Design and Implementation of a Custom Network Proxy Server

Overview

A network proxy server is an intermediary system that accepts client network requests and forwards them to destination servers, then relays server responses back to the clients. This project focuses on implementing a forward proxy that handles HTTP traffic over TCP. The implementation will demonstrate fundamental systems and networking concepts including socket programming, concurrency, request parsing, forwarding logic, logging, and rule-based filtering. Support for HTTPS tunneling, caching, and authentication is optional and may be implemented as extensions.

Objectives

The primary objectives of the project are:

1. To implement reliable TCP-based client–server communication using sockets.
2. To design and implement a concurrent network service capable of handling multiple clients.
3. To implement correct parsing and forwarding of HTTP requests and responses.
4. To implement configurable traffic control mechanisms, including logging and domain/IP filtering.
5. To produce a modular, documented, and testable codebase that can be extended with caching and HTTPS tunneling.

Deliverables

The project submission must include the following items:

1. **Source code**
    - Complete, documented source implementing the proxy server.

- Build instructions (Makefile or equivalent) and runtime invocation examples.

2. **Configuration files**
    - Filtering rule file (for example, a plain text file listing blocked domains/IPs).
    - Server configuration (listening address/port, thread-pool size, paths for logs and cache).
3. **Design document**
    - High level architecture diagram and component descriptions.
    - Concurrency model and rationale (thread-per-connection, thread pool, or event loop).
    - Data flow description for incoming request handling and outbound forwarding.
    - Error handling, limitations, and security considerations.
4. **Test artifacts**
    - Test scripts or commands (curl examples, automated test scripts) demonstrating: successful forwarding of requests, blocking of blacklisted domains, behavior under concurrent clients, and handling of malformed requests.
    - Sample log files produced during tests.
5. **Demonstration materials**
    - Short document or screenshots demonstrating proxy usage (e.g., `curl -x localhost:8888 http://example.com`). Optionally, include a short video or terminal recording.
6. **Optional extension deliverables (if implemented)**
    - Caching component with eviction policy (implementation and evaluation).
    - HTTPS CONNECT tunneling support description and tests.
    - Authentication mechanism and configuration.

Hints (to get started)

This section gives concrete, practical guidance for implementing the proxy server. The hints are grouped by design topic and include suggested starting points and common pitfalls.

1. Project layout

Recommended repository structure:

```
proxy-project/
├─ src/                # source files
├─ include/            # headers (C/C++)
├─ tests/              # test scripts and inputs
├─ config/             # example config files (blocked_domains.txt)
├─ docs/               # design doc and README
├─ Makefile or setup.py
```

-
- Keep networking, parsing, and filtering modules separate to improve testability.

2. Basic networking and connection handling

- Create a listening TCP socket bound to a configurable address and port.
- Use one of the following concurrency models:
    - **Thread-per-connection**: simple to implement; acceptable for prototyping. Use `pthread` in C or `threading` in Python.
    - **Thread pool**: use a fixed-size pool to control resource usage and queue incoming connections.
    - **Event-driven**: use `select`/`poll`/`epoll` (C) or `asyncio` (Python) for scalable I/O.
- Always handle partial reads/writes: network `recv`/`send` may return fewer bytes than requested.
- Set reasonable socket timeouts and ensure sockets are closed on errors.

3. Parsing HTTP requests

- At minimum, parse the HTTP request line and headers to obtain:
    - Method (e.g., GET, POST, CONNECT)
    - Request target (path or absolute URI for proxy requests)
    - Host header (for destination host if URI is relative)
- For proxy operation, the client may send absolute URIs (e.g., `GET http://example.com/path HTTP/1.1`) — handle both absolute and relative forms.
- Be careful with `Content-Length` and request bodies: for methods that include a body (e.g., POST), forward the exact number of body bytes specified by `Content-Length`.
- Handling of **chunked transfer encoding** is more complex; treat this as an optional extension. If not implementing chunked handling, you may forward bytes transparently between sockets without interpreting chunks.

4. Forwarding data

- After parsing destination host and port, open a TCP connection to the destination server and forward the client's request bytes. Use a loop to forward until the entire request (headers + body if present) has been sent.
- Relay server responses back to the client in a streaming fashion; do not buffer entire responses in memory for large transfers.

5. Filtering and configuration

Use a simple configuration file format for filters (one domain or IP per line). Example:

```
 # blocked_domains.txt
example.com
badsite.org
192.0.2.5
```

-
- Implement a canonicalization step for hostnames (lowercase, trim whitespace) before matching. Support simple suffix matching (e.g., blocking `*.example.com`) as an extension.

- When a request is blocked, return an appropriate HTTP error response (e.g., `HTTP/1.1 403 Forbidden`) and log the event.

6. Logging and metrics

- Log entries should minimally include: timestamp, client IP:port, requested host:port, request line, action (allowed/blocked), response status, and size transferred.
- Use a simple log rotation strategy or keep log files bounded in size for long-running tests.
- Optionally, implement a small metrics summary (requests per minute, top requested hosts) for the demonstration.

7. HTTPS tunneling (CONNECT method)

Support for HTTPS is typically done via the `CONNECT` method. When the proxy receives:

```
CONNECT host:port HTTP/1.1
```

-
    - Establish a TCP connection to `host:port`.
    - If successful, respond `HTTP/1.1 200 Connection Established` to the client.
    - Then forward bytes bidirectionally between client and server without interpreting them (the encrypted TLS stream passes through).
- Do not attempt to inspect or modify TLS payloads unless you implement explicit TLS interception (out of scope for a basic proxy).

8. Caching (optional)

- For caching, a standard approach is an in-memory key-value store where the key is the normalized request URI and the value is a cached response.
- Implement an eviction policy such as LRU using a hash map plus a doubly-linked list for O(1) operations.

- Store cache metadata (timestamp, content-length, headers) to respect cacheability rules (can be simplified for the project).

9. Testing strategies

- Simple tests:
    - `curl -x localhost:8888 http://example.com`
    - `curl -x localhost:8888 -I http://example.com` (HEAD request)
- Concurrent testing:
    - Use a script to start multiple `curl` processes in parallel or use `ab` or `wrk` (load testing tools) for basic concurrency tests.
- Blocking tests:
    - Add a domain to the blocked list and confirm that requests for that domain return `403 Forbidden`.
- CONNECT tests:
    - `curl -x localhost:8888 https://example.com` to verify CONNECT tunneling (requires proxy support and TLS handshake through tunnel).
- Inspect logs to verify the proxy's recorded actions.

10. Common pitfalls and recommendations

- Do not assume that `recv` returns a full HTTP header in one call. Implement a header-accumulation loop that reads until the header terminator (`\r\n\r\n`) is found.
- Be careful with persistent connections (`Connection: keep-alive`) — for simplicity you may choose to close connections after a single request/response cycle.
- Validate and sanitize configuration-driven inputs to avoid injection issues in logs or config parsing.
- Handle SIGINT/SIGTERM to allow graceful shutdown and resource cleanup.

Resources

1. **Programming references**
    - POSIX socket API documentation for `socket`, `bind`, `listen`, `accept`, `connect`, `recv`, `send`, and `setsockopt`.
    - Documentation for concurrency primitives: `pthread` (C), `threading`/`asyncio` (Python).
2. **Tools for testing and debugging**
    - `curl`, `wget` (client tools that support proxy usage).
    - `netcat`/`socat` for low-level socket testing.
    - `tcpdump` or `wireshark` for optional packet-level inspection during development (use with caution and appropriate permissions).