Lab: HTTP/2, TLS and SSH

In this lab, you will explore how modern transport and application-layer protocols improve upon their predecessors. You will start by comparing HTTP/1.1 and HTTP/2 using command-line tools (`curl` and `nghttp`), then analyze packet traces in Wireshark to understand the differences at the wire level.

Prerequisites

- `curl` (usually pre-installed)
- `nghttp` from the nghttp2 package
- `openssl` (version 1.1.1 or later for TLS 1.3)
- `tshark` and Wireshark for packet capture and analysis
- Python 3 with the `cryptography` module
- Root privileges (`sudo`) for packet capture

Installation on Ubuntu / Debian

```
sudo apt update
sudo apt install curl nghttp2-client wireshark tshark openssl python3-pip
pip3 install cryptography
```

Installation on Windows (WSL2)

These exercises require a Linux environment. On Windows, use **WSL2** (Windows Subsystem for Linux).

1. Install WSL2 if not already done (PowerShell as Administrator):

   ```
   wsl --install -d Ubuntu
   ```

   Restart your machine if prompted, then open the **Ubuntu** terminal from the Start menu.

2. Install the required packages inside WSL2:

   ```
   sudo apt update
   sudo apt install curl nghttp2-client wireshark tshark openssl python3-pip
   pip3 install cryptography
   ```

3. For Wireshark, you have two options:

   - **Option A** — Install Wireshark for Windows from https://www.wireshark.org and open `.pcap` files from the Windows side. Your WSL2 home directory is accessible at `\wsl$\Ubuntu\home\<username>\` in File Explorer.
   - **Option B** — Run `wireshark` inside WSL2 with a graphical display. This requires WSLg (included in Windows 11 and recent Windows 10 updates):

     ```
     sudo apt install wireshark
     wireshark capture.pcap
     ```

4. `tshark` may ask whether non-root users should be allowed to capture packets. Select **Yes**, then add your user to the `wireshark` group:

   ```
   sudo usermod -aG wireshark $USER
   ```

Log out and back in (or run `newgrp wireshark`) for the change to take effect. After this, `tshark` and the capture scripts work with `sudo`.

> **Note**
>
> WSL2 shares the host network stack. Traffic captured with `tshark` inside WSL2 is real network traffic, identical to what you would see on a native Linux machine.

Installation on macOS

```
brew install curl nghttp2 wireshark openssl python3
pip3 install cryptography
```

> **Note**
>
> The system `openssl` on macOS is actually LibreSSL. Some benchmarks may be missing. Use the Homebrew version:
>
> ```
> export PATH="/opt/homebrew/opt/openssl/bin:$PATH"
> openssl version    # should show OpenSSL, not LibreSSL
> ```

Test server

All exercises use port 8080 of the server `test.computer-networking.info`

The server hosts files of various sizes at `/files/`:

Path	Size
`/files/1KB`	1 KB
`/files/2KB`	2 KB
`/files/3KB`	3 KB
`/files/4KB`	4 KB
`/files/5KB`	5 KB
`/files/10KB`	10 KB
`/files/100KB`	100 KB
`/files/1000KB`	1 MB

HTTP/1.1 with curl

HTTP/1.1 (RFC 9112) is a text-based protocol running over TCP. Each request/response exchange occupies the TCP connection sequentially: the client must wait for the response to the first request before sending the next one. This is known as **head-of-line blocking at the application layer**.

To work around this limitation, browsers typically open **multiple TCP connections** in parallel (often 6 per hostname). In this section you will observe this behaviour using `curl`.

Download a single file

Use `curl` to fetch a small file from the test server:

```
curl -v -o /dev/null http://test.computer-networking.info:8080/files/10KB
```

The `-v` flag enables verbose output, which shows the HTTP request and response headers. The `-o /dev/null` option discards the downloaded data (we only care about the protocol exchange).

Observe the output and answer:

- What HTTP version does the server use in the response status line?
- What headers does the server include in the response?
- What is the `Content-Length` of the 10KB file?

Download multiple files sequentially

HTTP/1.1 can reuse a TCP connection for multiple requests (`Connection: keep-alive`), but requests are processed **one at a time**. Use `curl` to download four files sequentially over a single connection:

```
curl -v -o /dev/null \
    http://test.computer-networking.info:8080/files/1KB \
    http://test.computer-networking.info:8080/files/5KB \
    http://test.computer-networking.info:8080/files/100KB \
    http://test.computer-networking.info:8080/files/1000KB
```

Observe the verbose output carefully:

- Does `curl` open a new TCP connection for each file, or does it reuse the same connection? (Look for `Re-using existing connection`.)
- In what order are the files downloaded? Can a small file "overtake" a large one that was requested before it?
- Note the total time reported by `curl`.

> Hint
>
> Add `-w "Total time: %{time_total}s\n"` to see the total time at the end.

Capture and analyze HTTP/1.1 traffic

Capture the traffic while downloading multiple files. Open a terminal and start `tshark`:

```
sudo tshark -i any -p -w http11-multi.pcap -f "tcp port 8080"
```

In a second terminal, run the same `curl` command as in the previous exercise. Once the download completes, stop `tshark` with `Ctrl-C` and open the capture in Wireshark:

```
wireshark http11-multi.pcap
```

Filter	Description
`http`	Show only HTTP packets
`tcp.port == 8080`	Show all traffic to/from the server
`http.request`	Show only HTTP requests
`http.response`	Show only HTTP responses
`tcp.analysis.retransmission`	Show TCP retransmissions

In the Wireshark trace, answer:

- How many TCP connections were opened? (Look at the SYN packets or use `Statistics` → `Conversations` → `TCP`.)
- Can you identify each GET request and its corresponding response?
- Are the requests strictly sequential? (Look at the timing — does the next request start only after the previous response is fully received?)

HTTP/2 with nghttp

HTTP/2 (RFC 9113) is a binary-framed protocol that multiplexes multiple **streams** over a single TCP connection. Each stream carries one request/response pair, and DATA frames from different streams can be interleaved. This eliminates HTTP-level head-of-line blocking: a small file no longer has to wait behind a large one. However, HTTP/2 still runs over a single TCP connection.

The `nghttp` tool (from the nghttp2 package) is a command-line HTTP/2 client. Its `-v` flag shows individual HTTP/2 frames, including their stream IDs.

Download multiple files over HTTP/2

Use `nghttp` to download the same four files as before, but this time they are all fetched over **a single HTTP/2 connection**:

```
nghttp -v -n \
    http://test.computer-networking.info:8080/files/1KB \
    http://test.computer-networking.info:8080/files/5KB \
    http://test.computer-networking.info:8080/files/100KB \
    http://test.computer-networking.info:8080/files/1000KB
```

The `-n` flag discards downloaded data (like `-o /dev/null` in curl).

Study the verbose output and answer:

- How many streams were opened? (Look for `HEADERS` frames sent by the client)
- What stream ID was assigned to each file? (HTTP/2 stream IDs initiated by the client are odd numbers: 1, 3, 5, 7, …)
- Are `DATA` frames from different streams interleaved, or does one stream finish completely before the next one starts?
- Which file finished downloading first — the 1KB or the 1000KB file?

Capture and analyze HTTP/2 frames

Use the provided script to download files and capture packets simultaneously:

```
cd http2
sudo ./download-streams.sh -4 8080
```

(Use `-6` instead of `-4` if your network supports IPv6.)

The script creates a `.pcap` file. Open it in Wireshark:

```
wireshark http2-streams-port8080.pcap
```

Useful Wireshark display filters for HTTP/2

Filter	Description
`http2`	Show all HTTP/2 frames
`http2.type == 0`	DATA frames only
`http2.type == 1`	HEADERS frames only
`http2.streamid == 1`	Frames for stream 1 only
`http2.streamid == 3`	Frames for stream 3 only
`tcp.analysis.retransmission`	TCP retransmissions

In the Wireshark trace, answer:

- How many TCP connections are used?
- Can you see DATA frames from different streams arriving interleaved in the same TCP connection?
- Compare with the HTTP/1.1 trace captured earlier: how many TCP connections did HTTP/1.1 need for the same four files?

TLS: cryptographic building blocks

TLS (Transport Layer Security) secures TCP connections by providing confidentiality, integrity and authentication. Before studying the TLS handshake itself, it is important to understand the **performance** of the cryptographic primitives it relies on.

TLS 1.3 (RFC 8446) uses a small set of modern algorithms:

- **Key exchange**: Diffie-Hellman (finite-field) or ECDHE (elliptic-curve)
- **Symmetric encryption**: AES-128-GCM, AES-256-GCM, or ChaCha20-Poly1305
- **Digital signatures**: RSA, ECDSA (P-256), or EdDSA (Ed25519)
- **Hash functions**: SHA-256, SHA-384

In this part, you will use `openssl speed` and a Python script to measure how fast your machine can execute each of these algorithms.

Verify that `openssl` is installed and recent enough:

```
openssl version
```

You need OpenSSL 1.1.1 or later. See the installation instructions in the Prerequisites section above if the command is not found.

TLS in practice: certificates and handshake

You will now set up a real TLS 1.3 connection between a local server and client. The first step is to create the **certificates** that the server needs to prove its identity.

In the public Internet, certificates are issued by Certificate Authorities (CAs) that browsers and operating systems trust. For this lab, you will act as your own CA: you will create a root certificate, then use it to sign a server certificate.

The relevant standard is RFC 5280 (X.509 certificates). The `openssl` command-line tool can create keys and certificates.

Create a Certificate Authority (CA)

A CA is simply a key pair and a self-signed certificate. The CA's private key is used to sign other certificates; the CA certificate is distributed to clients so they can verify those signatures.

Step 1 — Generate a private key for your CA:

```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 \
    -out ca-key.pem
```

This creates a 2048-bit RSA private key in `ca-key.pem`.

Step 2 — Create a self-signed CA certificate:

```
openssl req -new -x509 -key ca-key.pem -days 365 \
    -out ca-cert.pem \
    -subj "/C=BE/O=CNP3 Lab/CN=CNP3 Lab CA"
```

This creates `ca-cert.pem`, valid for one year.

Inspect your CA certificate:

```
openssl x509 -in ca-cert.pem -text -noout
```

Answer:

- What is the **Subject** of your CA certificate?
- What is the **Issuer**? Why are they the same?
- What **signature algorithm** was used?
- What are the `notBefore` and `notAfter` dates?

Create a server certificate

Now create a certificate for the TLS server, signed by your CA.

Step 1 — Generate a private key for the server:

```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 \
    -out server-key.pem
```

Step 2 — Create a Certificate Signing Request (CSR). The CSR contains the server's public key and identity, but is not yet signed by the CA:

```
openssl req -new -key server-key.pem \
    -out server.csr \
    -subj "/C=BE/O=CNP3 Lab/CN=localhost"
```

Step 3 — Sign the CSR with your CA key to produce the server certificate. The `subjectAltName` extension is required — without it, modern TLS clients will reject the certificate:

```
openssl x509 -req -in server.csr \
    -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial \
```

```
    -days 365 -out server-cert.pem \
    -extfile <(printf "subjectAltName=DNS:localhost,IP:127.0.0.1,IP:::1")
```

Inspect the server certificate:

```
openssl x509 -in server-cert.pem -text -noout
```

Answer:

- What is the **Subject** of the server certificate?
- What is the **Issuer**? Does it match the CA certificate?
- Find the **Subject Alternative Name** extension — what DNS names and IP addresses are listed?
- What is the relationship between the CA certificate and the server certificate? (Think about the chain of trust.)

Establish a TLS connection

The script `tls/simple-server.py` is a minimal TLS 1.3 server. It accepts one connection, sends "Hello", and exits.

Start the server with your new certificate and key:

```
python3 tls/simple-server.py --cert server-cert.pem --key server-key.pem
```

The server listens on port 8443 by default and supports both IPv4 and IPv6.

The script `tls/tls-client.py` is a TLS 1.3 client that displays the negotiated parameters and the server certificate. It can also capture the traffic to a PCAP file and log the TLS session keys for Wireshark decryption.

First, restart the server:

```
python3 tls/simple-server.py --cert server-cert.pem --key server-key.pem
```

Then connect with the client. Since your CA is not in the system trust store, start by skipping certificate verification:

```
python3 tls/tls-client.py localhost -p 8443 --no-cert-verify
```

Answer:

- What TLS version, cipher suite, and ALPN protocol does the client report?
- What information is shown about the server certificate?

Now try **with** certificate verification:

```
python3 tls/tls-client.py localhost -p 8443
```

- What error do you get? Read the error message carefully.
- What would you need to do to make verification succeed?

> Hint
>
> The `ssl` module in Python loads the system trust store by default. Your self-signed CA is not in it. You could add it to the system store, or you could modify the client to load your CA certificate explicitly (`context.load_verify_locations("ca-cert.pem")`).

Inspect certificates of real web servers

The script `tls/show-cert.py` connects to any TLS server and displays its certificate chain in detail: subject, issuer, validity, signature algorithm, public key type and size, and Subject Alternative Names.

Try it on several real web servers:

```
python3 tls/show-cert.py www.computer-networking.info
python3 tls/show-cert.py www.belgium.be
python3 tls/show-cert.py www.elysee.fr
```

For each server, note:

- Who **issued** the certificate? (Which CA?)
- What **signature algorithm** and **public key type** are used?
- How many certificates are in the chain?
- What **Subject Alternative Names** are listed? Does the certificate cover more than one hostname?
- What is the validity period? How long is the certificate valid?

Compare the three servers:

- Do they all use the same CA?
- Do they all use the same key type and size?
- Do they all negotiate the same TLS version and cipher suite?

Capture TLS handshakes with real servers

The script `tls/capture-cert.sh` wraps `show-cert.py` with `tshark` to capture the TLS handshake and save the session keys for Wireshark decryption.

Capture a trace for each server:

```
cd tls
sudo ./capture-cert.sh www.computer-networking.info
sudo ./capture-cert.sh www.belgium.be
sudo ./capture-cert.sh www.elysee.fr
```

This produces, for each server, a `.pcapng` file with the TLS session keys embedded — for example `tls-www-computer-networking-info.pcapng`. Open it directly in Wireshark and the decrypted traffic is visible immediately:

```
wireshark tls-www-computer-networking-info.pcapng
```

(Similarly named files for the other two servers.)

For each trace, answer:

1. How many TCP packets does the TLS 1.3 handshake require before the first application data? Compare with a TLS 1.2 server if you can find one.
2. In the **ClientHello**, examine the `supported_versions` extension. What TLS versions does your client offer?
3. In the **ClientHello**, examine the `key_share` extension. What key exchange group(s) does the client propose?
4. In the **ServerHello**, what cipher suite did the server select? Is it the same for all three servers?
5. Find the **Certificate** message (it is encrypted in TLS 1.3, but the `.pcapng` file contains the keys so Wireshark decrypts it automatically). Does the certificate in the trace match what `show-cert.py` displayed?

> Note
>
> If you cannot run `tshark` yourself, pre-captured traces are available in the `tls/traces/` directory. The `.pcapng` files have the TLS session keys embedded — just open them directly in Wireshark and the decrypted traffic is visible immediately. Plain `.pcap` and `.keys` files are also provided. To regenerate all traces:
>
> ```
> cd tls
> sudo ./collect-traces.sh
> ```

Extract TLS keys with curl

The previous exercises used Python scripts that support `--keylog` to export TLS session keys. Many other TLS clients support the same mechanism through the `SSLKEYLOGFILE` environment variable: when this variable is set, the client writes the per-session keys to the specified file in the format defined by NSS. Both `curl` and web browsers (Firefox, Chromium) honour this variable.

Step 1 — Capture the traffic while fetching a page with `curl`:

```
sudo tshark -i any -p -w tls-curl.pcap -f "tcp port 443" &
sleep 2

SSLKEYLOGFILE=tls-curl.keys curl -s -o /dev/null https://www.computer-networking.info
sleep 1

sudo kill %1
```

This produces two files:

- `tls-curl.pcap` — the packet capture
- `tls-curl.keys` — the TLS session keys

You can open the capture in Wireshark and configure the key log file manually (as in the previous exercise), or you can embed the keys directly into the capture file.

Step 2 — Use `editcap` to create a self-contained `.pcapng` file with the TLS keys embedded:

```
editcap --inject-secrets tls,tls-curl.keys tls-curl.pcap tls-curl.pcapng
```

The resulting `tls-curl.pcapng` can be opened in Wireshark and the decrypted TLS traffic is visible immediately, without any manual configuration:

```
wireshark tls-curl.pcapng
```

Answer:

- Compare the TLS parameters (version, cipher suite, key exchange group) negotiated by `curl` with those observed using `show-cert.py` earlier. Are they the same?
- Examine the decrypted HTTP request. What `User-Agent` does `curl` send?

SSH

SSH (Secure Shell, RFC 4253) provides encrypted remote login and command execution. Like TLS, it uses asymmetric cryptography for authentication and key exchange, then switches to symmetric encryption for the data transfer. However, SSH has its own protocol and does **not** use X.509 certificates — instead, it relies on **host keys** and **authorized keys**.

In this part, you will set up a local SSH server using the Python `paramiko` library, generate keys, connect with the standard `ssh` client, and then inspect the negotiated algorithms with a custom client.

> Note
>
> All scripts are in the `ssh/` directory. The SSH servers listen on **localhost port 2222** to avoid conflicting with the system SSH daemon (port 22).

Generate a server host key

An SSH server needs a **host key** — an asymmetric key pair that identifies the server. The client uses the host key to verify that it is connecting to the right server (and not an impostor).

The script `ssh/generate_rsakey.py` generates an RSA key pair using paramiko:

```
cd ssh
python3 generate_rsakey.py --bits 2048 --out server_rsa.key
```

This creates `server_rsa.key` (the private key, permissions 600).

Answer:

- What is the key size in bits?
- What is the fingerprint printed by the script?
- Why are the file permissions set to 600 (owner read/write only)?

Start the SSH server (password authentication)

The script `ssh/simple-server.py` is a minimal SSH server that:

- Listens on `localhost:2222`
- Accepts one connection
- Authenticates with username `student` and password `password`
- Sends a greeting message and exits

Read the source code of `simple-server.py`, then start the server:

```
python3 simple-server.py
```

In the source code, identify:

- How is the host key loaded?
- What class handles authentication? What method checks the password?
- What happens after the client connects and authenticates?

Connect with the standard ssh client

With the server running, open a second terminal and connect using the standard `ssh` command:

```
ssh -o StrictHostKeyChecking=no -p 2222 student@localhost
```

Enter the password `password` when prompted.

Answer:

- What message does the server send?
- What warning does `ssh` display about the host key? Why?
- The `-o StrictHostKeyChecking=no` option disables host key verification. What would happen without it on the first connection?

> ### Hint
>
> After the first connection, SSH stores the server's host key in `~/.ssh/known_hosts`. If you regenerate the server key and connect again, `ssh` will warn you about a **host key mismatch** — this is an important security feature that detects man-in-the-middle attacks.

Inspect SSH algorithms with the protocol client

The standard `ssh` command does not show the negotiated algorithms by default. The script `ssh/simple-client.py` connects to an SSH server and prints all the protocol parameters: key exchange algorithm, host key type, ciphers, MACs, and compression.

Start the server again (it exits after one connection):

```
python3 simple-server.py
```

In a second terminal, run the inspection client:

```
python3 simple-client.py localhost -p 2222
```

Answer:

- What **key exchange algorithm** was negotiated?
- What **host key algorithm** does the server use?
- What **cipher** is used in each direction (client→server and server→client)? Are they the same?
- What **MAC algorithm** is used?
- What is the server's host key **fingerprint**? Does it match the fingerprint from the key generation exercise?

You can also try connecting to the system SSH daemon (if running):

```
python3 simple-client.py localhost
```

- Are the negotiated algorithms different? Why might they be?

Public key authentication

Password authentication is simple but has weaknesses: the password can be guessed or intercepted. SSH also supports **public key authentication**, where the client proves its identity using a private key, and the server checks it against a list of authorized public keys.

Step 1 — Generate a client key pair using `ssh-keygen`:

```
ssh-keygen -t rsa -b 2048 -f my_ssh_key -N ""
```

This creates two files:

- `my_ssh_key` — the private key (keep this secret)
- `my_ssh_key.pub` — the public key (give this to the server)

Step 2 — Create an `authorized_keys` file for the server. This file lists the public keys that are allowed to log in:

```
cp my_ssh_key.pub authorized_keys
```

Step 3 — Start the public key server. The script `ssh/pubkey-server.py` only accepts public key authentication:

```
python3 pubkey-server.py
```

Read the source code. Compared to `simple-server.py`:

- What method handles public key authentication instead of password?
- How does the server decide whether to accept a key?

Step 4 — Connect using your key:

```
ssh -o StrictHostKeyChecking=no -i my_ssh_key -p 2222 student@localhost
```

Answer:

- Does the connection succeed without asking for a password?
- On the server side, what key type and fingerprint are logged?
- What happens if you try to connect with a different key (generate a second key pair and try)?

Capture and analyze SSH traffic

SSH traffic is encrypted, so you cannot read the payload in Wireshark. However, the initial **key exchange** is visible in cleartext, and Wireshark can parse SSH protocol messages.

Start the server:

```
python3 simple-server.py
```

In a second terminal, start a capture:

```
sudo tshark -i lo -p -w ssh-handshake.pcap -f "tcp port 2222"
```

In a third terminal, connect:

```
ssh -o StrictHostKeyChecking=no -p 2222 student@localhost
```

After the connection closes, stop `tshark` with `Ctrl-C` and open the capture:

```
wireshark ssh-handshake.pcap
```

Useful Wireshark display filters for SSH

Filter	Description
`ssh`	Show all SSH packets
`ssh.protocol`	SSH version exchange
`ssh.message_code == 20`	Key Exchange Init (KEXINIT)
`ssh.message_code == 21`	New Keys
`tcp.port == 2222`	All traffic to/from the SSH server

In the Wireshark trace, answer:

1. Find the **version exchange** at the beginning. What SSH protocol version and software string does the client announce? The server?
2. Find the **KEXINIT** messages. Each side sends a list of supported algorithms. What key exchange algorithms, ciphers, and MACs does the client propose? What does the server propose?
3. After the key exchange, all subsequent packets are **encrypted**. Can you read any payload after the "New Keys" message?
4. How many packets does the SSH handshake take before the encrypted session begins? Compare with the TLS 1.3 handshake captured earlier.
5. Unlike TLS, SSH does not use certificates. How does the client know it is talking to the right server? What is the role of the `known_hosts` file?