UNIVERSITY OF TRANSPORT AND COMMUNICATIONS

FACULTY OF INTERNATIONAL EDUCATION

GRADUATION THESIS

TOPIC:

Building a secure proxy system for load balancing and cybersecurity

Lecturer	:	Lương Thái Lê
Student	:	Nguyễn Thành Hưng
Class	:	Information Technology Vietnamese-English 1
Student ID	:	212660183

Hà Nội – 2025

Table of Contents

Table of Figures

ACKNOWLEDGMENTS

During my studies at the University of Transport and Communications, I wish to thank all faculty whose support made my learning and research possible. I am particularly grateful to the instructors of the Faculty of International Education and related departments for their expert teaching and the knowledge they shared.

I extend special thanks to Mr. Lương Thái Lê, my project advisor, for her clear guidance, valuable advice, and consistent support throughout this project. Her insights helped me overcome challenges and complete my work on schedule.

I appreciate any feedback or suggestions from faculty that can help improve this project further.

I sincerely express my deepest gratitude !

Ha Noi, day … month … year 2025

Student author

Nguyễn Thành Hưng

INTRODUCTION

In recent years, cloud-native and microservice-based architectures have transformed the landscape of web application deployment. At the forefront of this transformation are reverse proxy systems, which serve as intermediary gateways that route client requests to appropriate backend servers while concealing server details and internal network topology.

Beyond routing, reverse proxies enhance application performance through intelligent load-balancing techniques such as Round Robin, Least Connections, and dynamic server weighting, ensuring efficient resource utilization and resilience under traffic spikes. They also support TLS termination, decrypting and encrypting secure communications to offload cryptographic workloads from origin servers.

In addition, modern proxies can integrate security mechanisms including web application firewalls, rate limiting, and intrusion detection systems to filter malicious payloads and mitigate distributed denial-of-service attacks near the network edge.

However, existing solutions often treat load balancing and security enforcement as separate concerns, leading to architectural complexity, increased latency, and potential coverage gaps. Addressing these challenges, this thesis outlines the design and implementation of a secure reverse proxy system that natively embeds both high-performance traffic management and comprehensive cybersecurity controls, providing a streamlined, centralized platform for protecting and scaling modern web services.

Chapter 1. OVERVIEW

1.1 Problem statement

1.1.1 Motivation

Modern web applications face growing traffic volumes and increasingly sophisticated attacks, making service availability and security paramount. A reverse proxy sits between clients and backend servers, providing both load balancing and a protective barrier against threats such as Distributed Denial-of-Service (DDoS) attacks.

In DDoS scenarios, malicious actors flood a server with requests, but when a reverse proxy is in place, it absorbs and filters such traffic, shielding origin servers from direct exposure. Similarly, distributing client requests across multiple servers prevents any single node from becoming a performance bottleneck, ensuring reliability under load.

1.1.2 Objectives and Scope

The primary goal of this project is to design and implement a Secure Proxy that:

- Balances load across backend servers via Round-Robin, Least-Connection, and Weighted algorithms.
- Mitigates DoS through advanced rate limiting and IP filtering.
- Detects anomalies with a simple Machine Learning model for adaptive rate limiting.
- UI Manager for view request logs and update config for proxy.

The scope is confined to the proxy layer (OSI Layer 7), without modifying backend application logic. We adopt Node.js and express-http-proxy for request forwarding, Docker for containerized deployment.

1.2 Survey of Current Solutions

1.2.1 NGINX

Figure 1.1: NGINX

NGINX is a high-performance web server and reverse proxy known for its event-driven architecture that efficiently handles thousands of concurrent connections with low memory usage. It supports proxying for HTTP, HTTPS, FastCGI, uwsgi, SCGI, memcached, and gRPC protocols, enabling a single gateway for diverse backend services.

NGINX offers multiple load-balancing algorithms—Round-Robin, Least Connections, and IP Hash—to distribute traffic evenly or based on session affinity, with built-in health checks and slow-start to prevent overloading new servers. Its proxy_cache directive allows caching of static and dynamic responses at the edge, significantly reducing origin load and improving client response times. TLS termination is handled natively, offloading SSL processing from upstream applications, and supports OCSP stapling and HTTP/2 for modern encryption standards.

Advantages:

- Performance & Scalability: NGINX's non-blocking, single-threaded model uses minimal resources under heavy load.
- Extensive Protocol Support: Proxying for multiple protocols from a unified configuration.
- Modular Security: Easily integrate WAF modules like ModSecurity for OWASP Top 10 defenses.

Disadvantages:

- Complex Configuration: Its rich feature set requires steep learning to master optimal tuning.
- Reload Overheads: While reloads are non-disruptive, dynamic per-request reconfiguration is limited without external APIs.

- Learning Curve: Advanced features such as Lua scripting and plugin management demand in-depth understanding.

Ideal Use Cases:

- High-traffic websites requiring robust SSL termination and caching.
- Environments demanding support for multiple backend protocols.
- Deployments needing modular security via third-party WAF integrations.

1.2.2 HAProxy

Figure 1.2: HAProxy

HAProxy is a free, fast, and reliable reverse proxy and load balancer for TCP and HTTP applications, designed for high availability and performance. Its event-driven, non-blocking I/O model processes millions of concurrent connections with minimal CPU usage, making it a go-to choice for performance-critical environments.

It supports advanced load-balancing algorithms—including Round-Robin, Least Connections, and Source IP Hash—with stick tables for session persistence and circuit breaking to isolate failing backends automatically. HAProxy includes built-in SSL/TLS termination, HTTP/2 support, and granular ACLs for sophisticated routing and request filtering. Security features such as connection rate limits and per-IP counters help mitigate DDoS attacks at the edge.

Advantages:

- Low Latency & High Throughput: Optimized C code delivers ultra-fast performance under heavy load.
- Health Checks & Failover: Continuous probing of backend servers ensures traffic is only sent to healthy nodes.
- Rich ACL Engine: Offers fine-grained control over routing based on headers, cookies, and IP addresses.

Disadvantages:

- No Static File Serving: Unlike NGINX or Caddy, HAProxy cannot serve static content, requiring an additional web server for that purpose.

- Configuration Complexity: The HAProxy configuration language can be intricate, especially for multi-step routing rules.
- Steep Learning Curve: Advanced features like stick tables and dynamic maps require in-depth knowledge to leverage fully.

Ideal Use Cases:

- Performance-critical applications needing minimal latency and maximum throughput.
- Architectures requiring sophisticated health-check logic and automated failover.
- Scenarios demanding fine-grained traffic control and robust security rules.

1.2.3 Caddy

Figure 1.3: Caddy

Caddy is a modern, open-source web server and reverse proxy written in Go, famous for its simple Caddyfile configuration and automatic HTTPS provisioning via Let's Encrypt. Its native support for HTTP/2 and HTTP/3, combined with a minimal memory footprint, makes it a strong candidate for microservices and developer-focused deployments.

Using the reverse proxy directive, Caddy can load balance across one or more backends using Round-Robin or random selection, with built-in health checks, buffering, and header manipulation options. By default, Caddy automatically obtains and renews TLS certificates for all sites, eliminating manual certificate management and ensuring secure connections out of the box. Caddy's plugin ecosystem extends functionality for authentication, rate limiting, and real-time configuration via its API.

Advantages:

- Ease of Use: Declarative Caddyfile syntax reduces setup time and cognitive overhead.
- Automatic HTTPS: Seamless SSL/TLS management with zero configuration.
- Dynamic Config & API: Live reload support and JSON API allow on-the-fly updates without restarts.

Disadvantages:

- Smaller Ecosystem: Fewer community resources and third-party modules compared to NGINX and HAProxy.
- Basic Load Balancing: Lacks the advanced routing and session persistence features found in HAProxy or NGINX.
- Less Battle-Tested: Newer project with less proven track record in extremely large-scale environments.

Ideal Use Cases:

- Small to medium-sized services needing quick, secure deployments.
- Developers and teams prioritizing simplicity and built-in HTTPS.
- Edge proxies in microservices architectures requiring dynamic configuration.

1.3 Technologies and Tools

1.3.1 Node.js & Express.js

Figure 1.4: NodeJS & ExpressJS

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It is built on Chrome's V8 JavaScript engine and is known for its event-driven, non-blocking I/O model, making it efficient and suitable for data-intensive real-time applications.

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It simplifies the development process by offering a suite of HTTP utility methods and middleware, enabling the creation of APIs and handling of various web functionalities with ease.

In our Secure Proxy System, Node.js serves as the runtime environment, while Express.js facilitates the creation of the server and routing mechanisms, ensuring efficient handling of client requests and responses.

1.3.2 express-http-proxy

The express-http-proxy module is an Express middleware that enables the creation of proxy servers. It allows the forwarding of incoming HTTP requests to another host and returns the response back to the original caller. This module supports features such as path rewriting, header manipulation, and response decoration, providing flexibility in handling proxied requests.

In our system, express-http-proxy is utilized to route client requests to appropriate backend services, facilitating modularity and maintainability in the proxy logic.

1.3.3 Helmet

Helmet is a collection of middleware functions designed to secure Express applications by setting various HTTP headers. These headers help protect the application from common web vulnerabilities, including Cross-Site Scripting (XSS), Clickjacking, and MIME-type sniffing.

By integrating Helmet into our application, we enhance security by configuring headers such as Content-Security-Policy, X-Frame-Options, and Strict-Transport-Security, thereby fortifying the application against a range of potential attacks.

1.3.4 Docker

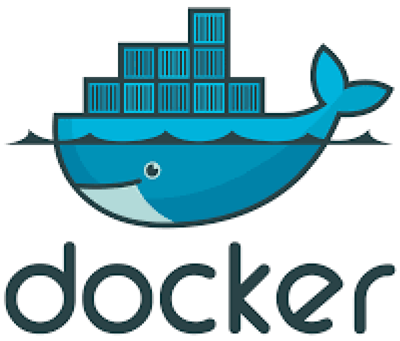


Figure 1.5: Docker

Docker is a platform that enables the creation and deployment of applications within lightweight, portable containers. These containers encapsulate an application and its dependencies, ensuring consistency across development, testing, and production environments.

In our setup, Docker is used to containerize the Node.js application, facilitating scalability and ease of maintenance. This containerized approach also allows for seamless integration with orchestration tools and cloud services.

Chapter 2. REVERSE PROXY OVERVIEW

2.1 What is Reverse Proxy?

2.1.1 Introduction to proxies

Proxy is an Internet server that stands between a client and every server that client wants to access, whole mission is to transport information and enforce controls to ensure secure internet access for clients, allowing encrypted connections and permitting only specific port and protocol.

When a client requests any webpage, that request will sent to Proxy Server, which will forwards it to the target website. Once the website responds, the proxy passes that reponse back to the requesting client. A proxy server can also be used to log internet usage and block access to prohibited sites.

Proxy Server checks every request can handle by cache or not, if not, then it will forward that request to Remote Server. It catchs all requests and check if it can handle that request, if not, it will passes request to servers.

A proxy server acts as a gateway between your device and the internet, masking your IP address and enhancing online privacy.

Purposes of a Proxy

- Increase connection speed: Proxies employ a mechanism called caching, which stores frequently accessed pages locally, allowing requests to be served internally rather than fetched from the Internet, thus speeding up access.
- Enhanced security: Since all client requests and responses must pass through the proxy, it provides a centralized point for enforcing security policies and monitoring traffic, improving protection.
- Filter and block unauthorized access: Proxies can be configured to log Internet usage and to block access to prohibited sites—such as pornographic or extremist content,… by applying URL, IP or content filters.
- Encrypt end-to-end data streams: Proxy servers can be set up to encrypt the data flow based on various parameters, organizations use this feature to establish encrypted connections between two locations, ensuring secure end-to-end communication.

Some types of Proxy Server

- Reverse Proxy: A reverse proxy operates on behalf of web servers rather than clients. It receives incoming client requests, determines which backend server should handle each one, and then forwards the request accordingly. Commonly used for load balancing, caching static assets, and compressing responses, it also masks the actual identities and locations of the servers it represents.

- Web Proxy Server: forwards the HTTP request, the request is sent to particular the proxy server responds. Ex: Apache, HA Proxy.
- Anonymous Proxy Server: this proxy not make an original IP address instead these servers are detectable still provides rational anonymity to the client device.
- Highly Anonymity Proxy: this proxy doesn't allow the original IP address and it as a proxy server to be detected.
- Transparent Proxy: this proxy neither hides the client's IP address nor conceals its own presence. It's often deployed to cache frequently accessed content, speeding up delivery without requiring any configuration on the client side. When combined with a gateway, it can automatically intercept and redirect traffic, with its use detectable through standard HTTP headers.

2.1.2 Introduction to Reverse Proxy

A reverse proxy is a gateway server placed at the network edge that intermediates between clients and one or more origin web servers. It intercepts client requests, inspects and filters out any invalid requests before forwarding approved ones to the back-end web servers, then returns the server's response to the client. Unlike a forward proxy—which operates on behalf of clients—a reverse proxy masks the identities of the origin servers while centralizing request handling and improving security, scalability, and load management.

Additionally, correctly applying a reverse proxy can boost performance and improve the scalability of your web applications running on servers. A reverse proxy can also conceal and mask the existence of your origin servers. When a client on the Internet wants to access your web server, the reverse proxy acts as that web server-clients communicate only with the proxy, believing it to be the real server. The reverse proxy then forwards their requests to the actual web server and relays the responses back to the clients.

Reverse Proxy Server takes the place of server when communicating with the client. Its built-in firewall-style features can be defend the web server against common online attacks. Without a reverse proxy to filter out malware, securing server becomes more difficult. The widely used reverse proxy operate over the HTTP and HTTPS protocols.

2.2 How does a Reverse Proxy work?

2.2.1 Model of Reverse Proxy solution

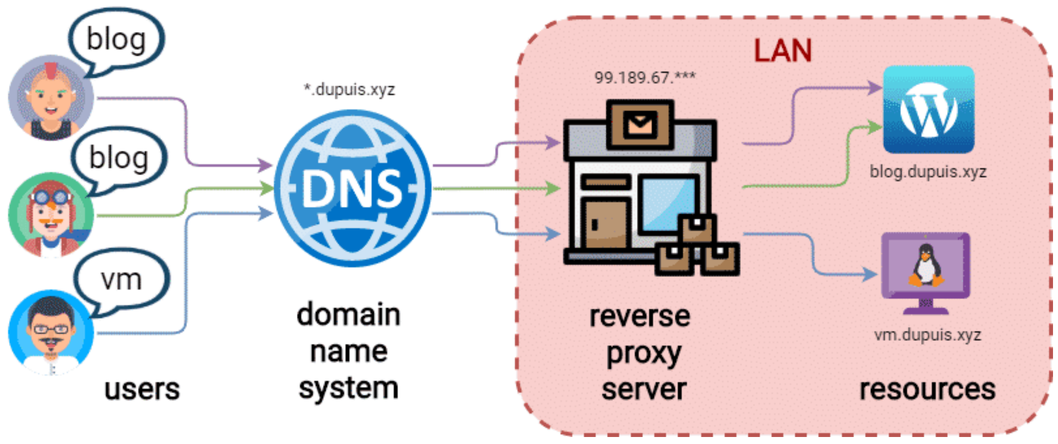


Figure 2.1: Model of reverse proxy

The reverse proxy architecture consists of five primary components:

- Public Endpoint: All client is directed to a single publicly routable IP address or DNS name, simpilifying external access and hiding internal server details.
- Proxy cluster: One or more proxy instances (often in active–active or active–passive configurations) terminate SSL/TLS, enforce access controls, and pre-filter requests to protect backend servers.
- Load distribution layer (optional): An internal load balancer or service mesh can spread validated traffic across multiple proxy nodes or segregated application pools, enhancing availability and fault tolerance.
- Origin (backend) servers: The actual web or application servers host business logic and content. These servers never connect the Internet directly, reducing their attack surface.
- Supporting services: Components integrate with the proxy layer to boost performance and security like cache, logging and monitoring, sercurity modules (rate limiting, DdoS mitigation).

2.2.2 Reverse Proxy mechanism

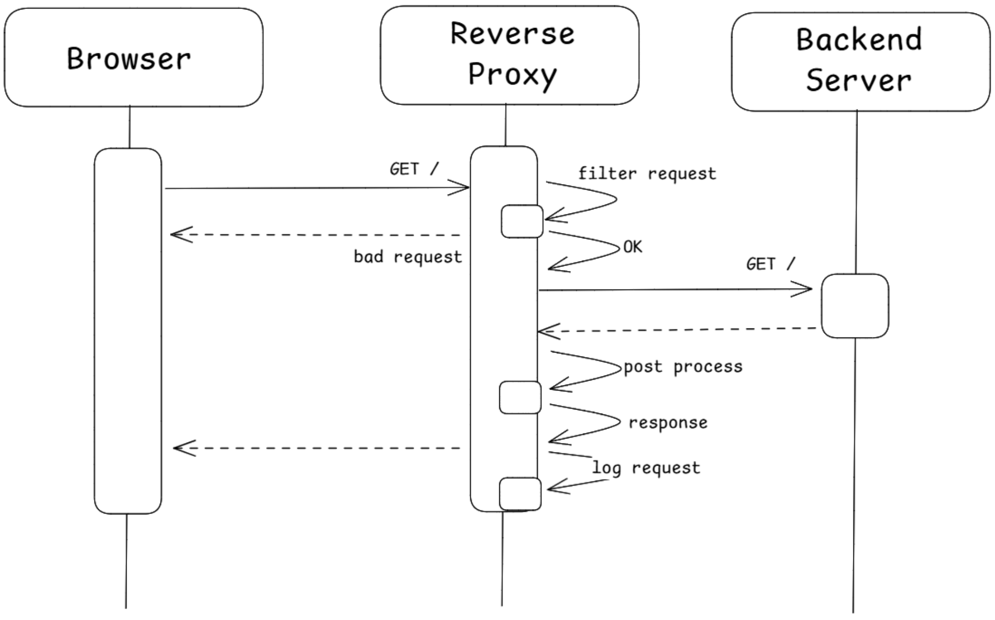


Figure 2.2: Reverse proxy mechanism

The reverse proxy processes each incoming request step by steps:

1. DNS Resolution: Clients perform domain lookup, receiving the reverse proxy's IP address, which centralizes all traffic at the proxy layer.
2. Connection Establishment & TLS Handshake: A TCP connection is established and, if using HTTPS, the TLS handshake is performed between client and proxy.
3. TLS Termination: The proxy decrypts incoming SSL/TLS traffic offloading cryptographic work from origin servers and may re-encrypt for internal transmission if end-to-end encryption is required.
4. Request Validation: HTTP methods, headers, URLs, and payloads are inspected against security policies; malformed or malicious requests are dropped or challenged immediately.
5. Caching & Compression: Static assets are served directly from cache when fresh; otherwise fetched from the backend and stored. Responses can also be compressed (e.g., gzip, Brotli) to reduce bandwidth.
6. Load Balancing: A healthy origin server is selected based on algorithms such as round-robin, least-connections, or IP-hash, periodic health checks ensure traffic is routed only to responsive nodes.
7. Header Manipulation: The proxy appends or modifies headers (e.g., X-Forwarded-For, Via) to preserve client metadata and conform to backend expectations.
8. Request Forwarding: The sanitized and annotated request is sent to the chosen backend server over the internal network.

9. Response Handling: Upon receiving the backend’s response, the proxy may cache it for subsequent requests and apply further compression or header rewrites before returning it to the client.
10. Logging & Metrics Collection: Transaction details such as latency, response codes, and throughput are logged and exported to monitoring systems for analysis, alerting, and capacity planning.

2.3 Load balancing with Reverse Proxy

One of the biggest advantages of a Reverse Proxy is its ability to balance the load across multiple servers. Web servers often handle a large volume of traffic. In addition, real-time traffic such as video or multimedia has become increasingly common.

As a result, a single web server may need to serve a large number of clients. In such situations, a Reverse Proxy can distribute requests to multiple servers so that each one handles a portion of the load, helping to prevent congestion. A Reverse Proxy can also detect when a server is overloaded and temporarily stop sending requests to it until the load decreases.

Load balancing is commonly categorized into two types: Layer 4 and Layer 7. Layer 4 load balancing operates on data from network and transport layer protocols such as IP, TCP, FTP, and UDP. Layer 7 load balancing distributes requests based on information found in application layer protocols like HTTP.

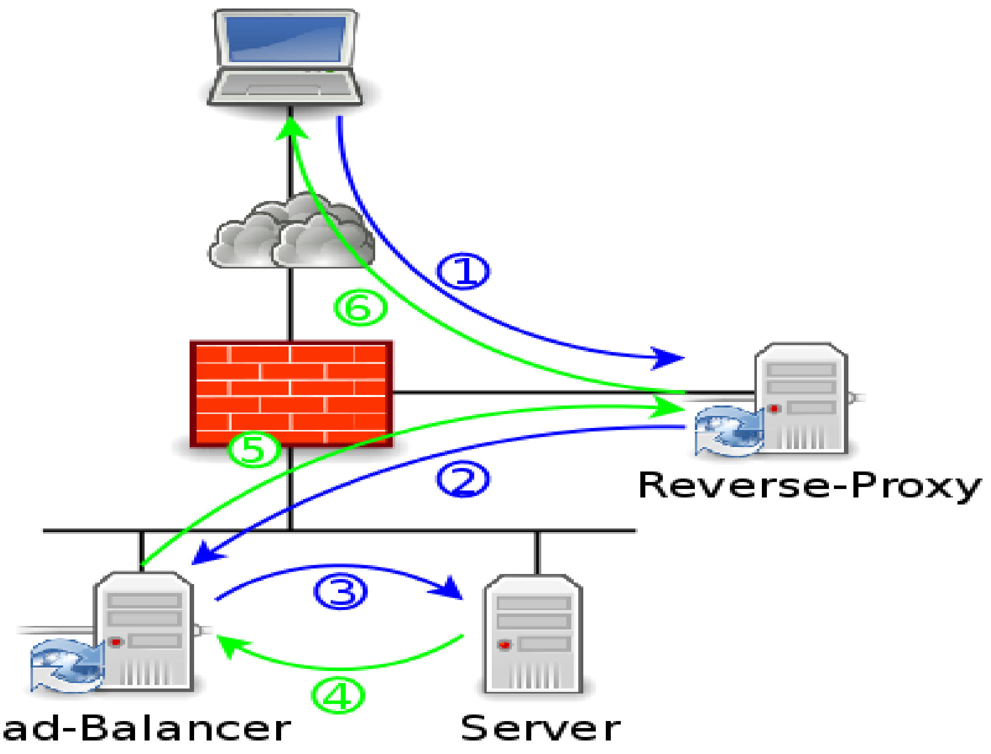


Figure 2.3: Flow with load balancer

Requests handled by both types of load balancing are distributed to specific servers based on predefined algorithms. Some of the standard industry algorithms include:

- Round Robin: Requests are assigned to servers in a sequential, rotating order, ensuring equal distribution over time.
- Random: A random server is selected.
- Failover: In case of failure, this exchange is tried on the next server.
- Weighted Round-Robin: Similar to Round Robin but assigns more requests to servers with higher capacity or priority, based on assigned weights.
- Weighted Random: The load balancing policy allows to specify a load distribution processing ratio for each server. Server selection using random distribution.

Layer 7 load balancing can further distribute requests based on specific application-level data, such as HTTP headers, cookies, or custom message data within the application itself.

This type of load balancing ensures reliability and availability by continuously monitoring the health of applications and only forwarding requests to servers and services that are able to respond in a timely manner.

2.4 Cache Reverse Proxy

The goal of caching is to accelerate content delivery by reducing the amount of repetitive work the web server has to perform. Storing a file in cache for reuse can save millions of disk access operations, thereby speeding up how quickly the browser delivers what the user needs.

Caching can operate at three different levels within a web application:

- Web Server Layer: Static content such as HTML, CSS, JavaScript, and images are cached to reduce load on the server and improve response time.
- Application Layer: Frequently used data or computations (e.g., user sessions, rendered views) are cached to avoid repeated processing.
- Database Layer: Query results are stored in cache to minimize the frequency and cost of accessing the database directly, significantly improving performance.

2.5 Security benefits of Reverse Proxy

2.5.1 Common Types of Attacks

- Denial-of-Service (DoS): type of cyberattck that only 1 internet-connected computer attacks a different computer with traffic especially a server to make it crash. It will send a multiple requests which is cause server to either crash or be unavaible to users of the websites.

- Distributed Denial-of-Service (DDoS): similar as the DoS attack but is more complicated in the way attack is lauched, the attack will start with the help of serveral systems located in different places. These systems can be computers or ‘bots’ that run in parallel with high traffic volume. An inherent advantage of a distributed attack is that it is difficult to track the origin and, therefore, put a stop to it.
- SQL Injection (SQLi): occurs when an attacker supplies malicious input—often in form fields or URL parameters—that alters the structure of database queries, allowing unauthorized data access or modification.
- Cross-Site Scripting (XSS): malicious scripts are injected into web pages viewed by other users, enabling session hijacking, defacement, or execution of arbitrary code in the victim’s browser context.
- Timing Attack: a side-channel exploit where an attacker measures the time taken for cryptographic or application operations to infer sensitive information, such as secret keys or validation logic.
- Protocol Abuse (Protocol-Level DDoS): exploit weaknesses in network protocols (e.g., TCP, UDP, FTP) by sending malformed or half-open requests that consume server resources, such as SYN floods or ping-of-death variants.
- Reconnaissance / IP Scanning: scanning IP ranges and open ports (using tools like Nmap) to map network services and identify potential vulnerabilities for later exploitation.
- Application-Layer Attacks (OWASP Top 10): Application-layer attacks target flaws in web application logic—such as broken authentication, insecure deserialization, or misconfigurations—and are catalogued in the OWASP Top 10 as the most critical risks to web security.

2.5.2 How a Reverse Proxy Prevents Attacks

- IP Concealment: By exposing only its own public IP, the proxy hides the addresses of origin servers, preventing direct targeting and reconnaissance.
- Web Application Firewall (WAF) Integration: Modern proxies often include or front WAF modules that inspect HTTP(S) traffic for attack signatures—such as SQLi, XSS, and CSRF—and block them at the edge.
- Rate Limiting and Throttling: Proxies can enforce per-client or per-endpoint request limits, stopping high-frequency request floods and brute-force attacks before they consume backend resources.
- SSL/TLS Termination and Enforcement: Offloading encryption tasks to the proxy allows centralized management of certificates and ensures that all incoming traffic is properly encrypted, preventing downgrade and man-in-the-middle attacks.

- Traffic Validation and Sanitization: Proxies perform deep inspections of headers, URLs, and bodies, rejecting malformed or non-compliant requests to stop protocol-level abuses and injection attempts.
- Health Checks and Intelligent Failover: By continuously monitoring origin server health, the proxy can remove unresponsive or compromised nodes from service, preventing failed or malicious servers from handling requests.
- Centralized Logging and Anomaly Detection: All traffic passes through the proxy, enabling comprehensive logging. Security teams can analyze logs for unusual patterns—such as spikes in error codes or unexpected URIs—and trigger automated defenses.
- Authentication and Access Control: Proxies can integrate with identity providers to enforce authentication (OAuth, JWT) and restrict access based on role or location, blocking unauthorized users at the gateway.

Through these combined measures, a reverse proxy not only acts as a barrier against common web threats but also provides a flexible platform for advanced security policies, ultimately safeguarding backend servers and ensuring continuous, reliable service.

2.6 Conclude

Thus, this chapter has provided an overview of Reverse Proxy, its operating mechanism, and its key benefits such as load balancing and enhanced security. The content presented here will serve as the theoretical foundation for the implementation and testing of the system in Chapter 3.

Chapter 3. Building a secure Reverse Proxy

3.1 Model of Reverse Proxy solution

To build secure reverse proxy, I use Docker Compose to define and run all components, eliminating the need to manually manage individual containers or networks.

The docker-compose.yml file declares five services: the proxy (Express.js), three backend application servers (backend1–backend3), and an AI threat-detection service (ai-service), each in its own container with isolated environments and shared networking. Docker Compose automatically creates a private overlay network, allowing service names (e.g., backend1) to resolve to container IPs, which simplifies inter-service communication and hides internal details from the outside world.

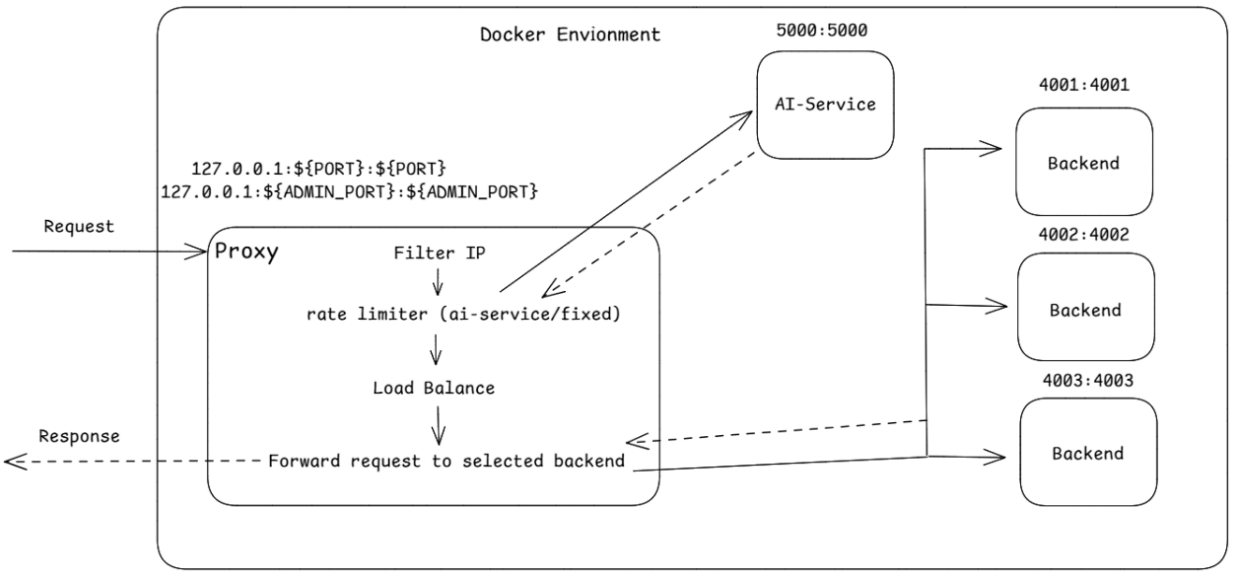

Docker Envionment
5000:5000
AI-Service
4001:4001
Backend
127.0.0.1:${PORT}:${PORT}
127.0.0.1:${ADMIN_PORT}:${ADMIN_PORT}
Request
Proxy
Filter IP
rate limiter (ai-service/fixed)
4002:4002
Backend
Load Balance
4003:4003
Backend
Response
Forward request to selected backend

Figure 3.1: Project architecture

1. Public Endpoint: The proxy service binds host ports ${PORT} and ${ADMIN_PORT} for admin page to the container, providing the single public entry point for all HTTP(S) traffic.
2. Proxy Cluster: Express.js instance within the proxy container terminate TLS (when enabled via config), apply security middleware (helmet, rate limiting, AI-driven threat checks, …), and perform request routing. This also enforce HTTP headers for security and logging via morgan to ‘/logs/access.log’.
3. AI Threat Detection (ai-service): Receives request from the proxy over HTTP to score malicious behavior before forwarding.
4. Backend Servers: Each backend service runs a Node.js application on ports 4001-4003 inside ‘node:18-slim’ containers, hidden from external access and only reachable via the proxy.

3.2 Features and Implementation

3.2.1 Proxy configuration and validate config

User setup environment variable `CONFIG_PATH` for config path. Config validated by ‘convict’ to make sure value in config set correctly and avoid invalid value. This is the example of config file:

```
{
  "proxy": {
    "port": 3000,
    "ssl": {
      "enabled": true,
      "key": "./ssl/key.pem",
      "cert": "./ssl/cert.pem"
    }
  },
  "trustProxy": {
    "enable": true,
    "trustList": ["loopback"]
  },
  "backends": [
    { "host": "backend1", "port": 4001, "weight": 1 },
    { "host": "backend2", "port": 4002, "weight": 2 },
    { "host": "backend3", "port": 4003, "weight": 1 }
  ],
  "loadBalancing": {
    "algorithm": "roundRobin"
  },
  "security": {
    "rateLimit": {
      "type": "ai-detect",
      "windowMs": 5000,
      "max": 300
    },
    "ipBlacklist": [""]
  }
}
```

3.2.2 Load Balacing

1. Round Robin: Distributes requests to backends in simple rotation: each new request goes to the next server in the list, looping back to the first after the last. In code, a `roundRobinIndex` variable is incremented modulo the number of backends to select the next target:

```
let roundRobinIndex = 0;
const target = backends[roundRobinIndex];
roundRobinIndex = (roundRobinIndex + 1) % backends.length;
return target;
```

2. Least Connections: Routes new requests to the server with the fewest active connections at that moment. Proxy track a connections array where each slot corresponds to a backend, increments occur on dispatch and decrements in a res.on('finish') handler:

```
let connections = new Array(backends.length).fill(0);
let minConnections = Infinity;
let minIndex = 0;
for (let i = 0; i < connections.length; i++) {
    if (connections[i] < minConnections) {
        minConnections = connections[i];
        minIndex = i;
    }
}
connections[minIndex]++;
return backends[minIndex];
```

```
res.on('finish', () => {
    const index = backends.findIndex(b => b.host === target.host && b.port === target.port);
    connections[index]--;
});
```

3. Weighted Round Robin: ssigns a weight to each backend so high-capacity servers receive more traffic. Algorithim compute 'totalWeight' as the sum of all backend's weight, pick a random threshold, then iterate until the cumulative weight exceeds it:

```
let weightIndex = 0;
let currentWeight = 0;
let maxWeight = backends.reduce((sum, b) => sum + (b.weight || 1), 0);
while (true) {
    weightIndex = (weightIndex + 1) % backends.length;
    currentWeight += backends[weightIndex].weight || 1;
    if (currentWeight >= Math.random() * maxWeight) {
        return backends[weightIndex];
    }
}
```

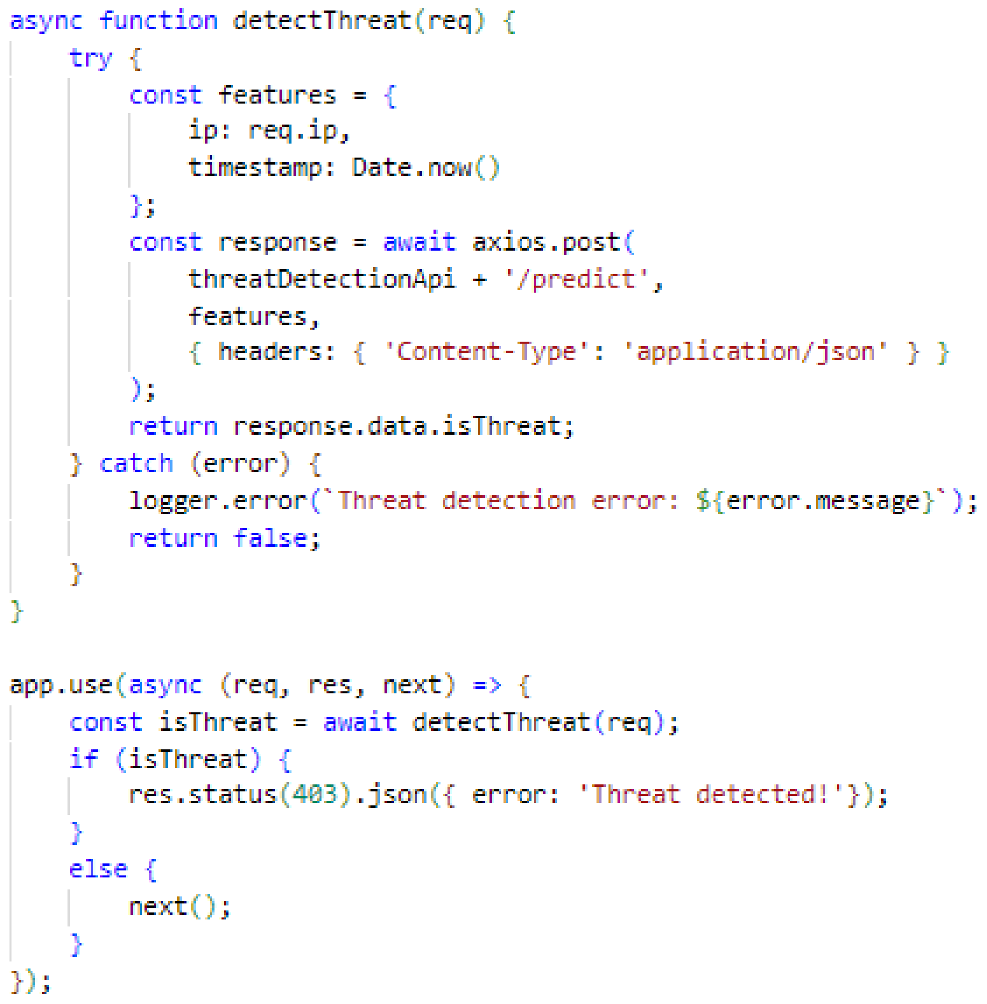
For this load balancing algorithm, user also needs to add weight to each backend.

3.2.3 Rate Limiting

1. Fixed rate limiting: Using the 'express-rate-limit' middleware, it cap requests per IP within a fixed window.

```
import rateLimit from 'express-rate-limit';
if (typeof rateLimit.windowMs !== 'number' || typeof rateLimit.max !== 'number') {
    throw new Error('windowMs and max must be set for fixed rate limiting');
}
const limiter = rateLimit({
    windowMs: rateLimitConfig.windowMs,
    max: rateLimitConfig.max,
    message: "Too many requests, please try again later.",
});
app.use(limiter);
```

2. AI-Driven Threat Detection: An external Flask-based service uses PySAD's IForestASD (Isolation Forest for streaming anomaly detection) to score request patterns in real time. The proxy extracts features IP address and timestamp and POST them to '/predict', if 'isThreat' is true, the request is blocked:



```
async function detectThreat(req) {
    try {
        const features = {
            ip: req.ip,
            timestamp: Date.now()
        };
        const response = await axios.post(
            threatDetectionApi + '/predict',
            features,
            { headers: { 'Content-Type': 'application/json' } }
        );
        return response.data.isThreat;
    } catch (error) {
        logger.error(`Threat detection error: ${error.message}`);
        return false;
    }
}

app.use(async (req, res, next) => {
    const isThreat = await detectThreat(req);
    if (isThreat) {
        res.status(403).json({ error: 'Threat detected!'});
    }
    else {
        next();
    }
});
```

The AI service maintains per-IP request histories over sliding windows (5 min, 1 h, 24h) updates the model incrementally.

How AI service works:

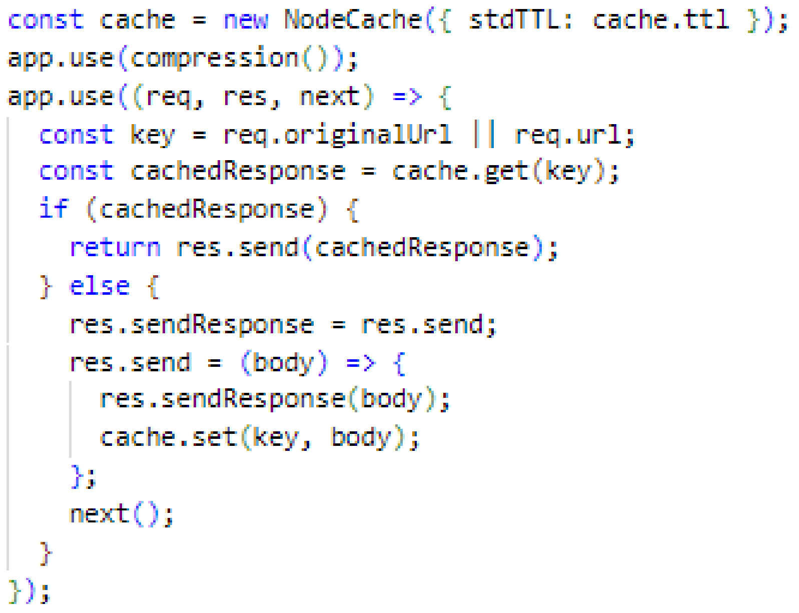
- Pretraining: the service reads the 'log/access.log' file, extracting the client IP and timestamp. For each IP, the service maintains a list of past request timestamps in map. the service calls 'model.fit_partial(X)' on the feature vector 'X = np.array([count_5min, count_1h, count_24h])', updating the model state.
- Real-Time Detection: The proxy sends JSON containing two keys: ip and timestamp. The service receive it and appends it to request history and also remove entries older than 24h. Then service counts over the three windows exactly as in pretraining, return a features list like `[10, 30, 500]' (requests in last 5 min, 1h, 24h). After that, service update the model and scoring with 'score = model.score_partial(X)' then compare with 'THRESHOLD'. The

response will return like ‘{ "isThreat": true, "score": 0.27 }’ and proxy use ‘isThreat’ to decide block malicious IPs in real time.

- All predictions are logged to ai.log via Python’s logging module at INFO level.

3.2.4 Caching

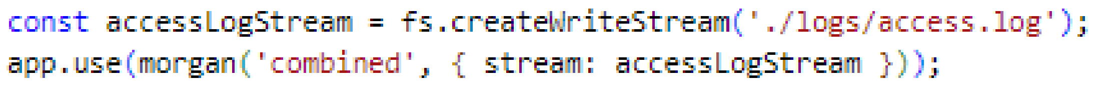
An in-memory cache via node-cache stores full HTTP responses keyed by URL to cut backend round-trips and disk I/O.



```
const cache = new NodeCache({ stdTTL: cache.ttl });
app.use(compression());
app.use((req, res, next) => {
  const key = req.originalUrl || req.url;
  const cachedResponse = cache.get(key);
  if (cachedResponse) {
    return res.send(cachedResponse);
  } else {
    res.sendResponse = res.send;
    res.send = (body) => {
      res.sendResponse(body);
      cache.set(key, body);
    };
    next();
  }
});
```

3.2.5 Logging

All HTTP transactions are logged in Apache Combined format to ‘/logs/access.log’ using ‘morgan’ package.



```
const accessLogStream = fs.createWriteStream('./logs/access.log');
app.use(morgan('combined', { stream: accessLogStream }));
```

3.3 Conclude

The constructed reverse proxy meets the dual goals of security and scalability. Its modular architecture allows insertion of Middleware for load balancing, AI-based threat detection, rate limiting, caching, and TLS termination without modifying origin servers. Performance tests can show the effect of caching, SSL and AI integrations incur acceptable overhead.

CONCLUSION

From practice, it shows that reverse proxies have become a vital part of modern web infrastructure. This growth reflects rising needs for privacy, security, and performance as more applications move to edge and microservices architectures.

Developing this type of application, in addition to ensuring service quality, must also ensure the confidentiality of information in the media environment. In particular, information protection is being developed by developers today. Thesis topic: “Building a secure proxy system for load balancing and cybersecurity” was carried out with the aim of researching and providing methods for network security in general and information protection on transmission lines in particular and has achieved the following results:

General introduction to concepts such as proxy, common cyber attacks and prevention measures. Learn the structure and operating principles of reverse proxy and load balancing. Built a complete reverse proxy solution with containerized via Docker Compose that includes an Express.js gateway, three backend servers, and a Flask-based AI service for anomaly detection with security features such as SSL/TLS termination, HTTP header hardening, IP filtering, and adaptive rate limiting to block common web attacks.

During the project implementation process, due to limited knowledge and limited time, the project is inevitably flawed. I look forward to receiving comments from teachers and friends.

REFERENCES

[1] Cloudflare. “What is a reverse proxy? | Proxy servers explained.” Learning Center, Cloudflare.

[2] GeeksforGeeks. “What is Proxy Server?” GeeksforGeeks, last updated 08 Apr, 2025.

[3] EdgeNext. “How Does AI Help Detect and Prevent DDoS Attacks in Modern Networks,” by Kaiyue, 13 May 2025. EdgeNext Blog.