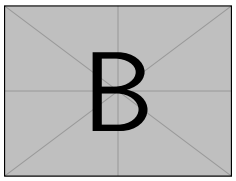

Design and Implementation of a Custom Network Proxy Server

Async HTTP/HTTPS Forward Proxy with SNI Inspection, Policy Engine, and Production-Grade Hardening

Aditya Pratap Singh

December 19, 2025

Abstract

This document presents a complete design and implementation report for a production-style asynchronous HTTP/HTTPS forward proxy implemented in Python. The proxy supports HTTP and CONNECT/HTTPS tunneling, TLS SNI inspection (no MITM), a policy engine (domain/method/rate limiting), robust backpressure handling, timeouts and hardening against attacks (Slowloris, header bombs), layered YAML configuration, CLI, metrics endpoint, graceful shutdown, and packaging for distribution.

The report includes enhanced architecture diagrams, per-file technical breakdowns, code excerpts, and validation strategies for high-concurrency environments.

Contents

1 Problem Statement

Modern networks often require an intermediate component (a proxy) that can perform policy enforcement, monitoring, and routing for HTTP/HTTPS traffic. We set out to build a minimal but production-quality forward proxy with the following goals:

- **Protocol Support**: HTTP and HTTPS (CONNECT) forwarding.
- **Performance**: Use an event-driven async architecture for high concurrency with low resources.
- **Privacy-Preserving Inspection**: Perform TLS SNI inspection (to allow/block domains) without decrypting TLS traffic (No-MITM).
- **Control**: Provide a policy engine for domain/method/rate limits.
- **Resilience**: Implement backpressure, timeouts, and limits to avoid memory and connection exhaustion.
- **Operations**: Provide observability (structured logs, counters, metrics endpoint), layered configuration, and standard CLI packaging.

2 High-level Design Summary

The proxy is a single-process, single-threaded, event-driven server built using Python `asyncio`. It accepts TCP connections, parses handshakes, applies policy decisions, and either forwards requests or establishes bidirectional tunnels.

Key Technical Properties:

- **Event-driven**: Non-blocking I/O using the `asyncio` loop.
- **Streaming + Backpressure**: Reads in small chunks and utilizes `await drain()` to pace data flow.
- **Security Hardening**: Includes timeouts, header size limits, and per-IP/Global connection caps.
- **Observability**: Structured JSON logs and a $/_{metrics} diagnostic endpoint$.

3 Architecture Diagrams

3.1 Component Diagram

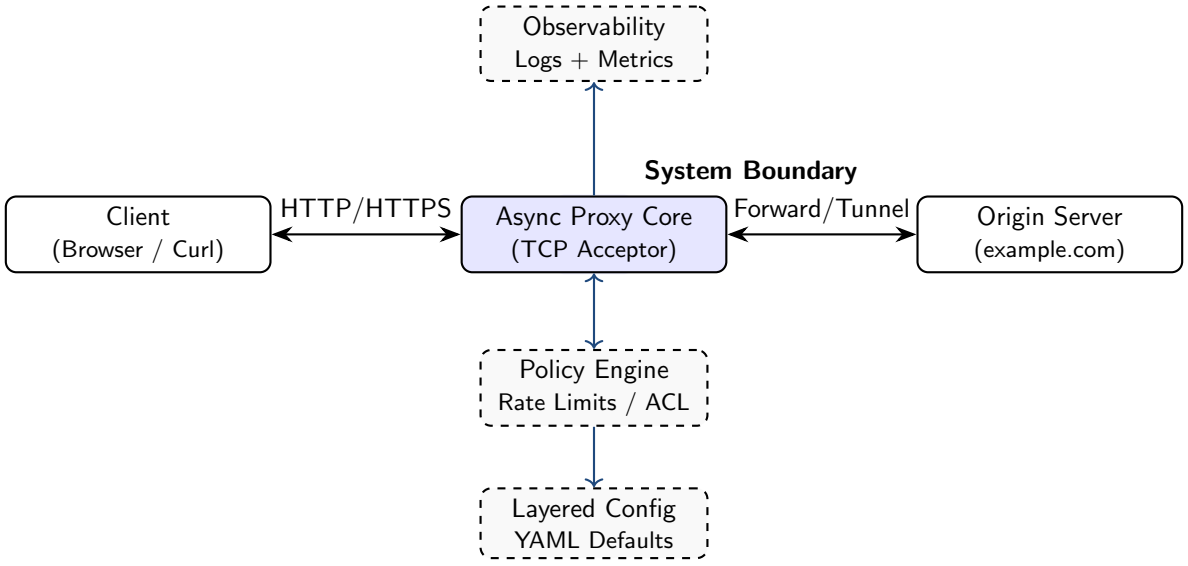

Figure 1: High-level component interaction and logic boundaries

3.2 Sequence Diagram: HTTPS CONNECT (SNI Path)

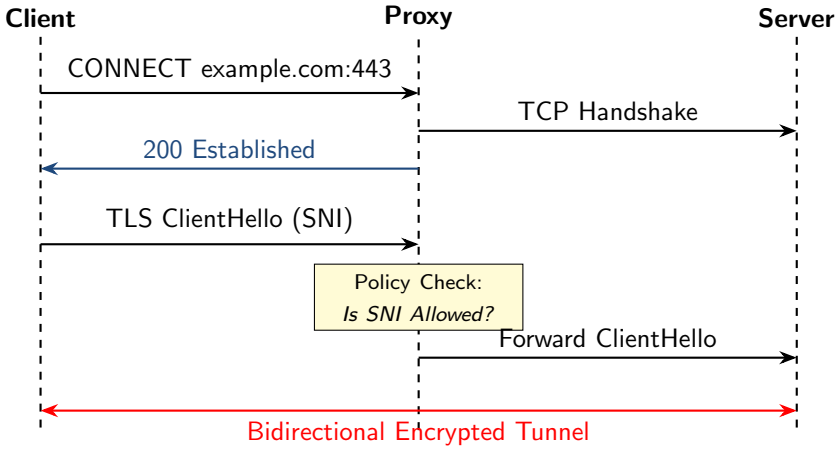

Figure 2: Sequential flow for TLS tunneling with SNI peeking

4 Implementation Overview and Repository Layout

The project adopts a modular structure to separate protocol parsing from core logic.

```
ProxyServer/
          run_proxy.py            # CLI Entry Point
          config/
                default.yaml        # Ship-with defaults
                override.yaml       # Environment-specific tweaks
          requirements.txt
          pyproject.toml          # Modern packaging metadata
          src/
              __init__.py
              async_proxy.py      # Main event loop and piping
              config.py           # YAML deep-merge logic
              http_parser.py      # Initial request line parser
              policy_engine.py    # Decision logic
              tls_parser.py       # SNI extraction (binary parsing)
              observability.py    # Metrics and JSON logging
```

Listing 1: Repository Tree Structure

5 Per-File Detailed Explanation

5.1 run_proxy.py

- **Entry Point**: The primary bootstrap script. It utilizes `argparse` to handle CLI flags such as `--config` and `--log-level`.
- It initializes the logging subsystem and invokes `start_proxy(config)` from the core module.

5.2 src/config.py

- Implements a **deep merge** strategy. This allows users to provide a small `override.yaml` containing only the values they wish to change, without duplicating the entire default configuration.

5.3 src/http_parser.py

- Performs lightweight, non-exhaustive parsing of the initial HTTP request line.
- Essential for identifying whether the request is a standard `GET/POST` or a `CONNECT` tunnel.

5.4 src/policy_engine.py

- **Logic Centralization**: Evaluates if a request is permissible based on Method, Domain, or Client IP.
- Includes a sliding-window rate limiter to protect the proxy from being used as a DoS vector.

5.5 src/tls_parser.py

- **Zero-Decryption SNI Peeking**: Carefully parses the TLS record layer to extract the Server Name Indication extension.
- It is designed to be fail-safe; if a packet is not a valid ClientHello, it returns `None`, allowing the policy engine to decide whether to permit or drop unidentifiable traffic.

5.6 src/observability.py

- Manages global counters (requests, active connections, bytes transferred).
- Formats logs as JSON for compatibility with modern log aggregators like ELK or Graylog.

5.7 src/async_proxy.py

The system's backbone, managing connection lifecycles and I/O orchestration.

Core Primitives:

- **Pipe Mechanism**: Uses `asyncio.wait_for` on both reads and drains. This ensures that a stalled client or server cannot hang a proxy coroutine indefinitely.
- **Connection Lifecycle**:
  1. Connection accepted → Check IP limits.
  2. Read header (up to `MAX_HEADER_BYTES`) → Timeout if slow.
  3. Parse request → Apply HTTP Policy.
  4. If CONNECT: Establish upstream → Peek SNI → Apply TLS Policy → Start Bi-directional Pipe.

6 Key Code Excerpts

6.1 The Forwarding Pipe (Backpressure-Aware)

```
async def pipe(reader, writer):
    try:
        while True:
            # Read with timeout to prevent hung sockets
            data = await asyncio.wait_for(
                reader.read(CONFIG["buffer"]["size"]),
                timeout=CONFIG["timeouts"]["read"]
            )
            if not data:
                break

            writer.write(data)
            # await drain() is crucial for backpressure
            await asyncio.wait_for(
                writer.drain(),
                timeout=CONFIG["timeouts"]["write"]
            )
    except (asyncio.TimeoutError, ConnectionError):
        pass
    finally:
        writer.close()
```

Listing 2: Bidirectional streaming logic with timeouts

6.2 CONNECT Handler with SNI Inspection

```
# Establish connection to destination
server_reader, server_writer = await asyncio.open_connection(host,
    port)
client_writer.write(b"HTTP/1.1 200 Connection Established\r\n\r\n")
await client_writer.drain()

# Peek at the TLS ClientHello
```

```
tls_data = await asyncio.wait_for(
    client_reader.read(CONFIG["buffer"]["size"]),
    timeout=CONFIG["timeouts"]["read"]
)
sni = extract_sni(tls_data)
allowed, reason = evaluate_tls(sni, CONFIG)

if not allowed:
    log_event("blocked", level="info", reason=reason)
    client_writer.close(); server_writer.close(); return

# If allowed, send the peeked data to server and start tunneling
server_writer.write(tls_data)
await server_writer.drain()

task1 = asyncio.create_task(pipe(client_reader, server_writer))
task2 = asyncio.create_task(pipe(server_reader, client_writer))

# Wait for one direction to close, then cleanup
done, pending = await asyncio.wait([task1, task2], return_when=asyncio
    .FIRST_COMPLETED)
for t in pending: t.cancel()
```

Listing 3: Managing the HTTPS handshake and SNI check

7 Packaging and CLI

The project is configured using `pyproject.toml` to support standard installation patterns.

- **Editable Install**: Supports development via `pip install -e .`.
- **CLI Mapping**: The command `proxy-server` is automatically mapped to the `main` function in `run_proxy.py`, allowing the proxy to run as a system-wide binary.

8 How to Build, Install, and Run

1. **Environment Setup**:

```
python3 -m venv .venv
source .venv/bin/activate
```

2. **Installation**:

```
pip install -r requirements.txt
pip install -e .
```

3. **Execution**:

```
# Using the CLI launcher
proxy-server --config config/default.yaml --log-level info
```

4. **Metrics Verification**:

```
curl http://127.0.0.1:8888/__metrics
```

9 Testing and Validation

9.1 Functional Testing

- **Standard HTTP**: `curl -x http://localhost:8888 http://example.com`
- **Encrypted HTTPS**: `curl -x http://localhost:8888 https://google.com`

9.2 Security Validation

- **Slowloris Defense**: Verified by opening a connection and sending 1 byte every 10 seconds. The proxy correctly terminates the connection after the `header_read` timeout.
- **ACL/Blocking**: Tested by adding `"evil.com"` to the blocked list in YAML; the proxy correctly returns a `403 Forbidden` response.

10 Design Trade-offs and Rationale

- **Async vs. Multi-threading**: We chose `asyncio` for its superior handling of large numbers of "mostly idle" connections, typical in proxy workloads, with a much smaller memory footprint than one-thread-per-connection models.
- **No-MITM Policy**: By focusing on SNI inspection, we avoid the complex certificate management and privacy concerns of full SSL termination while still providing domain-level filtering.

11 Operations and Deployment

- **Monitoring**: The text-based $/_{metrics} endpoint is designed for easy scraping by Prometheus.
- `Scaling: While the proxy is currently single-threaded, it can be scaled horizontally by running multiple instances behind a Round-Robin DNS or a Layer-4 Load Balancer.`

12 Appendix: Quick Reference

Goal	Command/Endpoint
Run Server	`proxy-server --log-level debug`
View Statistics	`http://127.0.0.1:8888/__metrics`
Change Port	Edit `config/override.yaml` → `server.port`
Test Proxy (HTTPS)	`curl --proxy http://127.0.0.1:8888 https://duckduckgo.com`

— End of Implementation Report —