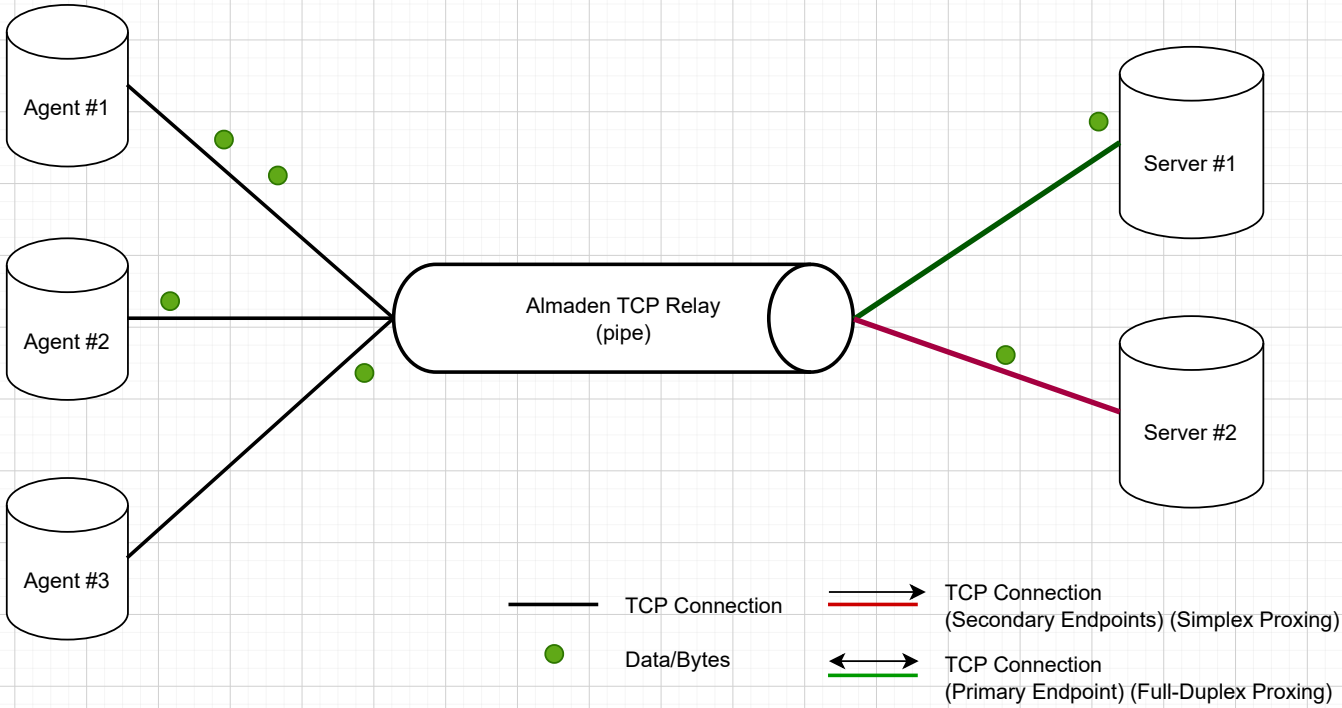
Almaden TCP Relay

Gabriel Henrique Hoppe Coelho

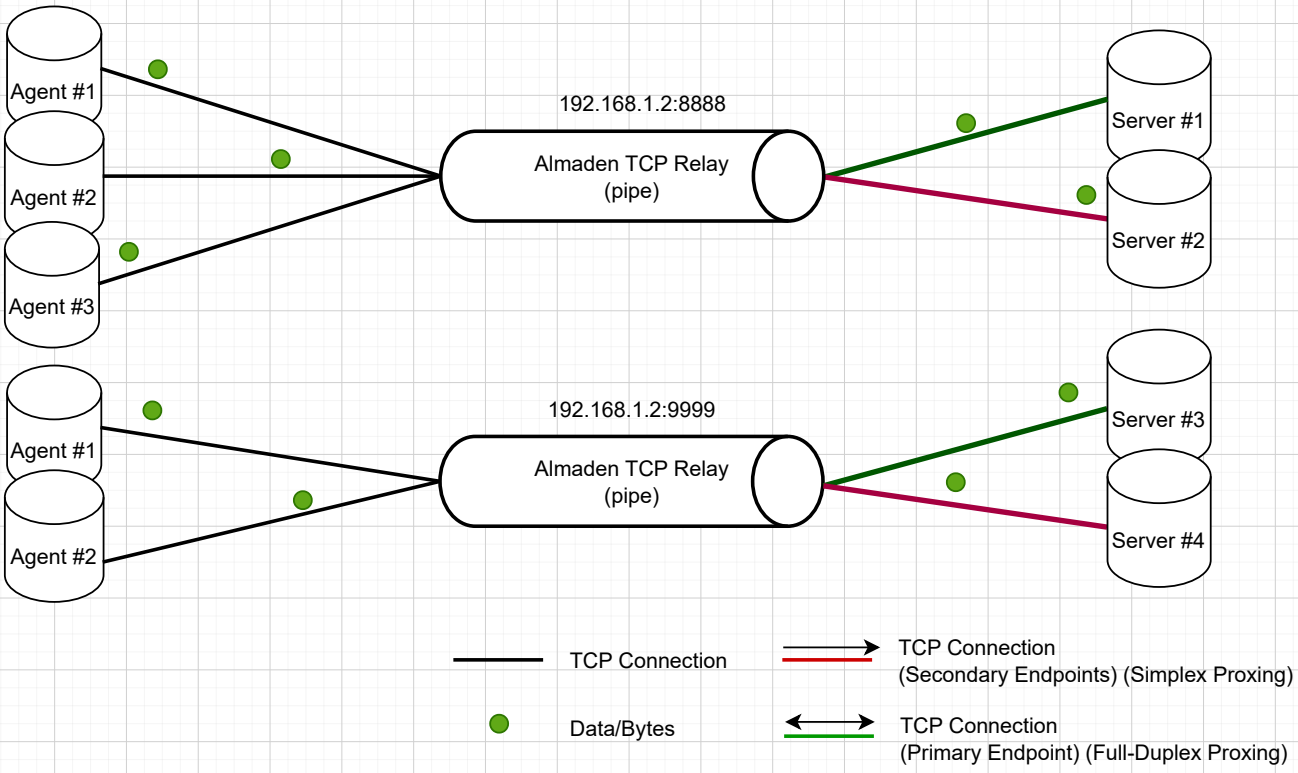
Summary	

About the Almaden TCP Relay

Almaden TCP Relay is a full-duplex TCP proxy, which means that any agent can send data through it using any protocol (HTTP, HTTPS, TLS/SSL, WebSocket, etc.).

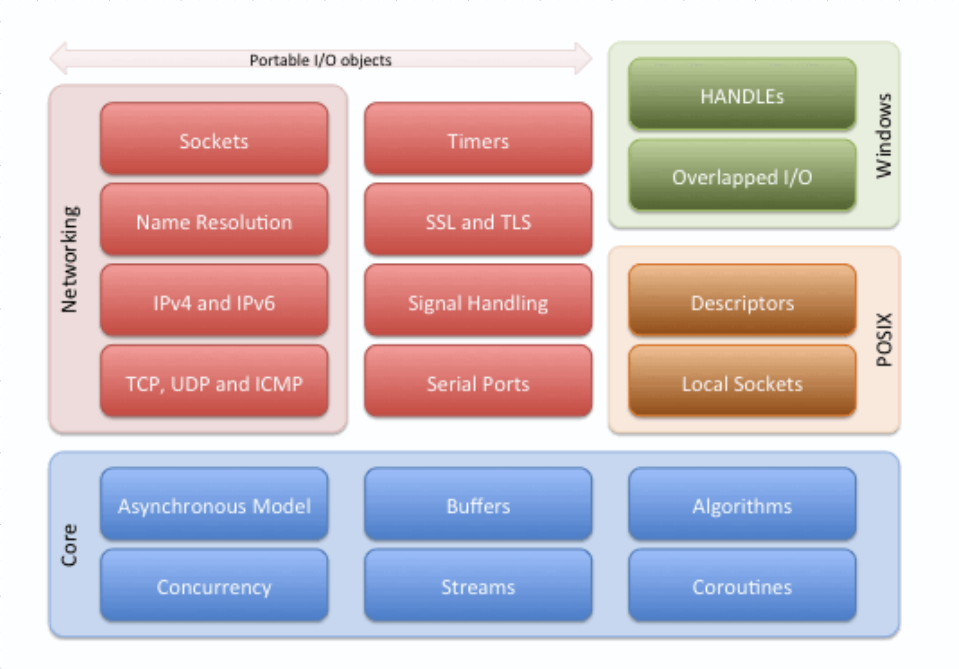


The relay is full-duplex and supports configuring secondary endpoints, making it possible to forward data from agents to multiple servers. Also, it is already possible to configure multiple pipes, as shown in the flowchart below.



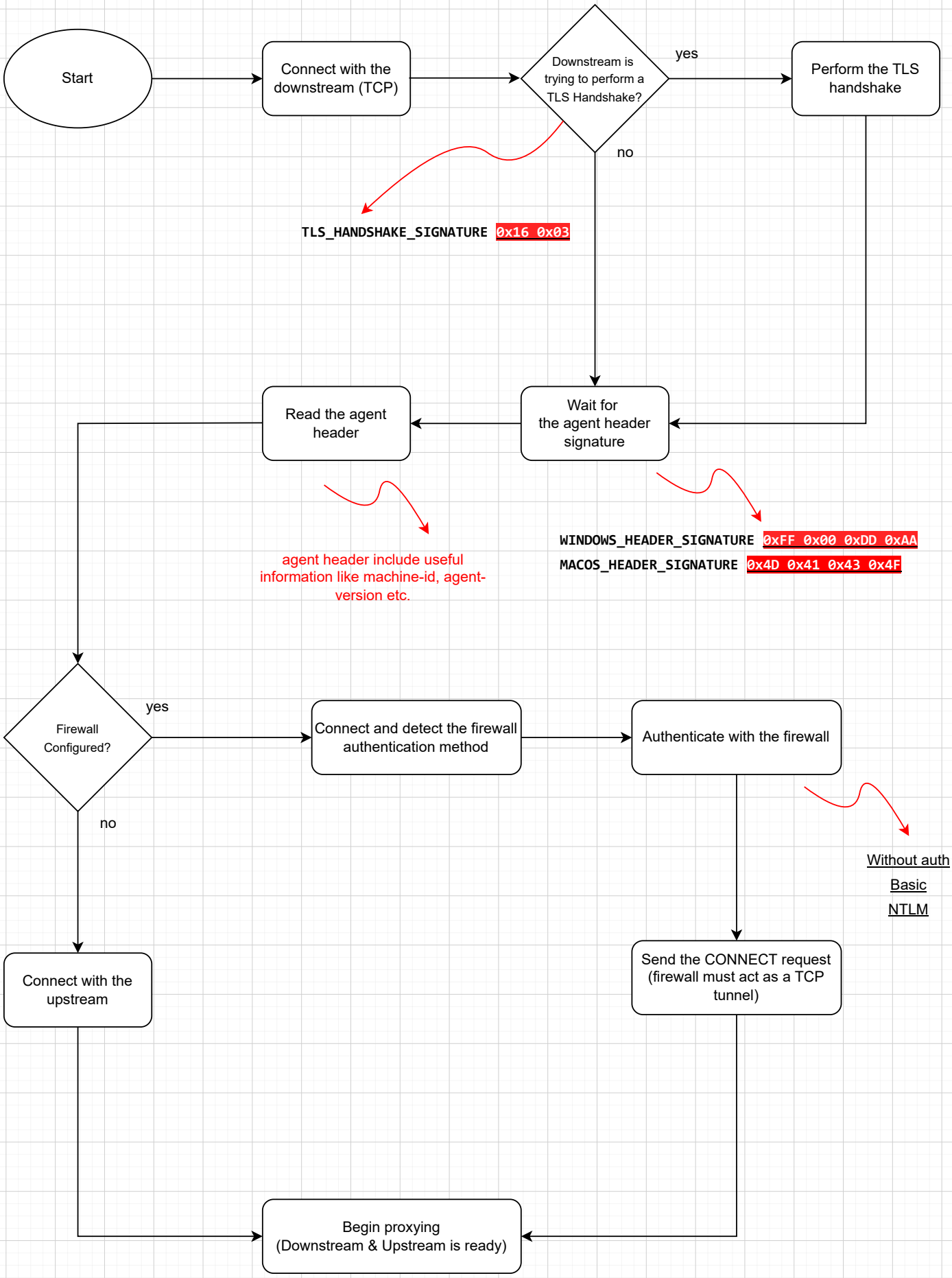
Technologies

In order to create the full duplex TCP proxy, we'll use the Asio C/C++ library (https://think-async.com/Asio/), this library is responsible to give us portable i/o objects.



We also will use the CMake to compile the code for both Windows and Linux operating systems.

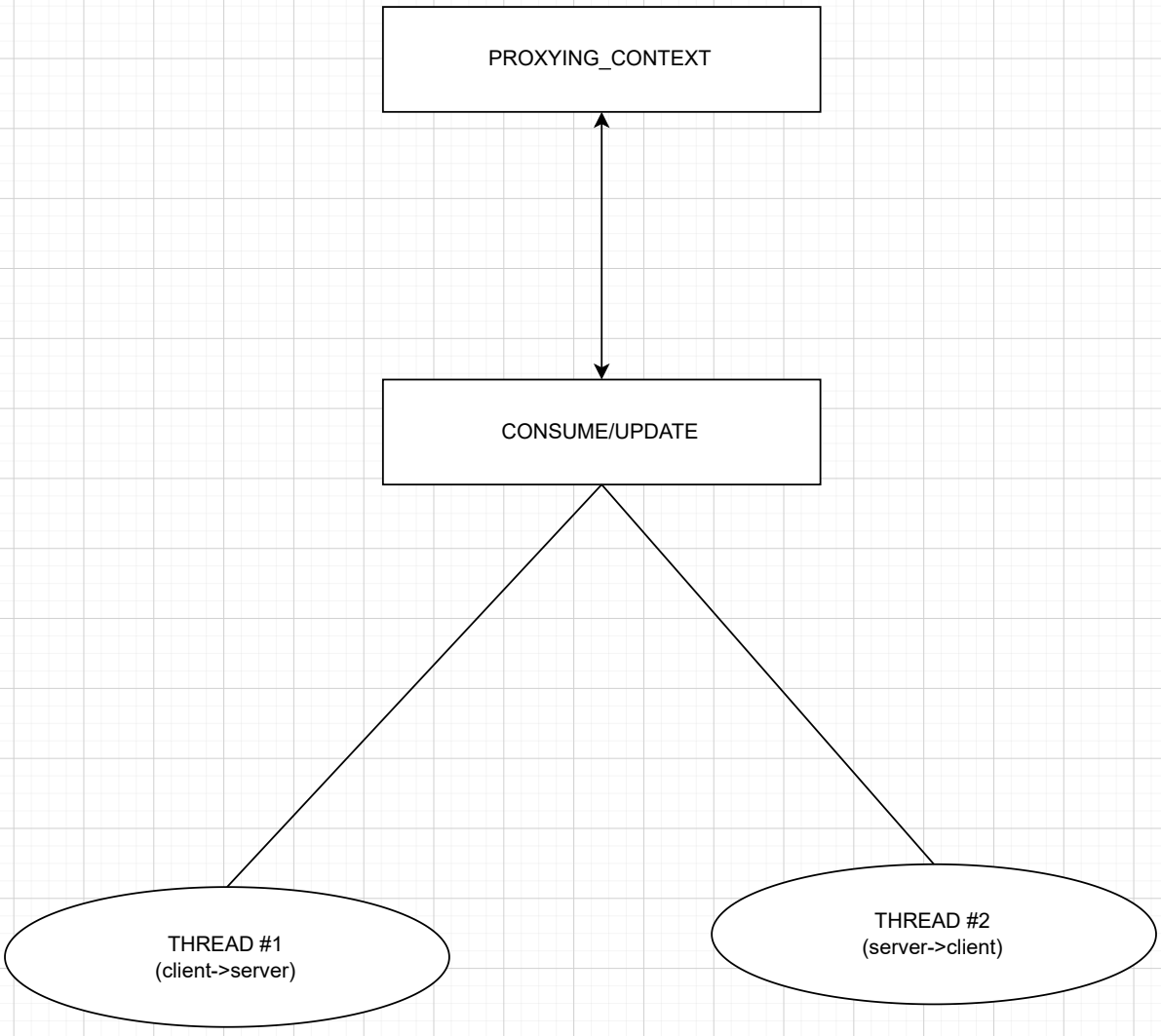
TCP Proxying Flow



Proxying Context

As previously mentioned, our proxy operates on a dual-threaded architecture. The first thread handles **upstream** traffic (reading data from the client and forwarding it to the server), while the second thread manages **downstream** responses (reading from the server and relaying data back to the client).

A key requirement is the exchange of metadata between threads, such as total bytes read from the client versus those received from the server. To facilitate this, we utilize a **shared proxy context pointer**, allowing both threads to access and update session metrics in real-time.



Proxying Context Structure

Proxying Context

downstream_socket	socket handle
upstream_socket	socket handle
downstream_buffer	array<1024*8>
upstream_buffer	array<1024*8>
bytes_read_from_downstream	size_t
total_bytes_read_from_downstream	size_t
bytes_read_from_upstream	size_t
total_bytes_read_from_upstream	size_t
bytes_written_to_downstream	size_t
total_bytes_written_to_downstream	size_t
bytes_written_to_upstream	size_t
total_bytes_written_to_upstream	size_t

HTTP Context

chunk_read_counter	socket handle
chunk_written_counter	socket handle
streaming_flag	array<1024*8>
streaming_finish_flag	array<1024*8>
streaming_finish_flag_state	size_t
client_header_size	size_t
client_body_length	size_t
client_body_counter	size_t

Proxying Context Fields

Network Handles

- **downstream_socket**: The active connection between the client and the relay.
- **upstream_socket**: The connection between the relay and the target server (primary endpoint).

Data Buffers

- **downstream_buffer**: Used to store data received from the client before it is forwarded to the server.
- **upstream_buffer**: Used to store data received from the server before it is relayed to the client.

Transmission Metrics

To maintain accurate tracking of the data flow, the following metrics are recorded:

Variable	Description
bytes_read_from_downstream	Bytes received from the client during the most recent read operation.
total_bytes_read_from_downstream	Cumulative bytes received from the client over the connection's lifetime.
bytes_read_from_upstream	Bytes received from the server during the most recent read operation.
total_bytes_read_from_upstream	Cumulative bytes received from the server over the connection's lifetime.
bytes_written_to_downstream	Bytes successfully transmitted to the client in the last write operation.
total_bytes_written_to_downstream	Cumulative bytes transmitted to the client over the connection's lifetime.
bytes_written_to_upstream	Bytes successfully transmitted to the server in the last write operation.
total_bytes_written_to_upstream	Cumulative bytes transmitted to the server over the connection's lifetime.

HTTP State Management

These fields are used for parsing and controlling the HTTP protocol within the context of proxying.

State & Control Flags

- **`streaming_flag`**:
  A boolean indicator that is set to `true` if the HTTP request utilizes **chunked transfer encoding** (`Transfer-Encoding: chunked`).
- **`streaming_finish_flag`**:
  A flag indicating that the **chunked-stream terminator** (`0\r\n\r\n`) has been successfully identified.
- **`streaming_finish_flag_state`**:
  A state tracker used by the search algorithm to persist the current matching position while detecting the streaming terminator across multiple packets.

Counters & Metrics

- **`chunk_read_counter`**:
  Tracks the number of chunks processed in the current request. This value is **reset** at the start of every new HTTP request.
- **`client_header_size`**:
  The size, in bytes, of the current HTTP request headers.
- **`client_body_length`**:
  The expected total size of the request body, parsed from the `Content-Length` header.
- **`client_body_counter`**:
  A running total of bytes read from the HTTP body so far, used to determine when the payload is complete.

TCP Interceptors

RFC (Read-From-Client Interceptor)

This interceptor read and processes data sent by the client to update the proxying context with relevant metadata.

Required Parameters:

- `proxying_context`
- `callback_function`

WTS (Write-To-Client Interceptor)

This interceptor can manipulate data read from the server before sending to the client.

Required Parameters:

- `proxying_context`
- `callback_function`

RFS (Read-From-Server Interceptor)

This interceptor read and processes data sent by the server to update the proxying context with relevant metadata.

Required Parameters:

- `proxying_context`
- `callback_function`

WTS (Write-To-Server Interceptor)

This interceptor can manipulate data read from the client before sending to the server.

Required Parameters:

- `proxying_context`
- `callback_function`

WTSS (Write-To-Secondary-Server Interceptor)

This interceptor can manipulate data read from the client before sending to the server.

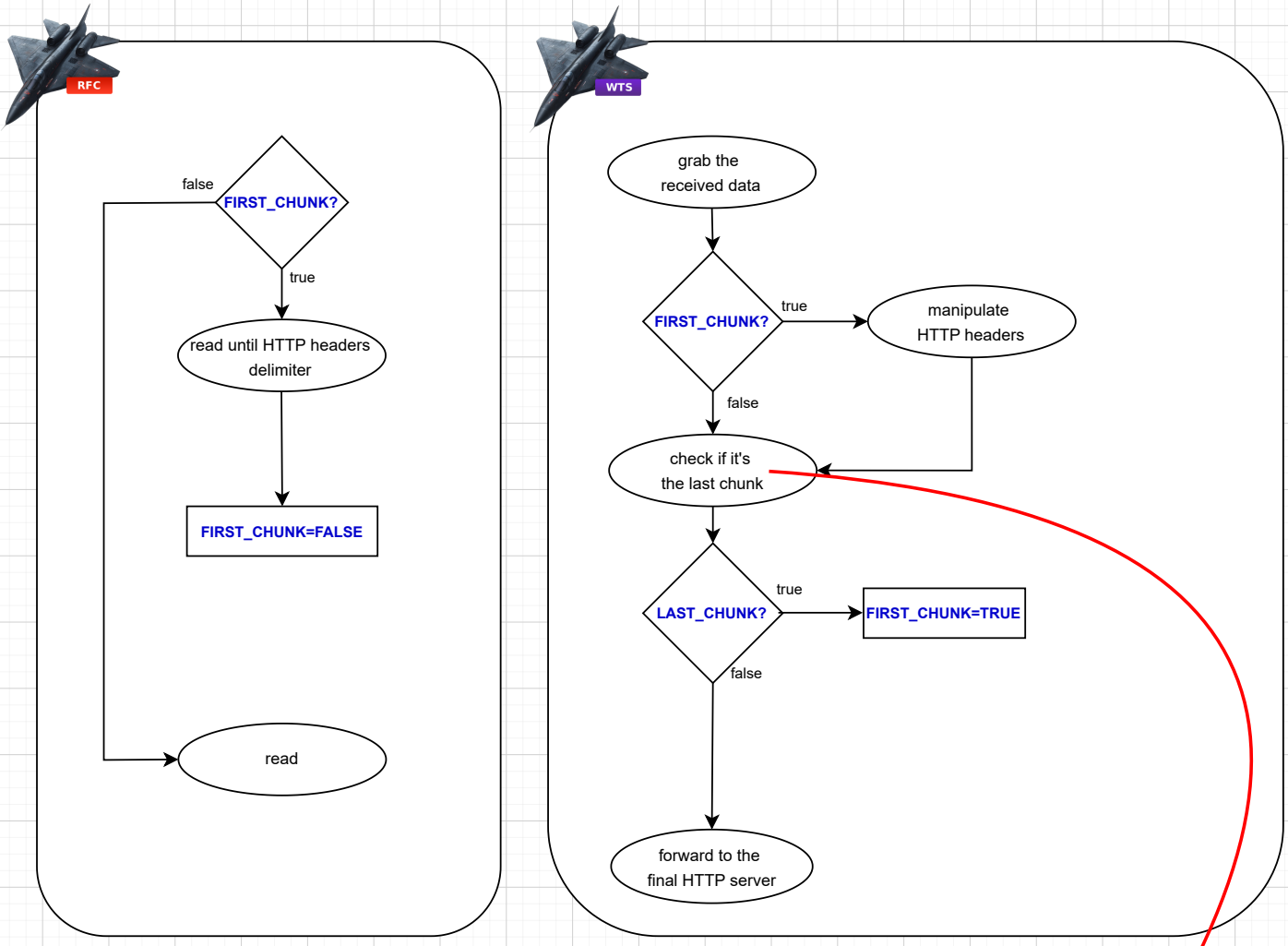
Required Parameters:

- `proxying_context`
- `secondary_endpoint_session (secondary server socket handle)`
- `callback_function`

HTTP 1.1 Proxying Flow

To successfully proxy HTTP/1.1 traffic, a proxy server must execute the following sequence:

1. **Buffer and Parse Headers:** Read the incoming stream until the header delimiter—the Double Carriage Return Line Feed sequence (`\r\n\r\n` or `0x0D 0x0A 0x0D 0x0A`)—is reached.
2. **Modify HTTP Headers:** Perform necessary header transformations. This typically includes updating the `Host` header to match the destination server and appending client information to headers like `X-Forwarded-For`.
3. **Forward Modified Headers:** Transmit the newly constructed header block to the destination endpoint.
4. **Stream the Request Body:** Forward the payload (body) from the client to the destination.
5. **Relay the Response:** Receive the response from the final endpoint and relay it back to the original client.
6. **Manage Connection Persistence:** If `Connection: keep-alive` is present (or as per the default behavior in HTTP/1.1), maintain the socket connection and repeat this process for subsequent requests on the same channel.



The logic for determining when the chunk is the last one varies depending on the transfer method used by the client:

1. Fixed-Length Termination (`Content-Length`)

When a `Content-Length` header is present, the proxy knows the exact size of the payload in advance.

- **Logic:** The proxy must track the cumulative total of bytes received.
- **Example:** If `Content-Length: 32`, the proxy stops reading once exactly 32 bytes of body data have been processed.

2. Streamed Termination (`Transfer-Encoding: chunked`)

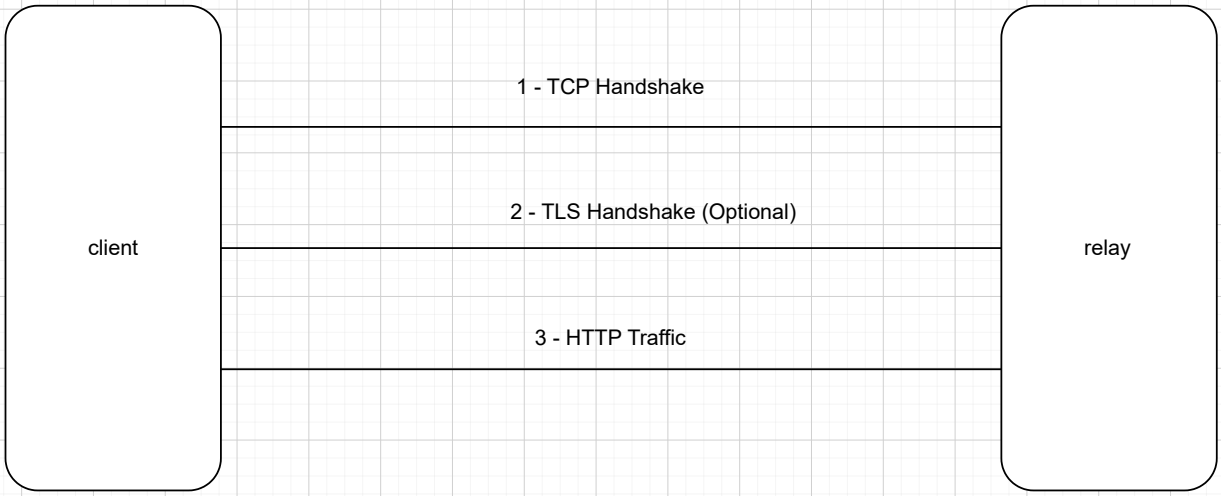
If the client is streaming data where the total size is unknown, it uses **Chunked Transfer Encoding**.

- **Completion:** The proxy must watch for the "Last-Chunk" signal, which is a zero-sized chunk. This is explicitly identified by the sequence `0\r\n\r\n` (the digit zero followed by two sets of CRLF).

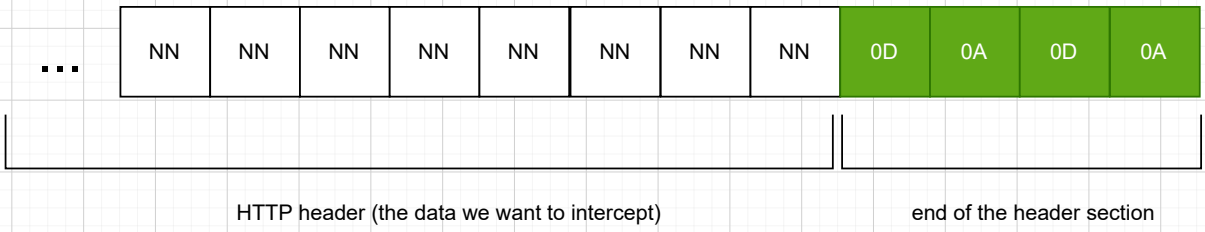
HTTP 1.1 RFC Interceptor

To ensure correct HTTP/HTTPS proxying and avoid security rejections (such as 403 Forbidden), the following headers must be managed:

- **Host**: Must be updated to the **final destination address**. This ensures the backend server recognizes the request for its own domain.
- **X-Forwarded-For**: Must contain the **original client's IP address**. This allows the backend to identify the real source of the traffic.
- **X-Forwarded-Proto**: Must be set to **`https`** if the client-to-proxy connection is encrypted (TLS). This prevents unnecessary redirect loops at the backend.
- **X-Forwarded-Host**: Must be the **original Host requested by the client** (typically the proxy's address). This is useful for the backend to reconstruct original URLs if needed.



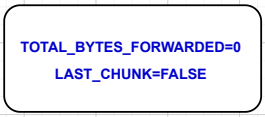
The best way to intercept HTTP headers is to read the traffic until you find the byte sequence that signals the end of the header section: **`\r\n\r\n`** (or **`0D 0A 0D 0A`** in hex).



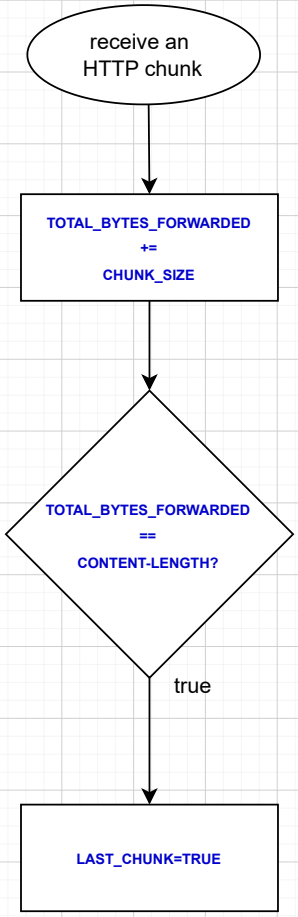
HTTP 1.1 WTS Interceptor (Fixed-Length Termination)

When the client **sends** the **Content-Length** in the HTTP header, this interceptor **is used** to check if the current chunk is the last part of the payload

GLOBAL VARIABLES

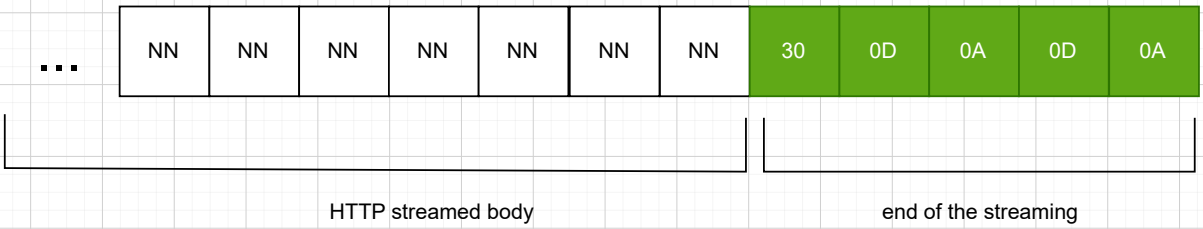


INTERCEPTOR FLOW



HTTP 1.1 WTS Interceptor (Streamed Termination) (Transfer-Encoding=chunked)

The best way to intercept an HTTP streamed body is to read the traffic until you find the byte sequence that signals the end of the streaming: 0**\r\n\r\n** (or **30 0D 0A 0D 0A** in hex).



GLOBAL VARIABLES

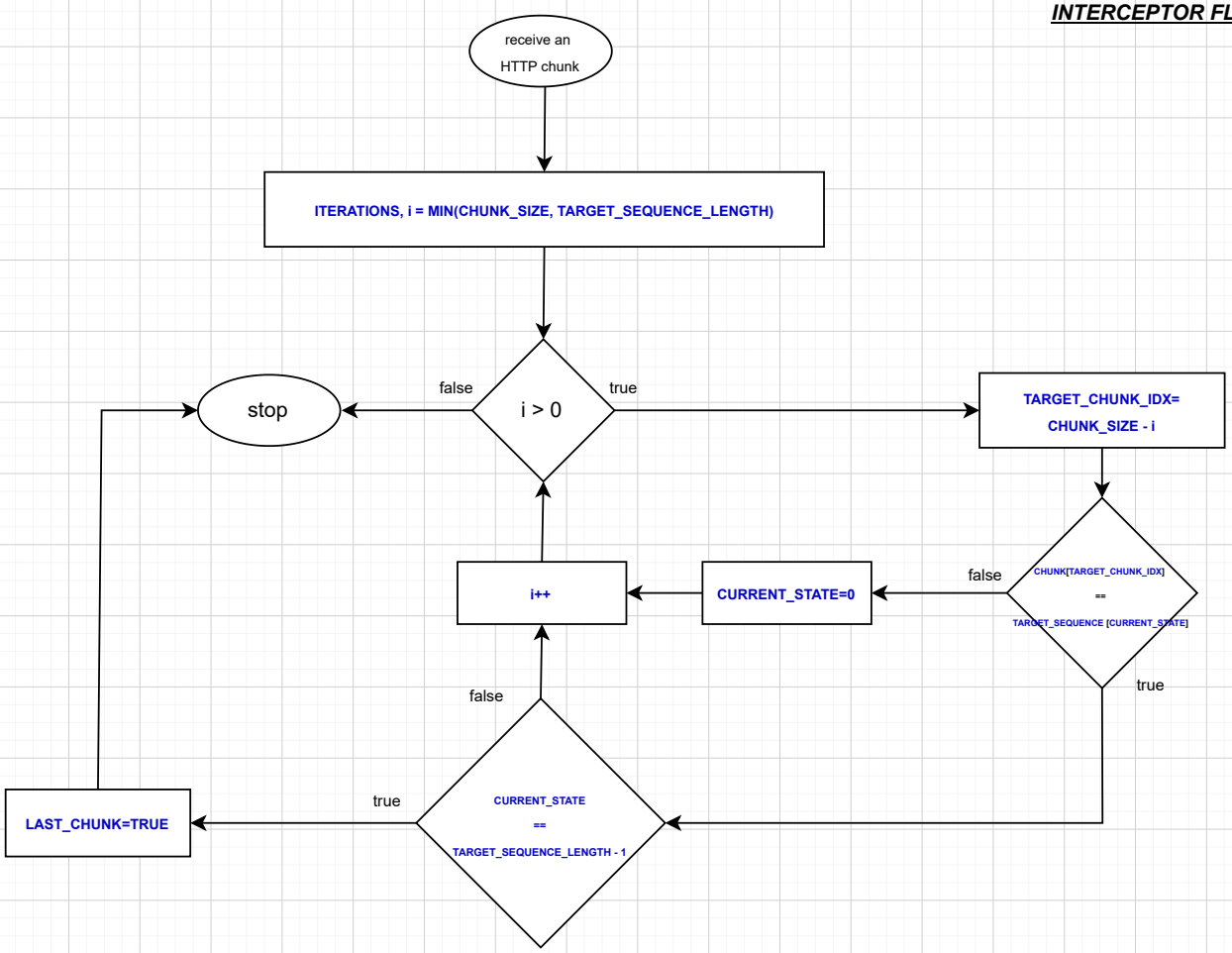
CONTENT_LENGTH=X
LAST_CHUNK=FALSE

TARGET_SEQUENCE =
{ 0x00, 0x0D, 0x0A, 0x0D, 0x0A }

CURRENT_STATE=0

the sequence can come split in different chunks; that's why we need a special algorithm

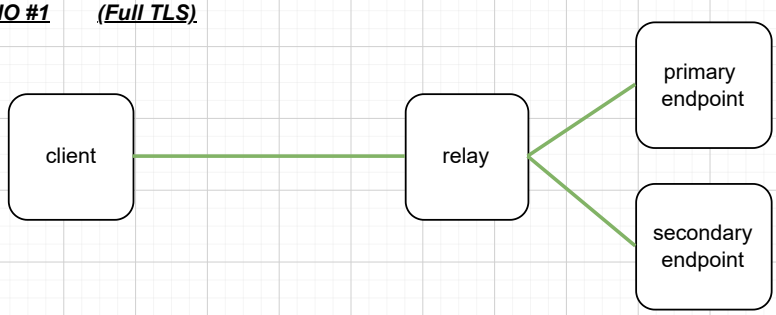
INTERCEPTOR FLOW



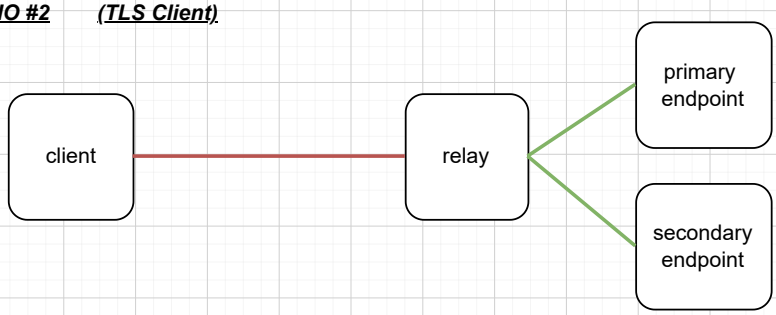
TLS Support

The relay supports TLS encryption for connections with the client, the final endpoint, and all secondary endpoints.

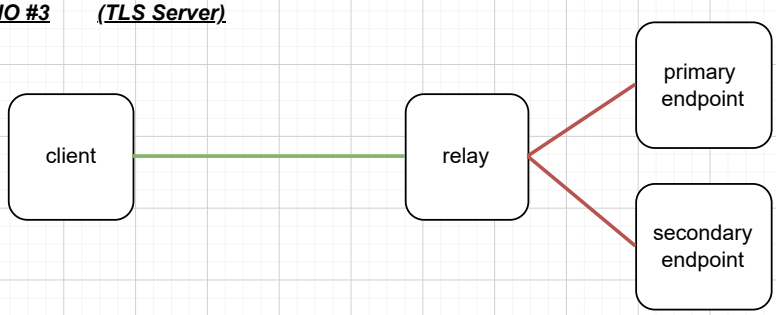
SCENARIO #1 ***(Full TLS)***



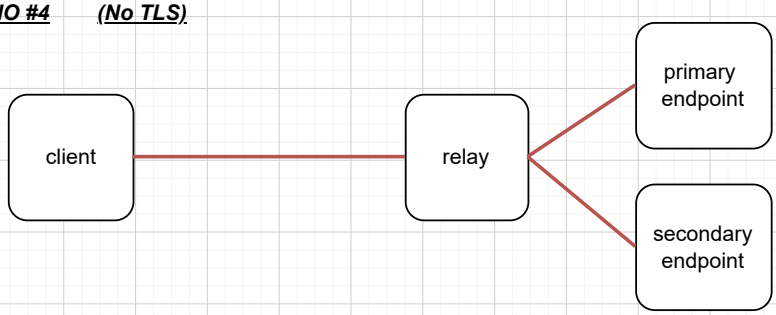
SCENARIO #2 ***(TLS Client)***



SCENARIO #3 ***(TLS Server)***



SCENARIO #4 ***(No TLS)***



TCP + TLS communication.

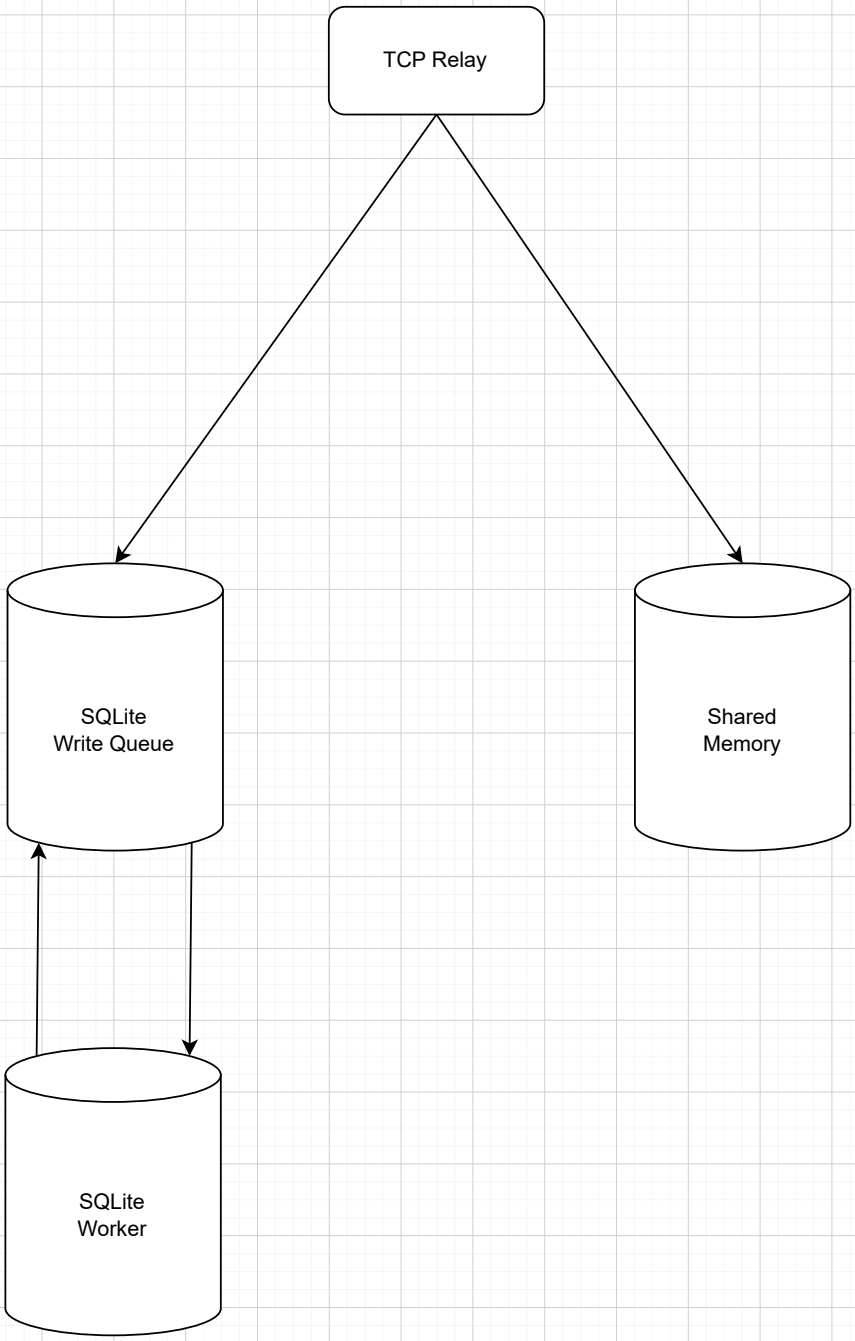
TCP only communication.

Telemetry

The relay must monitor and report connection counts and traffic data. This information will be stored in **shared memory** for real-time access across the Linux CLI and Windows UI. For long-term data persistence, the relay will utilize an **SQLite database.**

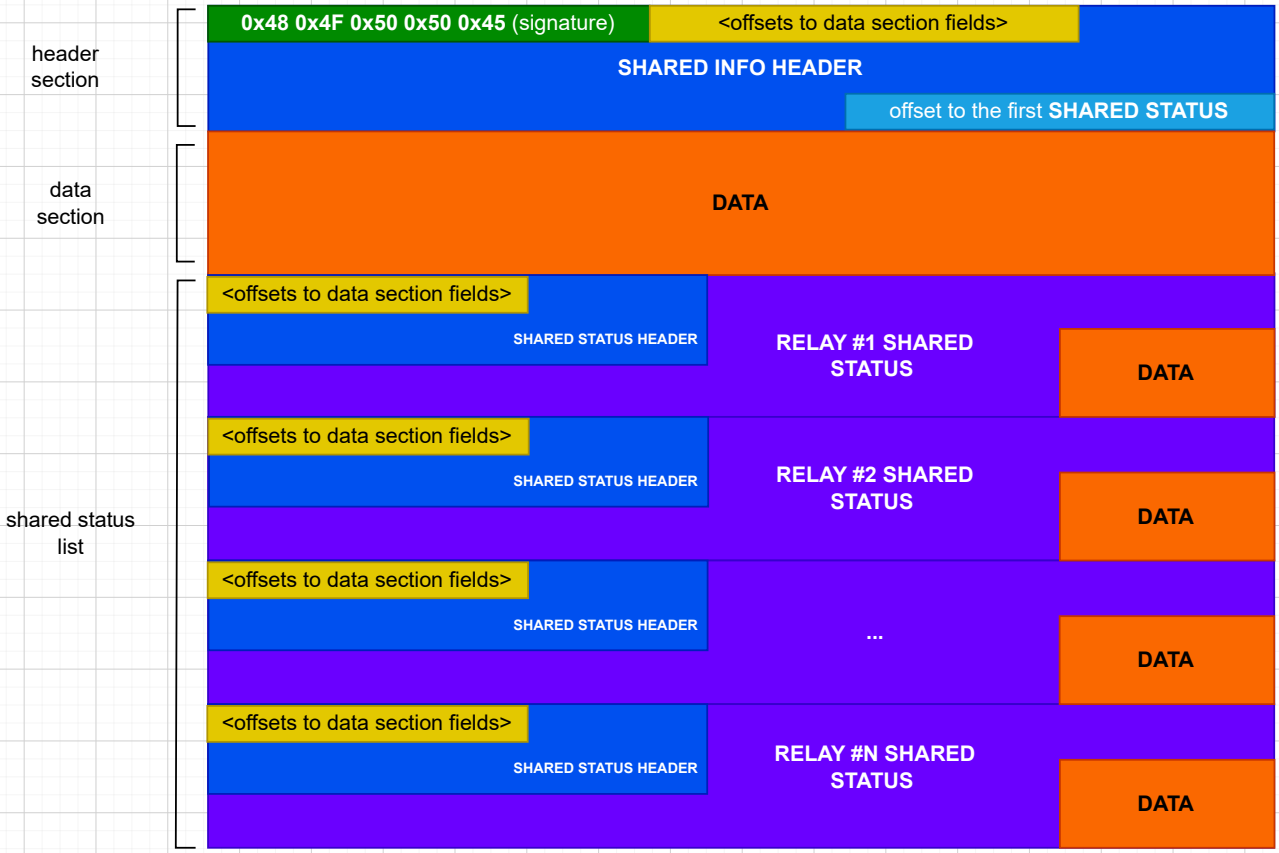
Because disk I/O is significantly slower than memory operations, writing to SQLite directly inside the proxying flow would create a performance bottleneck.

- **The Pipeline:** The relay pushes data packets into a **FIFO (First-In-First-Out) Queue**.
- **Asynchronous Processing:** A background worker thread consumes this queue, batching or executing writes to the SQLite database as the disk allows.
- **Benefit:** Protects the relay's core performance from disk "stutter" or high-latency spikes, ensuring traffic continues to flow even during heavy database indexing.



Shared Info

The shared status is a struct stored in the shared memory region, allowing other processes to access this information in real time.



The data section is designed to store variable-sized structures. To ensure backward compatibility and prevent breaking changes, the header contains pointers to these structures. This architecture allows different processes to interact without issues, even if the structure sizes evolve.

Note: Relay shared statuses also include a data section.

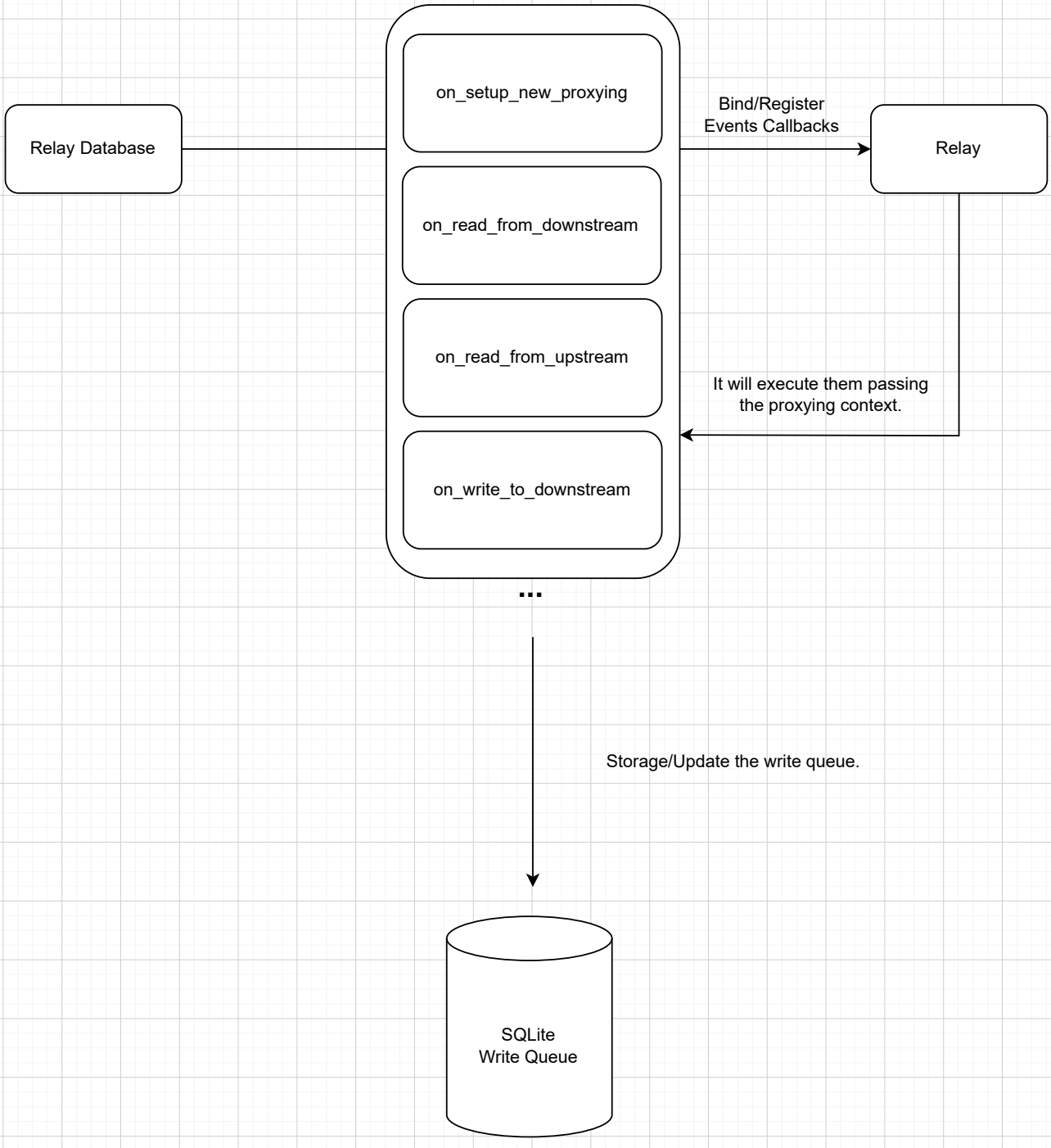
PS: The maximum number of relays is 32!

Relay Database

The **Relay Database** is a dedicated entity responsible for persisting essential data within a local SQLite instance. This component is vital for monitoring relay performance and tracking system activity.

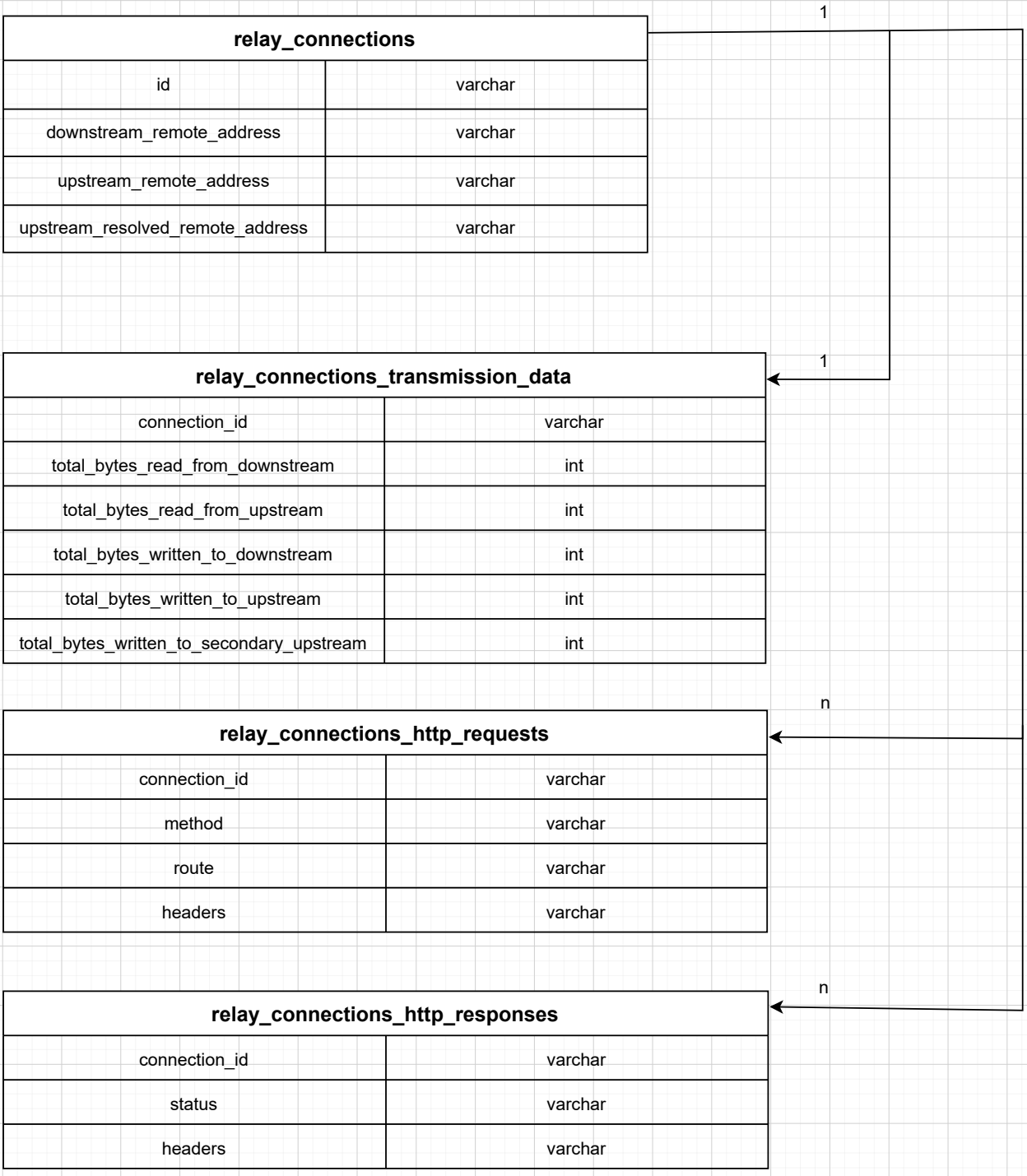
To capture this data, the database utilizes the relay's **bind/unbind mechanism**. By subscribing to specific callback functions, the Relay Database automatically logs relevant operational information as events occur.

Note: All relay callback functions receive a `ProxyingContext` object as a primary argument.



Relay Database Schema

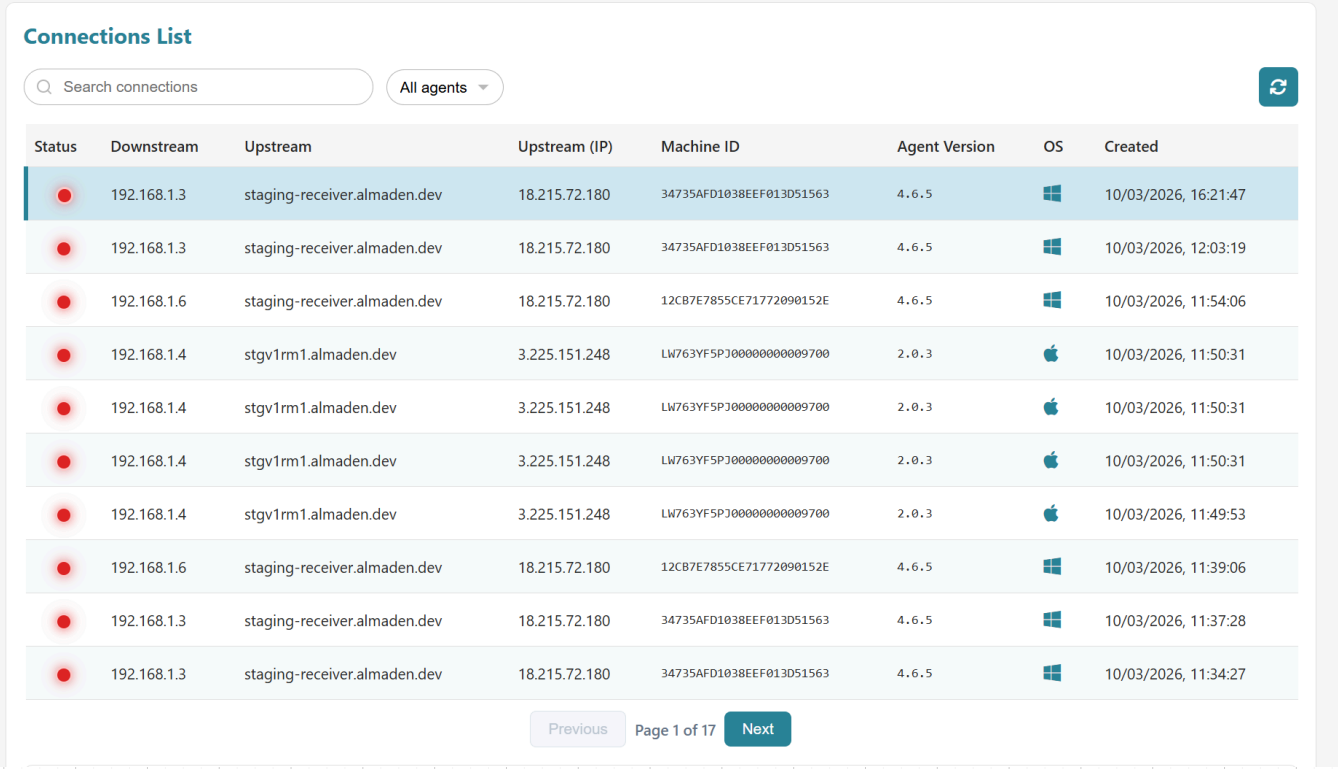
The database will have the following tables:



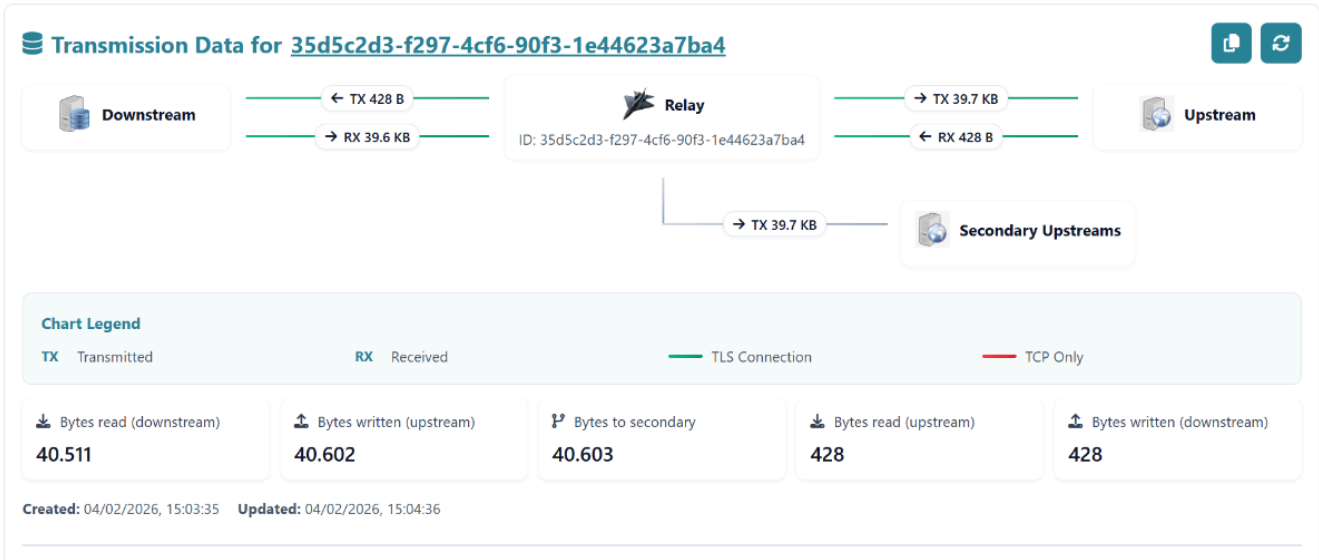
Relay UI

The Relay UI will provide valuable insights, such as a connection list and detailed data for each entry, including transfer rates and HTTP requests.

Connections List

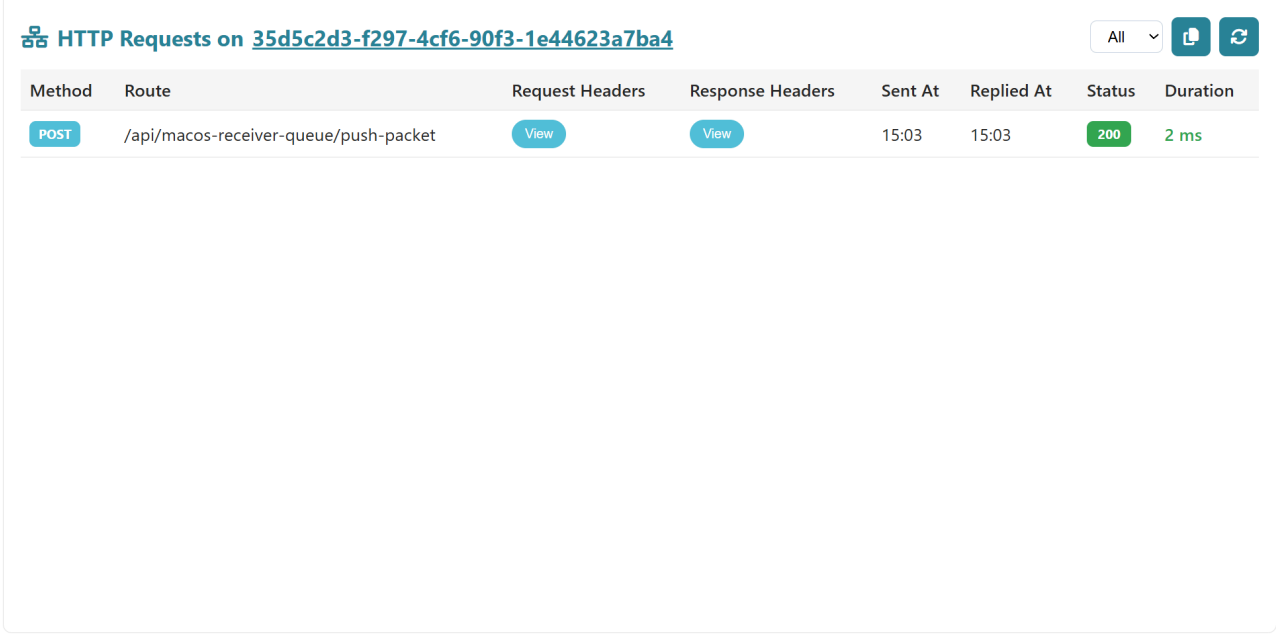


Transmission Data

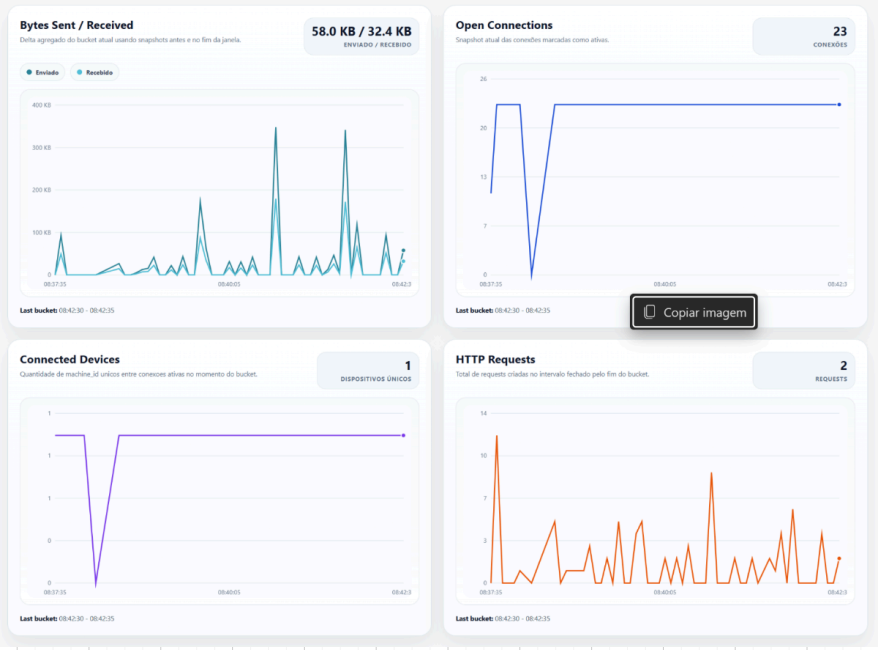


Relay UI

HTTP Requests



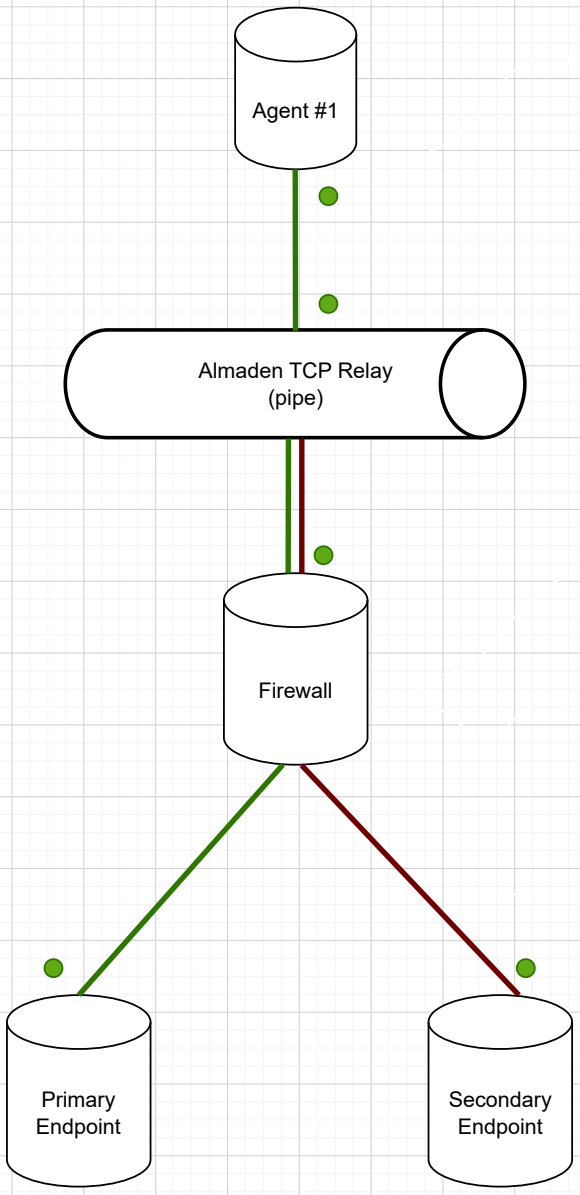
Monitor Counters



Firewalls

It is absolutely possible to use a relay even when there is a firewall in the way to the fiinal endpoint.

As long as the necessary ports and protocols required for the relay and authentication flow are allowed or can be tunneled through, the firewall does not inherently prevent the relay from working. In many cases, relay attacks operate over standard services (such as HTTP, SMB, or LDAP), which are often permitted by default or can be proxied, making this scenario entirely feasible.



1 - Before sending traffic through the relay the agent send a package informing what's the machine-id, operating system, agent-version etc.

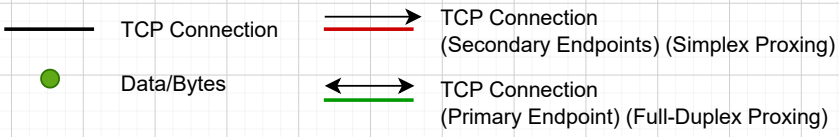
2 - The relay grab all this info.

3 - Before proxying any data, the relay detects the firewall authentication method (Basic or NTLM) and attempts to authenticate using the provided credentials.

4 - After a successful authentication, the relay sends a CONNECT request to the firewall so it can act as a TCP tunnel.

5 - The agent can now send data through the relay, and this traffic will be forwarded to the primary or secondary endpoints.

If split tunneling is enabled, the relay determines the appropriate endpoint, as the destination may vary depending on the agent's operating system.



Supported Firewall Authentication Methods

Authentication Methods		
Auth	**Available**	**Handshake Negotiation**
without authentication	Yes	N.A
Basic (username and password)	Yes	N.A
NTLMv2	Yes	NEGOTIATE_UNICODE NEGOTIATE_NTLM NEGOTIATE_ALWAYS_SIGN EXTENDED_SESSIONSECURITY NEGOTIATE_VERSION REQUEST_TARGET NEGOTIATE_128

Compiling for Windows

1 - Clone the vcpkg repository in C:\ location.

```
cd C:\
git clone https://github.com/microsoft/vcpkg.git
cd vcpkg
```

2 - Initialize the vcpkg.

```
.\bootstrap-vcpkg.bat
```

3 - Install the the ASIO library (https://think-async.com/Asio/).

```
.\vcpkg install boost-asio
./vcpkg install openssl:x64-windows
./vcpkg install sqlite3
```

4 - Clone the almaden-tcp-relay repository code.

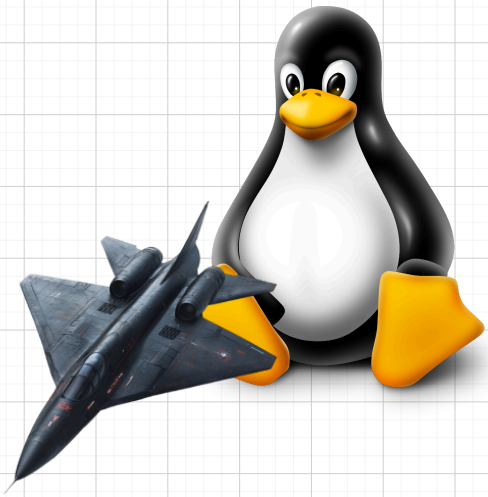
3 - Generate build files

```
cmake -B build -S . -DCMAKE_TOOLCHAIN_FILE=C:/vcpkg/scripts/buildsystems/vcpkg.cmake
```

4 - Build

```
cmake --build build --config Release
```

Compiling for Linux



1 - Clone the almaden-tcp-relay repository code.

2 - Install the ASIO library (https://think-async.com/Asio/).

```
sudo apt update
sudo apt install libboost-all-dev
sudo apt install libsqlite3-dev
```

3 - Generate build files.

```
cmake -B build -S .
```

4 - Build.

```
cmake --build build --config Release
```

Legacy Configuration

(WINDOWS) HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Automatos\Asa\Relay\3.0	
Key	**Type**
tcp_relay_port	REG_DWORD
windows_linux_receiver_address	REG_SZ
windows_linux_receiver_port	REG_DWORD
macos_receiver_address	REG_SZ
macos_receiver_port	REG_DWORD
firewall_type	REG_SZ ('explicit' or 'transparent')
firewall_address	REG_SZ
firewall_port	REG_DWORD
firewall_username	REG_SZ
firewall_password	REG_SZ
firewall_domain	REG_SZ

(LINUX) /opt/automatos/ada/etc/arelay.cfg	
Key	**Type**
TCP_RELAY_PORT	int
WINDOWS_LINUX_RECEIVER_ADDRESS	string
WINDOWS_LINUX_RECEIVER_PORT	int
MACOS_RECEIVER_ADDRESS	string
MACOS_RECEIVER_PORT	int
FIREWALL_TYPE	string ('explicit' or 'transparent')
FIREWALL_ADDRESS	string
FIREWALL_PORT	int
FIREWALL_USERNAME	string
FIREWALL_PASSWORD	string
FIREWALL_DOMAIN	string

Configuration File Locations

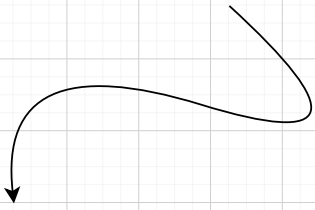
Location	
Windows	**$LOCALAPPDATA**/Almaden/tcp-relay-conf.json
Linux	/etc/almaden/tcp-relay-conf.json

Configuration File Diagram View

json	
relays	**relay_settings[]**

relay_settings (split-tunneling disabled)	
tls	bool
tls_security_level	int
tls_legacy_renegotiation	bool
split_tunneling	**FALSE**
app_layer_protocol	http or none
local_port	int
primary_endpoint	endpoint
secondary_endpoints	**endpoint[]**

relay_settings (split-tunneling enabled)	
tls	bool
tls_security_level	int
tls_legacy_renegotiation	bool
split_tunneling	**TRUE**
win/linux_app_layer_protocol	http or none
macos_app_layer_protocol	http or none
local_port	int
primary_endpoint	endpoint
secondary_endpoints	**endpoint[]**



endpoint (split-tunneling-disabled)	
tls	bool
remote_address	string
remote_port	int

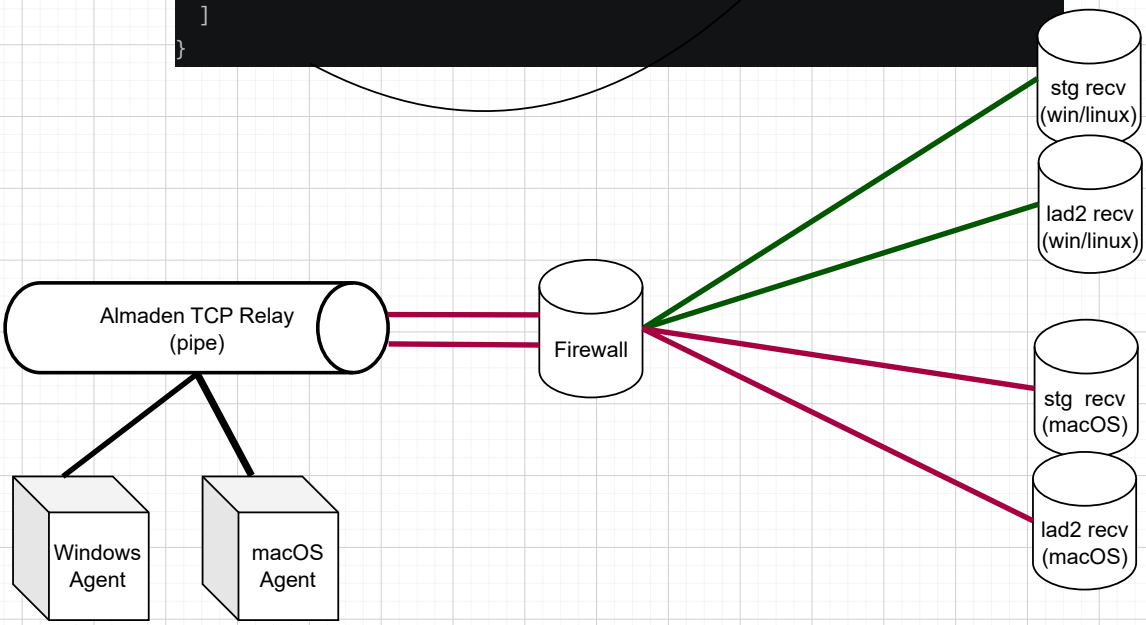
endpoint (split-tunneling-enabled)	
tls	bool
win/linux_remote_address	string
macos_remote_address	string
win/linux_remote_port	int
macos_remote_port	int

Configuration File Example

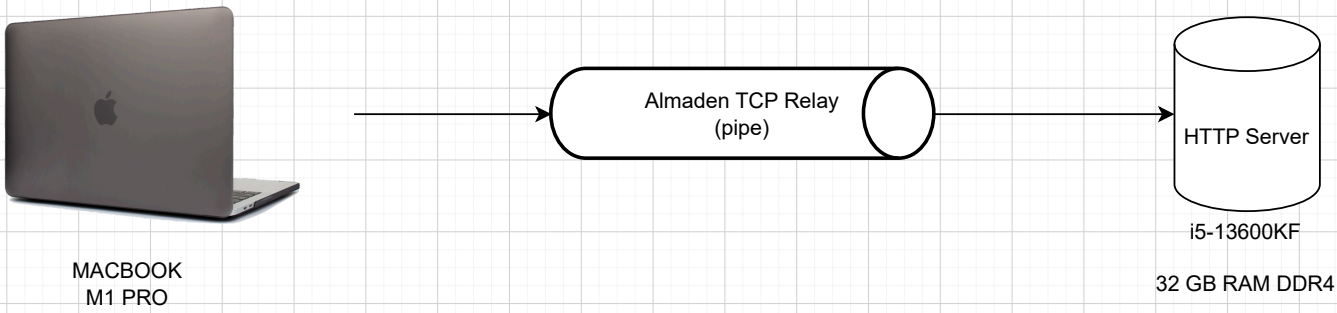
```
{
  "relays": [
    {
      "local_port": 8888,
      "tls": true,
      "tls_security_level": 0,
      "tls_legacy_renegotiation": true,
      "split_tunneling": true,
      "windows_and_linux_app_layer_protocol": "none",
      "macos_app_layer_protocol": "http",
      "primary_endpoint": {
        "tls": true,
        "windows_and_linux_remote_address": "staging-receiver.almaden.dev",
        "windows_and_linux_remote_port": 443,
        "macos_remote_address": "stgv1rm1.almaden.dev",
        "macos_remote_port": 443
      },
      "secondary_endpoints": [
        {
          "tls": true,
          "windows_and_linux_remote_address": "lad2-receiver.almaden.app",
          "windows_and_linux_remote_port": 443,
          "macos_remote_address": "lad2v1rm1.almaden.app",
          "macos_remote_port": 443
        }
      ],
      "firewall": {
        "type": "proxy",
        "firewall_address": "127.0.0.1",
        "firewall_port": 3128,
        "firewall_username": "gabrielh2c",
        "firewall_password": "Poulow1479$",
        "firewall_domain": "WORKGROUP"
      }
    }
  ]
}
```

This configuration creates split tunneling for Windows and macOS agents, with both agents reporting to two separate receivers at the same time.

And all traffic passes through an NTLM firewall.



Load Testing (Local HTTP Server)

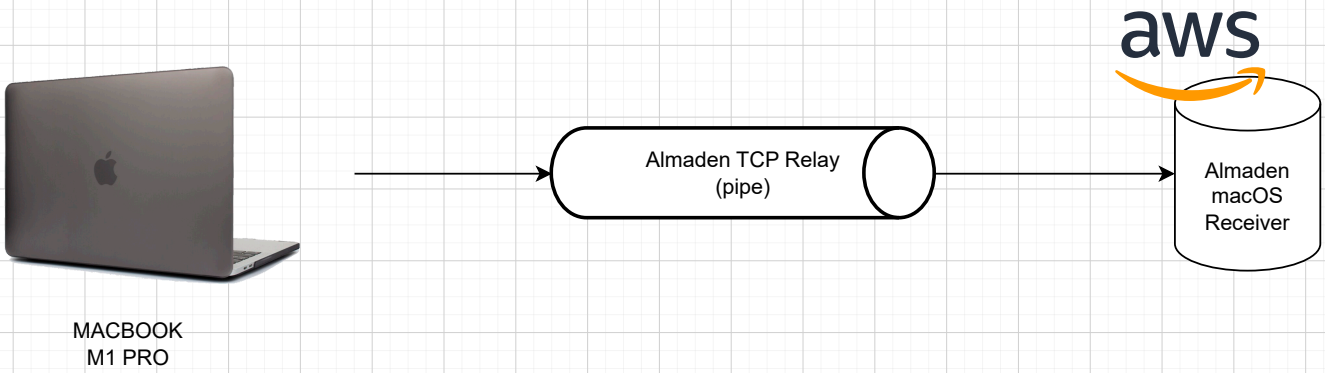


PS: Endpoint is an HTTP Server created using NodeJS and Fastify

Without the Almaden TCP Relay (Windows)			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	10.111	14.399	9.640
Bytes/Sec	1.72 MB	2.45 MB	1.64 MB
Using the Almaden TCP Relay (Windows)			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	9.335 (-7.67%)	13.623 (-5.38%)	9.426 (-2.21%)
Bytes/Sec	1.59 MB (-7.55%)	2.32 MB (-5.30%)	1.6 MB (2.43%)

Without the Almaden TCP Relay (Linux)			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	12.143	17.055	11.832
Bytes/Sec	2.06 MB	2.90 MB	2.01 MB
Using the Almaden TCP Relay (Linux)			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	11.447 (-5.73%)	18.351 (+7.06%)	12.077 (+2.02%)
Bytes/Sec	1.95 MB	3.12 MB	2.05 MB

Load Testing (Almaden HTTP Receiver)

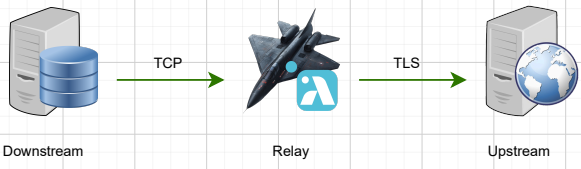


PS: Endpoint is the Almaden macOS HTTP Receiver

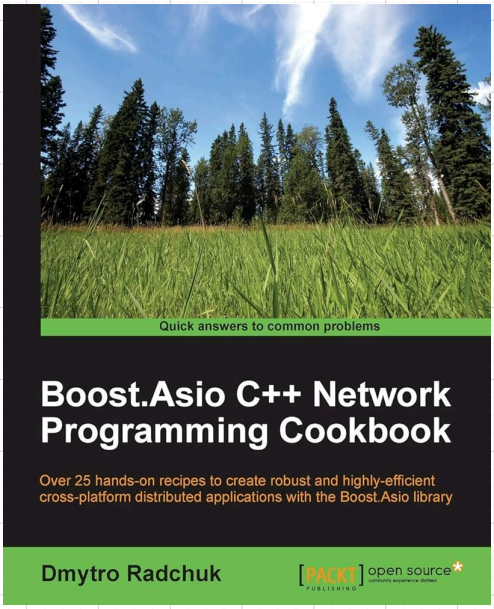
Without the Almaden TCP Relay			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	687	776	670
Bytes/Sec	199 kB	225 kB	192 kB
Using the Almaden TCP Relay (Full TLS)			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	682 (-0,7%)	696 (-10,30%)	668 (-0,2%)
Bytes/Sec	198 kB (-0,5%)	202 kB (-10,22%)	194 kB (+1,04%)
Using the Almaden TCP Relay (TCP/TLS)			
Stat	**50%**	**97.5%**	**Avg**
Req/Sec	690 (+0,04%)	724 (-7,18%)	680 (+1,49%)
Bytes/Sec	200 kB (+0,5%)	210 kB (-7,14%)	197 kB (+2,60%)

Full TLS:

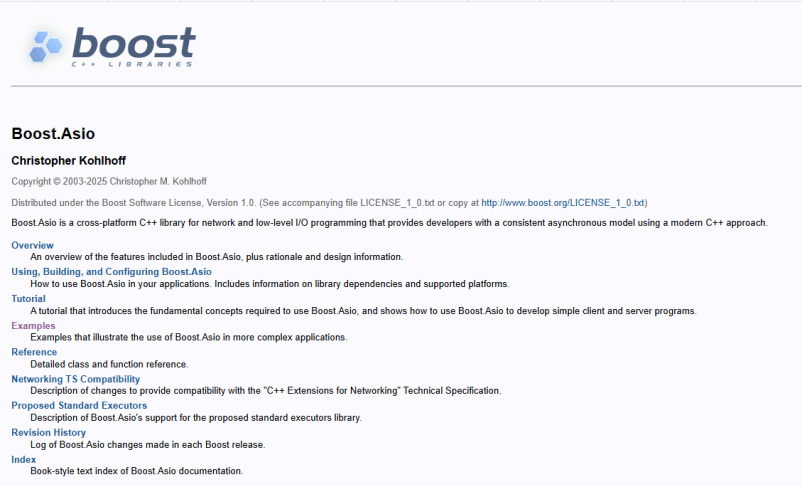
TCP/TLS:



References



Boost.Asio C++ Network Programming Cookbook: Over 25 hands-on recipes to create robust and highly-efficient cross-platform distributed applications with the Boost.Asio library.



Boost.Asio official documentation (https://think-async.com/Asio/boost_asio_1_36_0/doc/html/boost_asio.html)