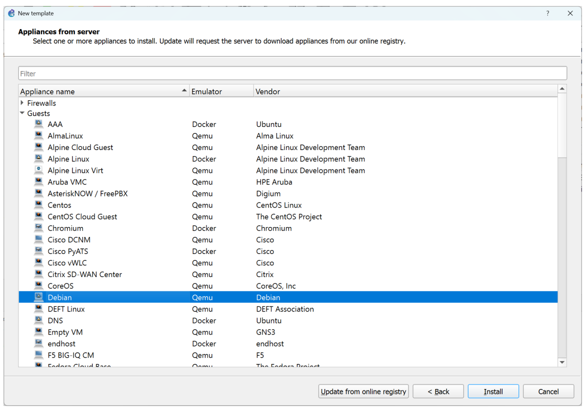
Figure 3.1



New template
Please select how you want to create a new template

[✓ Install an appliance from the GNS3 server (recommended)]
Import an appliance file (.gns3a extension)
Manually create a new template

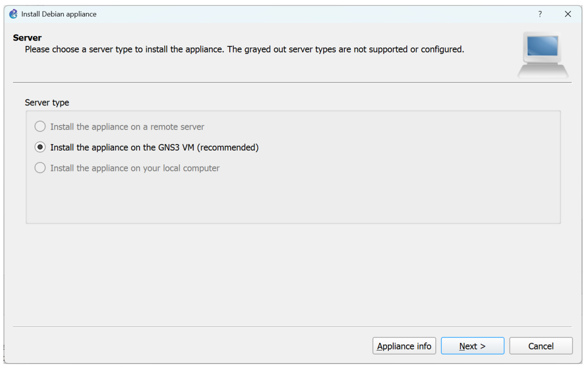
Figure 3.2



Appliances from server
Select one or more appliances to install. Update will request the server to download appliances from our online registry.

Appliance name	Emulator	Vendor
Firewalls		
Guests		
AAA	Docker	Ubuntu
AlmaLinux	Qemu	Alma Linux
Alpine Cloud Guest	Qemu	Alpine Linux Development Team
Alpine Linux	Docker	Alpine Linux Development Team
Alpine Linux Virt	Qemu	Alpine Linux Development Team
Aruba VMC	Qemu	HPE Aruba
AsteriskNOW / FreePBX	Qemu	Digium
Centos	Qemu	CentOS Linux
CentOS Cloud Guest	Qemu	The CentOS Project
Chromium	Docker	Chromium
Cisco DCNM	Qemu	Cisco
Cisco PyATS	Docker	Cisco
Cisco vWLC	Qemu	Cisco
Citrix SD-WAN Center	Qemu	Citrix
CoreOS	Qemu	CoreOS, Inc
Debian	Qemu	Debian
DEFT Linux	Qemu	DEFT Association
DNS	Docker	Ubuntu
Empty VM	Qemu	GNS3
endhost	Docker	endhost
F5 BIG-IQ CM	Qemu	F5

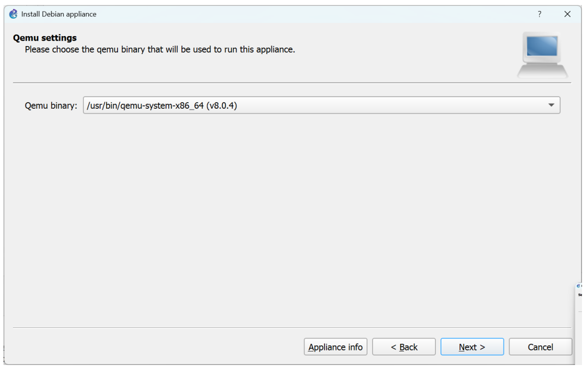
Figure 3.3



Server
Please choose a server type to install the appliance. The grayed out server types are not supported or configured.

Server type
Install the appliance on a remote server
[✓ Install the appliance on the GNS3 VM (recommended)]
Install the appliance on your local computer

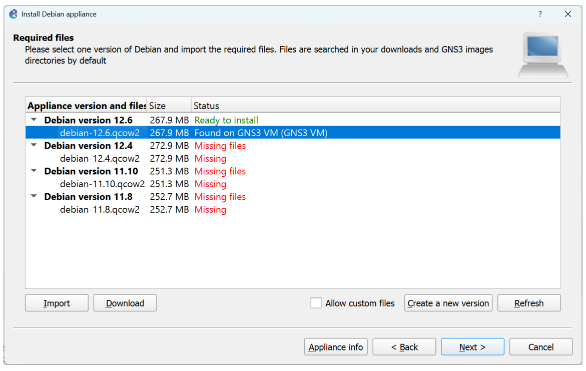
Figure 3.4



Qemu settings
Please choose the qemu binary that will be used to run this appliance.

Qemu binary: /usr/bin/qemu-system-x86_64 (v8.0.4)

Figure 3.5



Required files
Please select one version of Debian and import the required files. Files are searched in your downloads and GNS3 images directories by default

Appliance version and files	Size	Status
Debian version 12.6	267.9 MB	Ready to install
debian-12.6.qcow2	267.9 MB	Found on GNS3 VM (GNS3 VM)
Debian version 12.4	272.9 MB	Missing files
debian-12.4.qcow2	272.9 MB	Missing
Debian version 11.10	251.3 MB	Missing files
debian-11.10.qcow2	251.3 MB	Missing
Debian version 11.8	252.7 MB	Missing files
debian-11.8.qcow2	252.7 MB	Missing

Figure 3.6

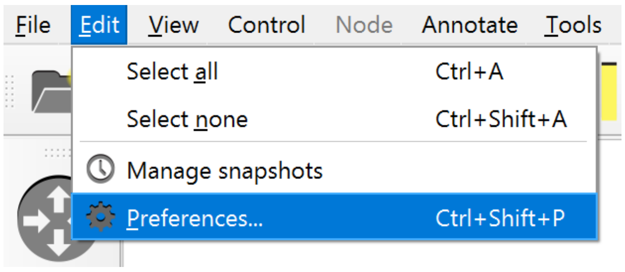


Figure 3.7

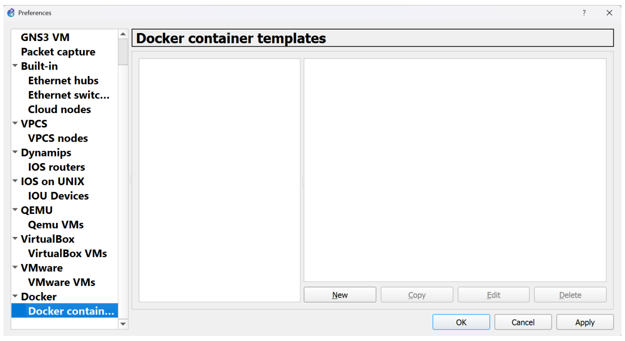
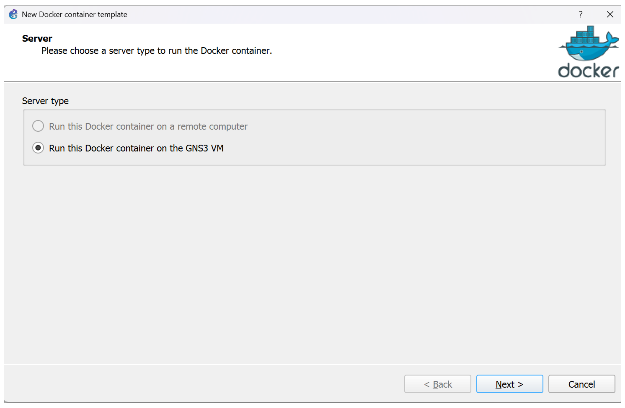


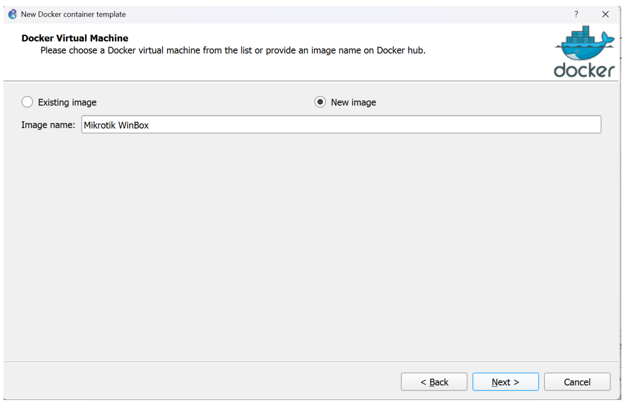
Figure 3.8



Server
Please choose a server type to run the Docker container.

Server type
Run this Docker container on a remote computer
[✓ Run this Docker container on the GNS3 VM]

Figure 3.9



Docker Virtual Machine
Please choose a Docker virtual machine from the list or provide an image name on Docker hub.

Existing image [✓ New image]
Image name: Mikrotik WinBox

Figure 3.10

4 Scenario A

As first solution for NAT Traversal we are going to create a Wireguard tunnel between peers we want to connect. In this overly simplified situation, B and C represent two 'home routers' provided by an ISP, whereas A is a core Internet router. Moreover, A in this case acts also as a Wireguard server: in a real life scenario we would use a proper server, since it is not feasible to ask to the ISP for this service. Since the two hosts (H1 and H2) cannot directly communicate because they are in two different LANs, we need a third peer that can forward packets between them. For this purpose, we are going to set up a VPN tunnel between the router and each peer, and then another channel, a virtual one, in order to prevent the router from being able to eavesdrop the traffic. Before implementing our solution, a theoretical introduction about Wireguard is due.

4.1 Topology

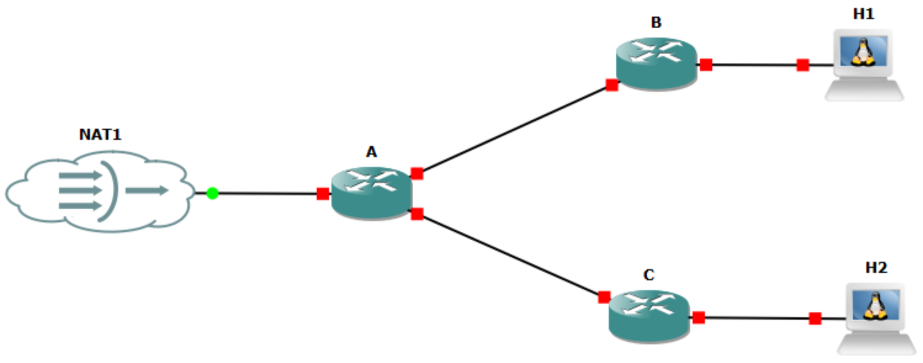


Figure 4.1

4.2 Wireguard

Wireguard is a modern VPN protocol that aims to be simple, fast and secure. It uses state-of-the-art cryptography and is designed to be easy to configure and use. Its purpose is to be faster and simpler than IPSec, avoiding the huge configuration headache, and this is the reason why this laboratory experience uses it instead of IPsec. Wireguard is based on the concept of "peers", which are the endpoints of the VPN tunnel. Each peer has a public and private key pair, which are used to authenticate and encrypt the packets exchanged between them. Wireguard uses UDP as the transport protocol and it is designed to be lightweight and efficient.

4.2.1 Wireguard Architecture

WireGuard is a modern VPN solution that establishes an encrypted Layer 3 network tunnel using UDP, which operates at Layer 4 in the network stack. It represents a much faster and easier substitute for IPsec, integrated directly into the Linux kernel. At its core, WireGuard relies on a minimalistic yet powerful design: virtual tunnel interface is built around basic associations of peer public keys to their corresponding IP addresses. The session is managed in the background, and the protocol features strong, state-of-the-art cryptography. One of the greatest advantages of WireGuard is that it is extremely lightweight and simple to use. The Linux kernel implementation is also very lean at approximately 4,000 lines of code. Its lean structure, paired with its use of efficient cryptographic primitives and its kernel-level operation, makes it a perfect option for high-performance usage on embedded devices and routers alike. From the administrative perspective, WireGuard interfaces such as ‘wg0‘ are stateless and can be set up with standard Linux networking tools such as ‘ip(8)‘ and ‘ifconfig(8)‘. For key exchange, WireGuard employs an OpenSSH-inspired system: peers manually exchange their static 32-byte public keys in advance through an out-of-band mechanism in order to have a simple and secure setup.[2]

4.2.2 Wireguard Cryptography

Wireguard uses the following cryptographic primitives:

- ChaCha20 for symmetric encryption, authenticated with Poly1305, using RFC7539’s AEAD construction
- Curve25519 for ECDH
- BLAKE2s for hashing and keyed hashing, described in RFC7693
- SipHash24 for hashtable keys
- HKDF for key derivation, as described in RFC5869

The communication begins with key exchange handshake, which requires 1-RTT as it involves the initiator sending a message to the responder, who will send back another message. After this step, both peers have a shared pair of symmetric keys, one for sending and the other one for receiving. WireGuard does not send any responses to unauthenticated packets and does not memorize state before authentication, making the protocol not exploitable from network scanner and illegitimate peers. However, having the authentication in the first packet represents a vulnerability against replay attack during handshake. In this way the responder will regenerate its ephemeral key, invalidating the session. In order to avoid this issue, the protocol includes 12-byte of timestamp in the first message, which are encrypted and authenticated. The receiver will consider only the greatest timestamp received per peer, and will discard packets with a timestamp lower or equal to what expected. Furthermore, timestamp guarantees that an ongoing secure session cannot be disrupted by replay attack. In order to achieve non-repudiation, the first message is generated as result of a Diffie-Hellman (Curve25519 ECDH) computation based on peers’ static keys used during authentication. As a consequence, an attacker could be able to forge the first message by compromising either one of the static key, but they would not be able to complete the entire handshake. WireGuard offers the opportunity to use a 256-bit-long pre-shared key as an additional layer of encryption between peers. Combining both Curve25519 and pre-shared key is an acceptable trade-off to mitigate a possible future progress in quantum computing: in the future, quantum computers could be able to break Curve25519 and decrypt past traffic if a pre-shared key was not used and the traffic has been previously captured and stored.

Keys

As already mentioned, the protocol begins with an handshake that sets the symmetric keys used during communication. This mechanism, in order to ensure perfect forward secrecy, occurs every few minutes and is based on the current timestamp. Key exchange has some fair properties, as:

- Avoids key-compromise impersonation
- Avoids replay attacks
- Achieves perfect forward secrecy
- Achieves "AKE security"
- Performs identity hiding

The initiator generates their ephemeral key, hash, chaining key and temp as:

```
initiator.chaining_key = HASH(CONSTRUCTION)
initiator.hash = HASH(HASH(initiator.chaining_key || IDENTIFIER) || responder.
    static_public)
initiator.ephemeral_private = DH_GENERATE() # generate a random Curve25519 private
    key, returning 32 bytes of output
```

and then sends the message:

```
msg = handshake_initiation {
    u8 message_type
    u8 reserved_zero[3]
    u32 sender_index
    u8 unencrypted_ephemeral[32]
    u8 encrypted_static[AEAD_LEN(32)]
    u8 encrypted_timestamp[AEAD_LEN(12)]
    u8 mac1[16]
    u8 mac2[16]
}
```

which contains:

```
msg.message_type = 1
msg.reserved_zero = { 0, 0, 0 }
msg.sender_index = little_endian(initiator.sender_index)
msg.unencrypted_ephemeral = DH_PUBKEY(initiator.ephemeral_private)
initiator.hash = HASH(initiator.hash || msg.unencrypted_ephemeral)
temp = HMAC(initiator.chaining_key, msg.unencrypted_ephemeral)
initiator.chaining_key = HMAC(temp, 0x1)
temp = HMAC(initiator.chaining_key, DH(initiator.ephemeral_private, responder.
    static_public))
initiator.chaining_key = HMAC(temp, 0x1)
key = HMAC(temp, initiator.chaining_key || 0x2)
msg.encrypted_static = AEAD(key, 0, initiator.static_public, initiator.hash)
initiator.hash = HASH(initiator.hash || msg.encrypted_static)
temp = HMAC(initiator.chaining_key, DH(initiator.static_private, responder.
    static_public))
initiator.chaining_key = HMAC(temp, 0x1)
key = HMAC(temp, initiator.chaining_key || 0x2)
msg.encrypted_timestamp = AEAD(key, 0, TAI64N(), initiator.hash)
initiator.hash = HASH(initiator.hash || msg.encrypted_timestamp)
msg.mac1 = MAC(HASH(LABEL_MAC1 || responder.static_public), msg[0:offsetof(msg.mac1)
    ])
if (initiator.last_received_cookie is empty or expired)
```

```
    msg.mac2 = [zeros]
else
    msg.mac2 = MAC(initiator.last_received_cookie, msg[0:offsetof(msg.mac2)])
```

The receiver will decrypt the message and will do all these operations in reverse. Initiator and responder exchange another message to conclude the handshake. At the end of each exchange the state is identical on both peers.

Key Derivation

Once transmitter and receiver exchanged those two messages, the initiator is able to compute keys for sending and receiving data.

```
temp1 = HMAC(initiator.chaining_key, [empty])
temp2 = HMAC(temp1, 0x1)
temp3 = HMAC(temp1, temp2 || 0x2)
initiator.sending_key = temp2
initiator.receiving_key = temp3
initiator.sending_key_counter = 0
initiator.receiving_key_counter = 0

temp1 = HMAC(responder.chaining_key, [empty])
temp2 = HMAC(temp1, 0x1)
temp3 = HMAC(temp1, temp2 || 0x2)
responder.receiving_key = temp2
responder.sending_key = temp3
responder.receiving_key_counter = 0
responder.sending_key_counter = 0
```

And then all previous chaining keys, ephemeral keys, and hashes are zeroed out. The initiator uses their initiator.sending_key to encrypt the message, while the receiver uses responder.receiving_key to read it.[26]

4.3 Routers

We can now proceed with the practical activity. We will first create the template for the routers and then configure Router A.

4.3.1 Installation in GNS3

Proceed as explained in Section 3.1.2, choosing 'Routers' as machine type, 'MikroTik RB4011iGS+' as the device to install and 7.8 as the OS version, which is the latest at the time of writing.

4.3.2 Configuration

First, we have to configure router A, so start the device, open the console and set the admin password:

1. start the device from GNS3
2. double click on the router to open the console
3. the default credentials are the following: admin as username, while the password is not set (just press enter)

4. 'Do you want to see the software license?' 'n'
5. Now the router will ask to change your password: you can freely choose any alphanumeric passphrase

We then proceed with configuring the interfaces, IPv4 addressing and Internet connectivity. Before running the commands listed in the following block, drag the 'NAT' cloud and connect it to port 1 of A: this is required in order to connect to the Internet from inside our GNS3 environment.

```
# list interfaces
/interface/print
# print the dhcp client configuration: since the NAT block acts also as a DHCP server
    , we need to have an active DHCP client on ether1. MikroTik by default enables
    dhcp client on ether1.
/ip/dhcp-client/print
# test Internet connectivity
/tool/ping address=1.1.1.1
```

Configure the router to provide internet connectivity and DHCP auto configuration for the devices we will connect downstream by creating a bridge interface for the clients, adding the ports we will connect them to, and configure IPv4 addressing.

```
/interface/bridge add
# for B
/interface/bridge/port add bridge=bridge1 interface=ether2
# for C
/interface/bridge/port add bridge=bridge1 interface=ether3

/ip/address/add address=10.0.0.1/24 interface=bridge1 network=10.0.0.0

#create the address pool from which the dhcp server takes addresses to give out
/ip/pool/add name=dhcp\_pool0 ranges=10.0.0.2-10.0.0.254
```

Create the DHCP server and configure DHCP network paramers

```
/ip/dhcp-server/add interface=bridge1 address pool=dhcp\_pool0
/ip/dhcp-server/network> add address=10.0.0.0/24 gateway=10.0.0.1 dns-server=1.1.1.1
```

We now have to configure the NAT behaviour of router A: since packets originate from multiple hosts in a private network but we only have one IP address to reach the Internet, given out by our NAT cloud in GNS3, we have to set up Network Address Translation. There are two types of NAT in MikroTik routers:

- source NAT or srcnat. This type of NAT is performed on packets that are originated from a natted network. A NAT router replaces the private source address of an IP packet with a new public IP address as it travels through the router. A reverse operation is applied to the reply packets traveling in the other direction.
- destination NAT or dstnat. This type of NAT is performed on packets that are destined for the natted network. It is most commonly used to make hosts on a private network to be accessible from the Internet. A NAT router performing dstnat replaces the destination IP address of an IP packet as it travels through the router toward a private network.

We are now interested in source NAT, since we want our packets to reach the Internet and their replies should get back to the originating devices.

```
/ip/firewall/nat/add chain=srcnat action=masquerade out-interface=ether1
```

Repeat all the previous configuration for both B and C, changing just the IPs.

4.3.3 WireGuard on Router A

We are going to configure WireGuard on router A, since it can be reached by both hosts. As on Linux, on MikroTik routers Wireguard tunnels are set up using interfaces, so we have to create a new one. By default, when creating a new Wireguard interface, the router will also generate public and private keys for us. Take note of the public key, as we will need it later when configuring the tunnel on our Debian hosts.

```
/interface/wireguard/add listen-port=51820
/interface/wireguard/print
/ip/address> add interface=wg1 address=172.16.0.1/24
```

The interface has been correctly set up but no peers have been configured: we will generate the keys on our hosts and later configure them in router A using their public keys.

4.4 Debian on QEMU

We have chosen to use Debian as the operating system for our hosts since it is fairly well-known and widely used in networking environments. A reasonable choice would have been to use Debian on Docker, why did we choose to run it in QEMU? Because Docker does not do virtualisation but only containerisation: Docker containers don't have their own kernel but share it with the host. However, as we need to perform (limited and simple) modifications to the Wireguard kernel module to demonstrate Wireguard traffic decryption (in Section 4.4.4), virtualization is more practical.

4.4.1 Installation in GNS3

Proceed as explained in Section 3.1.2, choosing 'Guests' as machine type, 'Debian (QEMU)' as the device to install and 12.6 as the OS version, which is the latest at the time of writing.

4.4.2 Debian configuration

We just need to configure network connectivity; to do so, open the console and perform the following operations:

```
sudo nano /etc/network/interfaces
# decomment dhcp part (the lines starting with auto and iface under the DHCP config
    comment) & C-o Enter C-x
sudo service networking restart
```

4.4.3 Wireguard configuration

We can now configure the Wireguard tunnel by creating a config file. A useful tool is wg-quick, which can be installed by running `sudo apt install wireguard`.
First, we need to create a pair of asymmetric keys: `wg genkey | tee privatekey | wg pubkey > publickey`.

These commands will create two files: privatekey and publickey. We will now have to set up a peer on router A using the newly created public key on the current host. On Router A, run

```
/interface/wireguard/peers/add interface=wg1 public-key=<the public key you just
    generated> allowed-address=172.16.0.2
```

On the debian host, create the following file, name it wg1.conf and place it in the /etc/wireguard directory

```
    [Interface]
    Address = 172.16.0.2/32
    PrivateKey = <the private key you just generated>

    [Peer]
    PublicKey = <router A public key>
    Endpoint = 10.0.0.1:51820
    AllowedIPs = 0.0.0.0/0
```

Repeat the same steps on the other debian host, using a different IP (e.g. 172.16.0.3) We can now run `sudo wg-quick up wg1` on both machines.
What happens if we try to ping one from the other (e.g. 172.16.0.3 from 172.16.0.3)? It does not work, since the host we are trying to ping has not contacted router A yet (no handshake), as we can confirm using Wireshark. If we keep pinging H2 from H1 and, at the same time, ping H1 from H2, both machines will start displaying successful ICMP pings. If both machines stop sending packets (i.e. ping is stopped), and if we wait for some time, the association in the NAT table in each router (B and C) will expire, so if we retry pinging H2 from H1, it won't work. In fact, Wireguard is a "silent" protocol: if we don't send packets in a Wireguard tunnel, no data will be exchanged. Wireguard provides a simple, yet effective, solution to this issue: dummy "keep-alive" packets, the behaviour of which can be set by specifying the PersistentKeepalive parameter (for each peer) to an appropriate number of seconds. If, for instance, we set this value to 15, Wireguard will send a keep-alive packet to the related peer every 15 seconds, so as to make the NAT box (B and C) see there is still an ongoing communication (even if no useful data is exchanged) and prevent it from forgetting our hosts in its NAT table.

4.4.4 Wireguard traffic decryption

GNS3 provides us with a powerful tool: Wireshark packet analysis in virtual links. If we start it on the link between H1 and B, for instance, while any traffic is being transported over the Wireguard tunnel we have just set up, we will notice that there are handshake packets, as explained in section 4.2.2, and Wireguard traffic packets. However, the payload of the data packets is unintelligible, since it has been encrypted by Wireguard: only the peers (in this case, router A, host H1 and host H2) can decrypt it. For didactic purposes, to help us understand how Wireguard works in practice, we will now decrypt tunnelled traffic using Wireshark and a simple script used to extract ephemeral keys from the Wireguard kernel module on one of the Debian hosts. Indeed, from section 4.2.2, we know that there is (currently) no way for external attackers (without private key knowledge nor ephemeral key) to decipher our Wireguard communications.

Tool compilation and setup

We will perform the following operations only on one of the two Debian hosts: for this laboratory, we have chosen to work on host H1. First, increasing the root partition size is required: otherwise, the available size on the guest would not be enough to download the required tools and headers and compile the extract-headers project. We decided to increase the disk size by 4096MB so we have to first power off the VM and then increase the image size from GNS3 using the built-in menu. Then, we have to make this space available to our root partition in Debian by extending it using the following commands:

```
sudo apt install parted
sudo parted /dev/sda
print
# parted will ask if we want to increase the size of /dev/sda to the entire disk size
    . Type  Fix  and press Enter.
resizepart
# choose the correct partition (it should be partition 1) and the end (in megabytes)
    based on the output of the print subcommand
quit
reboot
```

We have to upgrade debian packages (and, most importantly, kernel) to the latest available version:

```
sudo apt upgrade
# reboot needed to use new kernel
sudo reboot
```

We can now download the wireguard-tools git project, which contains the code for the extract-handshakes utility, the linux headers for our specific kernel version, and the linux kernel source, since some files will be needed in order to compile the handshake extractor.

```
sudo apt install git build-essential linux-headers-`(uname -r)`
git clone https://git.zx2c4.com/wireguard-tools
# we download the linux source for version 6.1, since it is the one we are currently
    using. If you want to double check, run uname -a
wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.tar.xz
tar xvf linux-6.1.tar.xz
```

Before compiling the project, we have to edit the import line in the 'offset-finder.c' file since we want it to use the header files we have just downloaded as part of the Linux kernel source. Therefore, open the file with the editor of your choice and edit the line which imports the 'noise.h' header to *#include /home/debian/linux-6.1/drivers/net/wireguard/noise.h*.
Then, run `sudo make`. There is still another edit we have to perform before continuing with traffic decryption: the provided script has been written assuming that every OS/kernel uses the same symbols, but that is not the case. In our configuration, we had to change line 46 from `echo p:wireguard/idxadd index_hashtable_insert ${ARGS[*]}` to `echo p:wireguard/idxadd wg_index_hashtable_insert ${ARGS[*]}`.

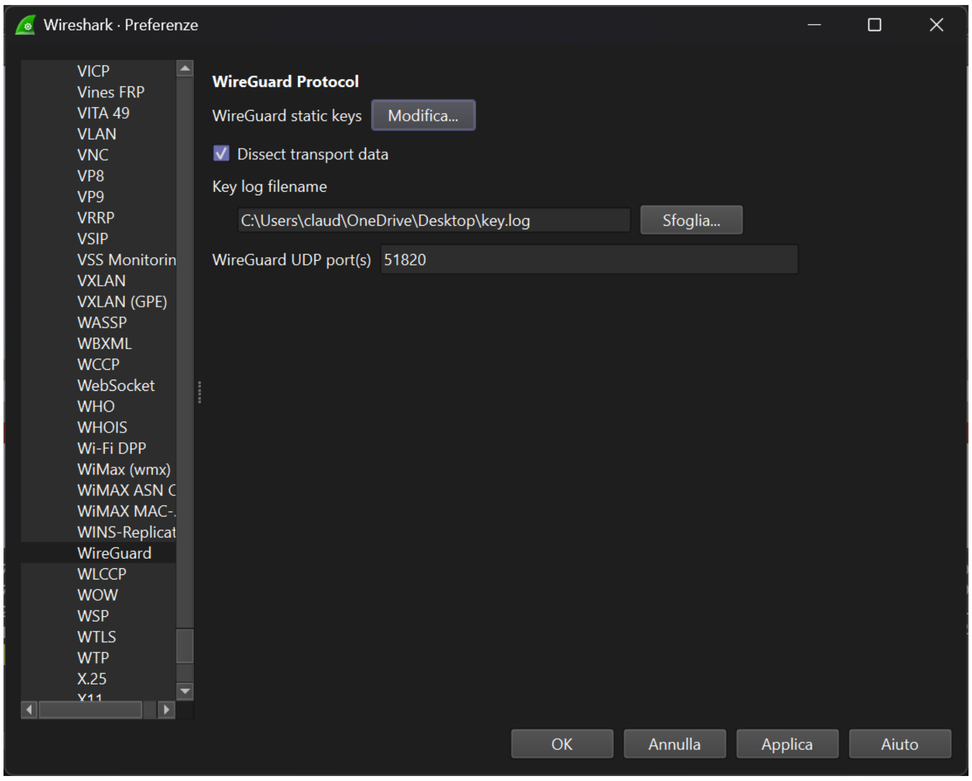
We are now ready to start a new Wireguard tunnel, for which we will capture the keys we need for decryption. First, start capturing traffic from GNS3 using Wireshark by right clicking on the link between H1 and B and selecting "Start capture". Using `screen`, open a new bash session and launch (as root)

```
modprobe wireguard
./extract-handshakes.sh
```

There should be no output following the second command, since we have not started any Wireguard tunnel yet. Exit from the screen using `Ctrl-a d` and create the tunnel using `sudo wg-quick up wg1`.

Now, reopen the screen using `screen -r` and store the four keys the script printed on the standard output. We will put these in Wireshark:

1. Go to the drop-down menu 'Edit' and then click on 'Preferences'
2. A new window will appear, so go to '>Protocols' and expand it.
3. Scroll through the list until you find 'Wireguard' and click on it (Figure 4.2)
4. Import your key file
5. click 'Apply' and then 'Ok'



WireGuard Protocol

WireGuard static keys

☑ Dissect transport data

Key log filename

C:\Users\claud\OneDrive\Desktop\key.log

WireGuard UDP port(s) 51820

Figure 4.2

Exchange some traffic (e.g. ping 1.1.1.1) and we will see the decrypted packets!

4.4.5 Confidentiality

There is still one issue to address: what if hosts H1 and H2 don't trust router A? In our example, we have complete control over the router but in real scenarios the Wireguard server in use might be provided by third parties and, as per our threat model, we might want to prevent them from eavesdropping our conversations. The configuration we have just set up, however, does not provide any guarantee in this regard. A simple and effective solution is creating a second tunnel inside the already created one, in which the two hosts are set up as Wireguard peers w.r.t each other, leaving out router A. In this way, traffic will be encrypted both in the internal tunnel and in the external one.

5 Scenario B

5.1 Topology

Now we are going to set up a network that simplifies a real world scenario: routers B and C act as the 'home routers' provided by an ISP to customers, while A can be thought of as a core Internet router, connected both to B and C and to a relay server, the role of which will be explained in the section below.

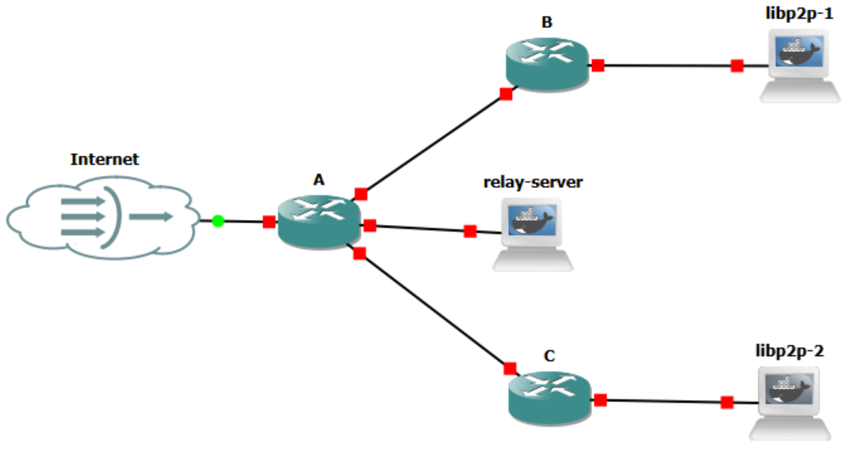


Figure 5.1

5.2 libp2p

Peer-to-peer protocols are widely used in many different scenarios, from real time video-communication, of which an early example is Skype, to file sharing. libp2p, short for Library peer-to-peer, is a network framework which includes many protocols, libraries and specifications that allow the development of peer-to-peer communication. libp2p has many advantages that makes it suitable for our purposes:

- Modularity: it allows developers to customise the network stack as they prefer
- Security: libp2p verifies peer identity by means of public key cryptography and encrypts communications between peers using strong cryptographic algorithms.
- Piercing NAT Barriers: it allows p2p communications even when they are behind NAT devices or firewalls.
- Decentralizations: it is designed with the aim of enabling distributed web
- Interoperability: applications written in different languages can communicate smoothly.

- Documentation: libp2p shows a well-explained and complete documentation that help developers to build their projects.
- Message Distribution and Dissemination: it uses publish/subscribe (pubsub) [8] to implement a flexible and efficient way for exchanging messages by means of the use of protocols like gossipsub. pubsub is a system that allows peers to congregate around a topic they are interested in.

All these properties are the reason why we decided to use this framework. [10]

5.2.1 libp2p Security

The communications between peers take place on authenticated channels between them. Each peer has a pair of keys: one private, kept secret, and a public one, shared with all other peers. Together they allow the peer to establish a secure channel to communicate with other peers. Every p2p peer is identified by a Peer ID [7]which is a cryptographic hash of the peer's public key. When peers establish a connection, the hash can be used to verify that public key used to create the channel is the same one used to identify the peer. Peer IDs are defined as a compact binary format, usually encoded in Base58. libp2p does not provide a built-in system for authorisation, but it can be easily developed as Peer IDs so that their corresponding keys allow to authenticate remote peers and, therefore, an authorisation system can be built upon these data. A simple authorisation system could, for instance, map permissions to Peer IDs and our new, custom application could leverage this map to reject requests from unauthorised peers. It is certainly possible to develop other authorisation systems not based on peer ID.[9]

5.2.2 Hole Punching in libp2p

libp2p hole punching process [6] can be divided into two phases: preparation and proper hole punching. During preparation phase the framework uses a protocol named AutoNAT to discover if a node is behind NAT or firewall (this is equivalent to STUN protocol in ICE). Then nodes dynamically discover and bind to relay nodes on the network (AutoRelay). At the end of this first phase the nodes then connect to and request reservations with the relay nodes by means of a transport protocol named Circuit Relay [4]. Moreover, a node can advertise itself as reachable through the relay node (This is equivalent to TURN protocol in ICE).
Hole punching phase is composed by two steps. In the first one Circuit Relay is still used to establish a secure relay connection through the public relay node, so each node establishes a direct connection with the relay node. Node B requests a relayed connection to A through the relay node, creating a bi-directional secure channel by means of TLS 1.3. The last part of the mechanism implements a synchronisation mechanism to coordinate hole punching (DCUtR).

5.2.3 DCUtR

Relay nodes act as proxies in NAT Traversal, which can be expensive to scale and maintain, obtaining as result low bandwidth and high-latency connections. Generally, hole punching would require, in addition to a relay node, also an infrastructure named signalling server. libp2p2 offers a solution which eliminates the need for centralised signalling servers, allowing us to use distributed relay nodes only.
DCUtR (Direct Connection Upgrade through Relay)[5] is the libp2p protocol which implements this solution; it involves synchronizing and opening connections to each peer. The protocol starts at the end of the relay connection from A to B. [13]

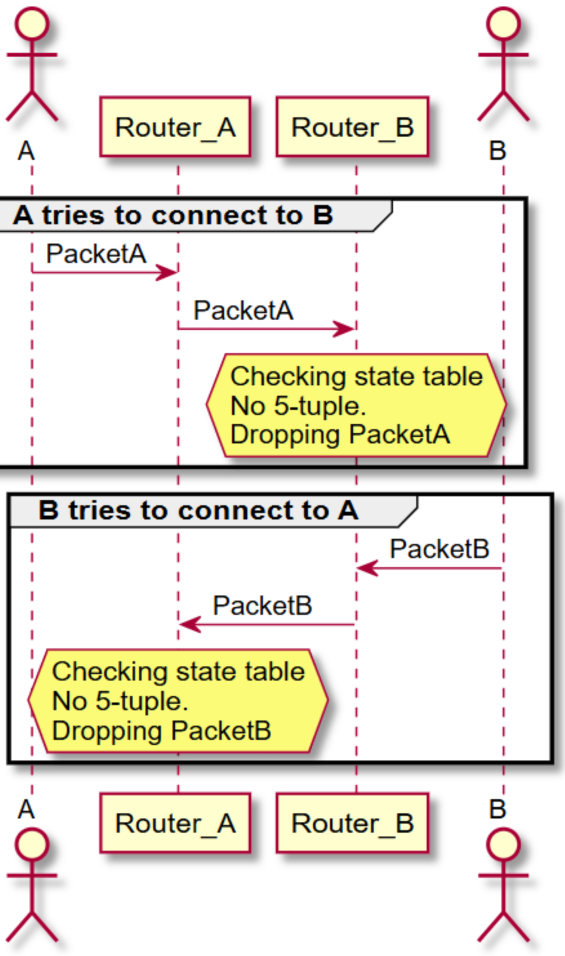


Figure 5.2

- B opens a stream to A using DCUtR
- B sends to A a Connect message containing its observed addresses from identify and starts a timer to measure RTT of the relay connection
- A responds back with a Connect message containing its observed addresses.
- B then sends a Sync message and starts a timer for half the RTT, measured from the time between sending the initial Connect and receiving the response. The purpose of the Sync message and B's timer is to allow the two peers to synchronize so that they perform a simultaneous open that allows hole punching to succeed.
- Upon receiving the Sync, A immediately dials the address to B
- Upon expiry of the timer, B dials the address to A in case of TCP address, or it starts to send UDP packets filled with random bytes to A's address in case of QUIC address. At the end we have a connection, TCP or QUIC, where A is the client and B is the server
- Once a single connection has been established, A SHOULD cancel all outstanding connection attempts.

5.3 Code

5.3.1 Relay Server

As far as the relay server is concerned, we used the example code provided in the rust-libp2p git repository [11], as it showcases how to create a relay node that can route messages between different peers in a p2p network. The first step performed by the server consists of creating its own keys and peer ID. Then, it instantiates a swarm [12], which represents the state of the network as a whole. Swarm needs three things to be correctly created: a peer identity, which we have already generated; an implementation of transport layer (e.g. TCP, QUIC, UDP, etc.) and a definition of how the swarm should behave once it is connected to a node, named NetworkBehaviour. The most relevant component is the behaviour, which is in this case made up of three parts: relay, ping and identify. Relay implements the Circuit Relay protocol and is set to default options in our example, while Ping just checks for connectivity with connected peers and identify collects information of a peer sent in protocol messages. Then the program loops indefinitely listening for new connections and logging them to the console.

5.3.2 Chat

To demonstrate libp2p modularity, we developed a new simple chat protocol between two peers; the Rust implementation we will use for this activity is available on Github [23] but cloning our repository is not mandatory, since we have also provided a Docker image with ready-to-run binaries, as explained in Section 5.4.
Our implementation supposes there are one listener and one dialler. The program creates a swarm with a relay client behaviour and four associated protocols: ping, identify (as before), dcutr and chat. Chat protocol is trivial: it just uses RequestResponse abstraction provided by libp2p to send and receive messages. We followed an event based approach to develop our code by using Tokio [25], an asynchronous Rust runtime to develop networking applications, which allows us to react when the user inserts data from the console and when incoming messages arrive.

5.4 Demo

We have created and published a ready to use Docker image that makes the laboratory experience straightforward, since it contains all we need to run the relay server and to launch the chat clients. We have to retrieve the image, named `cfvescovo/rust-libp2p` [22], as explained in section 3.1.3 and drag it into the GNS3 environment as illustrated in Figure 5.1.

5.4.1 Router A

Configure another IP address for the interface linked to the relay server.

```
/ip/address/add address=100.0.0.1/24 interface=ether3
```

5.4.2 Relay Server

Right click on the relay server machine and edit network configuration: make sure that the DHCP part is commented out and configure statically the interface. The configuration should be:

```
auto eth0
iface eth0 inet static
    address 100.0.0.2
    netmask 255.255.255.0
    gateway 100.0.0.1
    up echo nameserver 1.1.1.1 > /etc/resolv.conf
```

Then, we have to launch the relay server. For this purpose just double click on it to open the console and run

```
relay-server --port <port> --secret-key-seed <seed>
```

Take note of the address of the relay server.

5.4.3 Chat

We have to launch our chat program with some parameters:

- mode: this should be set to 'listen' on the first peer and 'dial' on the second one, as dial also requires the remote peer ID parameter.
- relay-address: this is the aforementioned address of the relay server
- remote-peer-id: required only for dial mode, it should be the peer ID of the listening peer
- secret-key-seed: optional, use it if you want deterministic peer IDs

First, right click on the machine you want to use as the listener (i.e. the one that will receive the first message from the dialler) and launch the chat program in `dial` mode, using the relay address of the relay server. Take note of the local peer ID, it will be printed on the console.
Then, open the console on the other peer (i.e. the dialler) and launch the chat program in `dial` mode, using the relay address of the relay server and, this time, also specify the remote-peer-id of the listener. Send any message and it will be printed on the other machine! We can verify it works both ways by sending messages from the listener too.

5.5 Bonus activities

The simple chat protocol we have implemented only demonstrates one-to-one communication. If you are interested, however, you can extend it to implement group functionality by leveraging the pubsub protocol provided by libp2p [8]. Moreover, another extension you could develop consists of creating a Wireguard tunnel between the nodes after having performed the DCUtR dance with libp2p, removing the need for a third-party server like the one we employed in 4. This approach is similar to the one taken by Tailscale [24], a partially open-source, zero-config, software-defined mesh virtual private network.
Both activities are outside of the scope of this laboratory but, given the technical and practical background built during this experience, you should now be well-equipped to explore these enhancements independently. They offer valuable opportunities to deepen your understanding of peer-to-peer networking and secure communication, and could serve as the foundation for more advanced distributed applications or even future research projects.

Bibliography

[1] Cybersecurity and Infrastructure Security Agency. *Securing Network Infrastructure Devices.* Accessed: 2025-05-30. Sept. 2006. URL: https://www.cisa.gov/news-events/news/securing-network-infrastructure-devices.

[2] Jason A. Donenfeld. *WireGuard: Next Generation Kernel Network Tunnen.* Tech. rep. e2da747. Accessed: 2025-05-30. June 2020. URL: https://www.wireguard.com/papers/wireguard.pdf.

[3] Internet Corporation for Assigned Names and Numbers (ICANN). *Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied.* Accessed: 2025-05-30. Feb. 2011. URL: https://itp.cdn.icann.org/en/files/announcements/release-03feb11-en.pdf.

[4] libp2p Documentation Team. *Circuit Relay.* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/nat/circuit-relay/.

[5] libp2p Documentation Team. *Direct Connection Upgrade through Relay (DCUtR).* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/nat/dcutr/.

[6] libp2p Documentation Team. *Hole Punching.* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/nat/hole-punching/.

[7] libp2p Documentation Team. *Peer ID.* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/fundamentals/peers/#peer-id.

[8] libp2p Documentation Team. *Publish/Subscribe Overview.* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/pubsub/overview/.

[9] libp2p Documentation Team. *Security Considerations.* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/security/security-considerations/.

[10] libp2p Documentation Team. *What is libp2p.* Accessed: 2025-05-30. 2025. URL: https://docs.libp2p.io/concepts/introduction/overview/.

[11] libp2p Project. *rust-libp2p: The Rust Implementation of the libp2p Networking Stack.* Accessed: 2025-05-30. 2025. URL: https://github.com/libp2p/rust-libp2p.

[12] libp2p Rust Documentation Team. *libp2p::swarm - Rust Documentation.* Accessed: 2025-05-30. 2025. URL: https://docs.rs/libp2p/latest/libp2p/swarm/index.html.

[13] libp2p Specification Team. *DCUtR Specification.* Accessed: 2025-05-30. 2025. URL: https://github.com/libp2p/specs/blob/master/relay/DCUtR.md.

[14] NAT Wiki Contributors. *NAT.* Accessed: 2025-05-30. 2025. URL: https://en.wikipedia.org/wiki/Network_address_translation.

[15] Petko D. Petkov. *Hacking The Interwebs.* Accessed: 2025-05-30. Jan. 2008. URL: https://www.gnucitizen.org/blog/hacking-the-interwebs/.

[16] Rapid7 Community. *Universal Plug and Play (UPnP) Security Risks.* Accessed: 2025-05-30. 2013. URL: https://web.archive.org/web/20160305060436/https://community.rapid7.com/docs/DOC-2150.

[17] T. Reddy et al. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. Tech. rep. RFC 8656. Accessed: 2025-05-30. Internet Engineering Task Force, Feb. 2020. URL: https://www.rfc-editor.org/rfc/rfc8656.

[18] RIPE Network Coordination Centre. *The RIPE NCC has run out of IPv4 Addresses*. Accessed: 2025-05-30. Nov. 2019. URL: https://www.ripe.net/about-us/news/the-ripe-ncc-has-run-out-of-ipv4-addresses/.

[19] Rochester Institute of Technology. *A Re-examination of network address translation security*. Accessed: 2025-05-30. 2010. URL: https://repository.rit.edu/cgi/viewcontent.cgi?httpsredir=1&article=1764&context=other.

[20] J. Rosenberg et al. *Session Traversal Utilities for NAT (STUN)*. Tech. rep. RFC 5389. Accessed: 2025-05-30. Internet Engineering Task Force, Oct. 2008. URL: https://www.rfc-editor.org/rfc/rfc5389.

[21] J. Rosenberg et al. *STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)*. Tech. rep. RFC 3489. Accessed: 2025-05-30. Internet Engineering Task Force, Mar. 2003. URL: https://www.rfc-editor.org/rfc/rfc3489.

[22] C. Sanna and C. F. Vescovo. *cfvescovo/rust-libp2p*. Accessed: 2025-05-30. 2025. URL: https://hub.docker.com/r/cfvescovo/rust-libp2p.

[23] C. Sanna and C. F. Vescovo. *ncs-project*. Accessed: 2025-05-30. 2025. URL: https://github.com/cfvescovo/ncs-project.

[24] Tailscale Inc. *Tailscale: VPN Service for Secure Networks*. Accessed: 2025-05-30. 2025. URL: https://tailscale.com/.

[25] Tokio Project. *Tokio: An Asynchronous Runtime for Rust*. Accessed: 2025-05-30. 2025. URL: https://tokio.rs/.

[26] WireGuard Project. *Protocol & Cryptography - WireGuard*. Accessed: 2025-05-30. 2025. URL: https://www.wireguard.com/protocol/.

[27] Wireshark Documentation Team. *Display Filter Reference: WireGuard Protocol*. Accessed: 2025-05-30. 2025. URL: https://www.wireshark.org/docs/dfref/w/wg.html.

[28] WireShark Wiki Contributors. *Wireshark Wiki*. Accessed: 2025-05-30. 2025. URL: https://en.wikipedia.org/wiki/Wireshark.