COMP3331/9331 Computer Networks and Applications

HTTP Implementation - Assignment for Term 2, 2025

Due: 16:59 Thursday, 31 July 2025 (Week 9)

Date	Description
29/05/2025	Initial release

- ⇦ Useful tools for testing and debugging
- ⇦ Frequently asked questions

Introduction

Background

The **Hypertext Transfer Protocol (HTTP)** is the backbone of the World Wide Web, enabling everything from browsing websites and streaming videos to online banking, shopping, and social media. Every time you load a webpage, send a message, or access cloud-based applications, HTTP facilitates communication between your device and remote servers, making modern internet usage seamless and efficient.

HTTP allows the use of intermediaries to handle requests through a chain of connections. One common type of intermediary is a **proxy**, a message-forwarding agent that sits between a **client** (such as a Web browser) and an **origin server**. Instead of connecting directly to the origin server, the client sends its request to the proxy, which then forwards it on behalf of the client. The origin server's response is then relayed back through the proxy to the client.

Because a proxy handles both incoming and outgoing requests, it effectively acts as **both a client and a server**—a server to the clients making requests and a client to the servers fulfilling them.

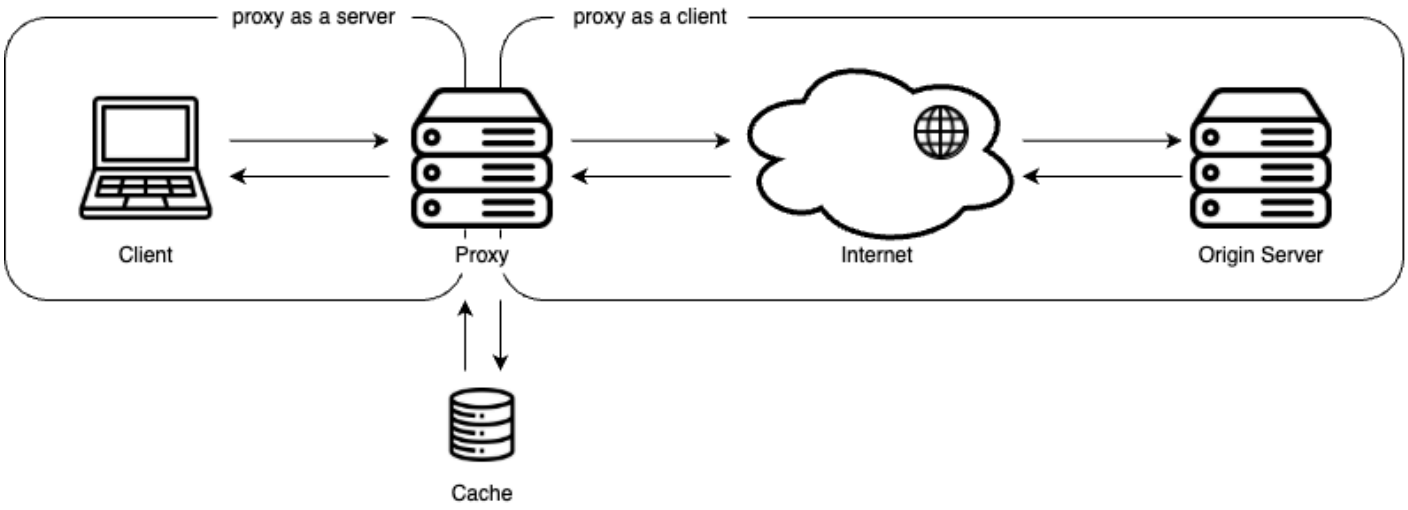


Figure 1: The proxy as a middleman

Proxies serve a variety of purposes, including:

- **Filtering requests**, such as blocking access to restricted websites.
- **Transforming responses**, such as compressing or converting images to reduce bandwidth usage.
- **Caching responses**, to improve performance and reduce redundant network traffic.
- **Anonymising clients**, by stripping identifying information from requests.

Task

In this assignment, you will implement your own HTTP/1.1 proxy in **C, Java or Python**. Your proxy should:

- Support the `CONNECT`, `GET`, `HEAD`, and `POST` methods.
- Handle multiple requests over a single connection (persistence).
- Handle multiple client connections concurrently (concurrency).
- Cache responses for potential reuse in future requests.

Your proxy **must** compile and run within the CSE environment. Ensure it is thoroughly tested in that environment.

You are only permitted to use the basic libraries for socket programming. You **must not** use any ready-made server or HTTP libraries to implement any aspects of this assignment. Doing so will likely result in a mark of **zero**. If in doubt, please check with course staff on the forum.

This is an **individual assignment** and is worth **20 marks**.

Scope

This assignment specification is the **authoritative reference** for your implementation. While it is based on HTTP standards, to simplify your task certain aspects may **deviate from or contradict official specifications**. In such cases, the requirements outlined here take precedence. If you encounter any ambiguity, seek clarification via the course forum rather than relying on official specifications.

Learning Objectives

By completing this assignment, you will gain a deeper understanding of:

- **HTTP** as an application-layer protocol and its role in web communication.
- The **client-server model** and how intermediaries like proxies interact with clients and servers.
- **TCP socket programming** for handling network communication.
- **Concurrent programming** to manage multiple connections efficiently.
- **Caching mechanisms** to improve performance and reduce redundant requests.

Resources

- Lecture Notes: Application Layer Part 1 – slides on HTTP and Web caches
  - See Moodle for the lecture recording
  - Related textbook chapter: 2.2 The Web and HTTP
- Lecture Notes: Application Layer Part 2 – slides on socket programming
  - See Moodle for the lecture recording
  - Related textbook chapter: 2.7 Socket Programming: Creating Network Applications
- Sample Client-Server Programs and Networking Programming Resources
- Lab 2 – HTTP and Socket Programming
- Lab 3 – A Simple Web Server

- Frequently Asked Questions
- Useful Tools for Testing and Debugging
- Programming Tutorial - in week 7, more details to come
- Help Sessions - starting week 7, more details to come
- MDN Web Docs – a great resource for web technologies
- RFC 9110 – HTTP Semantics (optional, for those interested)
- RFC 9111 – HTTP Caching (optional, for those interested)
- RFC 9112 – HTTP/1.1 (optional, for those interested)

Deliverables

- Source code
  - The main entry point of your program **must** be named one of:
    - `proxy.c` (if implementing in C)
    - `Proxy.java` (if implementing in Java)
    - `proxy.py` (if implementing in Python)
  - You are free to create additional helper files and name them as you wish.
- A `Makefile` (not required for Python)
  - Running `make` should produce an executable named:
    - `proxy` (if implementing in C)
    - `Proxy.class` (if implementing in Java)
- A Report named `report.pdf`

Interface

Your proxy **must** accept 4 command-line arguments:

- `port`: the TCP port the proxy will listen on for client connections. We recommend using a random port number between 49152 and 65535 (the dynamic port number range).
- `timeout`: a strictly positive integer, in seconds. It's the duration a client or server may be idle or unresponsive.
- `max_object_size`: a strictly positive integer, in bytes. It's the maximum object size that can be cached.
- `max_cache_size`: a strictly positive integer, in bytes, and at least equal to `max_object_size`. It's the total amount of object data that may be cached.

It should be initiated as follows:

```
$ ./proxy          <port> <timeout> <max_object_size> <max_cache_size>  # for C
$ java Proxy       <port> <timeout> <max_object_size> <max_cache_size>  # for Java
$ python3 proxy.py <port> <timeout> <max_object_size> <max_cache_size>  # for Python
```

For example, running one of:

```
$ ./proxy          60893 10 1024 1048576  # for C
$ java Proxy       60893 10 1024 1048576  # for Java
$ python3 proxy.py 60893 10 1024 1048576  # for Python
```

Would mean the proxy should:

- Listen on port `60893` (avoid this port on CSE, other students will likely try and use it simply because it's in the spec!).
- Timeout clients/servers after `10` seconds.
- Not cache any objects greater than `1024` bytes.
- Not allow the cache to grow any greater than `1048576` bytes.

You must ensure that none of these values are hard-coded. They must be configurable via the command-line arguments.

HTTP Concepts

Requests vs. Responses

HTTP is a stateless request/response protocol used for exchanging **messages** over a network connection. These messages fall into two categories: **requests**, which clients send to initiate an action on the server, and **responses**, which servers return to fulfill those requests.

Both types of messages share a common structure. They begin with a **start-line**, followed by zero or more **header field lines** (collectively called the **headers** or **header section**), an empty line marking the end of the headers, and an optional **message body**. The line terminator for the start-line and header fields is the sequence CRLF (`\r\n`, representing carriage return and line feed).

Structurally, the only difference between request and response messages is the **start-line**:

- **Requests**: Include a **request-line** specifying the **HTTP method**, **request target**, and **protocol version**.
- **Responses**: Include a **status-line** with the **protocol version**, **status code**, and an optional **reason phrase**.

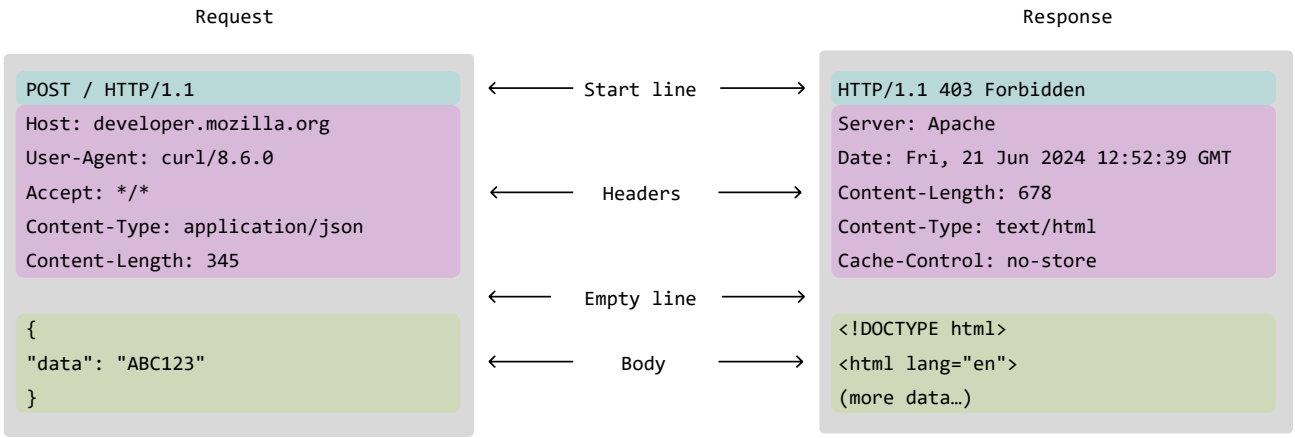


Figure 2: Anatomy of an HTTP message (Source: MDN Web Docs)

HTTP Methods

The request method defines the purpose of the request and what the client expects as a successful result.

- `GET`: Requests a resource from the server.
  - The request has **no body**, but the response **tyically does**.
- `HEAD`: Requests the headers (only) that would by returned if it were a `GET` request.
  - Neither the request **nor** the response has a body.
- `POST`: Sends data to the server, often to create or update a resource.
  - Both the request **and** the response **typically** have a body.
- `CONNECT`: Establishes a **tunnel** to a server, typically used for **HTTPS over a proxy**. The client sends `CONNECT`, and if successful, raw data flows between the client and the server.
  - Neither the request **nor** the response has a body.

Request Target Forms

The **request target** specifies the resource on which the method should be applied. The form of the request target varies depending on the HTTP method and whether the request is sent through a proxy.

- **Origin-Form**
  - Used in most requests sent directly to an origin server (e.g., when a client connects to a web server).
  - The request target consists of just the path and optional query string.
    - If the path component is empty, the client **must** send `/` as the path within the origin-form of a request-target.

  Example:

```
GET /index.html?query=123 HTTP/1.1
```

- **Absolute-Form**
  - Used when a request is sent through a proxy (except for `CONNECT` requests).
  - The request target includes the scheme (`http`), hostname, optional port, path, and query string.
    - If a port is omitted, it defaults to port `80` for `http`.

  Example:

  ```
  HEAD http://example.com:8080/index.html?query=123 HTTP/1.1
  ```

- **Authority-Form**
  - Used exclusively by the `CONNECT` method when requesting a proxy to establish a tunnel to a remote server.
  - The request target consists of only the hostname and port (without a scheme or path).
    - The port **must** be included explicitly. If omitted, the request is invalid.

  Example:

  ```
  CONNECT example.com:443 HTTP/1.1
  ```

Response Status Codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped into five classes:

1. Informational responses (100 – 199) - ***Note**: your proxy will **not** encounter these during marking*
   - The request was received but still being processed.
2. Successful responses (200 – 299)
   - The request was successfully received, understood, and accepted.
3. Redirection responses (300 – 399)
   - Further action needs to be taken in order to complete the request.
4. Client error responses (400 – 499)
   - The request contains bad syntax or cannot be fulfilled.
5. Server error responses (500 – 599)
   - The server failed to fulfill an apparently valid request.

Header Fields

Header fields provide metadata about an HTTP message. They follow this format, where the field name and field value are colon separated:

```
field-name: field-value
```

Field names are case-**in**sensitive, and there should be no leading or trailing whitespace.

Field values themselves do not have any leading or trailing whitespace, however a field value may be surrounded by optional whitespace.

Some header fields may be multi-valued, as an ordered, comma-separated list:

```
field-name: field-value-1, field-value-2, ..., field-value-n
```

As part of this assignment, before forwarding a message, your proxy will need to identify, and potentially remove, modify, or insert, particular header fields. These fields are briefly described here. Any other fields that may be present should simply be forwarded unchanged.

1. **`Connection` / `Proxy-Connection`**

   The `Connection` header allows the sender to list desired control options for the current connection. Most notably, it controls whether the network connection stays open (i.e. **persists**) after the current transaction finishes.

   If a sender does not intend to keep the connection open beyond the current request-response exchange, then they must specify a `close` directive:

   ```
   Connection: close
   ```

   Otherwise, they *may* specify a list of desired control options, for example:

   ```
   Connection: keep-alive, upgrade
   ```

   In HTTP/1.1, connections are persistent by default, meaning unless there's an explicit `close` directive in the message, the connection *should* remain open, regardless of whether there is a `keep-alive` directive.

   Some legacy user agents may send a non-standard `Proxy-Connection` header instead of or in addition to `Connection`, attempting to control connection persistence with proxies. This is not part of the official HTTP specification, but proxies should handle both headers appropriately. Effectively, they should be regarded as equivalent, except where both are present, in which case `Connection` takes precedence.

   Note, the header values, like the header name, are **not** case sensitive.

2. **`Transfer-Encoding`**

The `Transfer-Encoding` header lists any transformations that have been applied to the content in order to form the message body.

For example, the following header indicates that the content has been compressed in `gzip` format and that it will be sent as a series of chunks:

```
Transfer-Encoding: gzip, chunked
```

Chunked transfer encoding allows data to be sent in multiple parts, each prefixed with its size, without knowing the total size in advance.

If a message is received with both a `Transfer-Encoding` and a `Content-Length` header, the `Transfer-Encoding` overrides the `Content-Length`.

Note, the header values, like the header name, are **not** case sensitive.

3. **`Content-Length`**

   If a message doesn't have a `Transfer-Encoding` header, the `Content-Length` header can specify the expected size of the content in **bytes**.

   When a message includes content, `Content-Length` can help determine where the data and message end.

   For messages without content, it indicates the size of the selected representation. For example, a response to a `HEAD` request does not include content, but may include a `Content-Length`.

4. **`Via`**

   The `Via` header is used for tracking message forwards. The syntax is a comma-separated list of `<protocol-version> <pseudonym>` pairs.

   For example, the following header indicates that the message passed through an intermediary that supports HTTP/1.0 with the pseudonym `foo`, then an intermediary that supports HTTP/1.1 with the pseudonym `bar`.

   ```
   Via: 1.0 foo, 1.1 bar
   ```

Message Body

The message body (if any) is used to carry content for the request or response. The message body is identical to the content unless a transfer coding has been applied.

The rules for determining when a message body is present in a message differ for requests and responses:

- **Requests**: The presence of a message body is signaled by a `Content-Length` header field.

- **Responses**: The presence of a message body depends on both the request method to which it is responding and the response status code.

The length of a message body is determined by one of the following (in order of precedence):

1. Any response to a `HEAD` request and any response with a `204` (No Content) or `304` (Not Modified) status code is always terminated by the first empty line after the header fields, regardless of the header fields present in the message, and thus cannot contain a message body.
2. If a `Transfer-Encoding` header field is present in a response, the message body length is determined by reading the connection until it is closed by the server.
    - ***Note**: this is a significant deviation from HTTP/1.1 standards, made for the purpose of simplifying the assignment.*
3. If a `Content-Length` header field is present without `Transfer-Encoding`, its decimal value defines the expected message body length in bytes. If the sender closes the connection or the recipient times out before the indicated number of bytes are received, the recipient **must** consider the message to be incomplete and close the connection.
4. If this is a request message and none of the above are true, then the message body length is zero (no message body is present).
5. Otherwise, this is a response message without a declared message body length, so the message body length is determined by the number of bytes received prior to the server closing the connection.

Message Parsing

As a general guide, when parsing an HTTP message, the typical approach is:

1. Read the start-line into a structured representation.
2. Read each header field line into a hash table (keyed by field name) or similar, until encountering an empty line.
3. Use the parsed data to determine whether a message body is expected.
4. If a message body is indicated, read it as a stream, either for the specified length or until the connection is closed.

Requirements & Features

Basic Non-Persistent Proxy

In the simplest case, your proxy will be non-persistent and handle requests sequentially, supporting `GET`, `HEAD`, and `POST`. While it **must** accommodate multiple client connections over time, it only needs to handle one connection at a time, with each connection limited to a single request-response exchange.

This is illustrated in the following example of a *typical* non-persistent transaction, starting with Figure 3.

***Important reminder**: All header field names, and all header field values that we are dealing with directly, should be treated as case-**in**sensitive.*

1. **Client-Proxy Connection**
    - The client initiates a connection with the proxy, which the proxy accepts.
2. **Client Request**
    - The proxy should start receiving and parsing the request message, for example:
        - Parse the start line to extract the method, request target, and protocol version.
        - Parse the *absolute-form* request target to extract the hostname, optional port, and *origin-form* path + optional query.
            - In this case, no port is given, so it defaults to port `80`.
        - Read each header field line into a data structure, until encountering an empty line, marking the end of the header section.
        - Use the parsed data to determine whether a message body is expected and if more data may need to be read.
            - In this case, the message is a request and the method is `GET`, so no body is expected and the message is considered complete.
    - The proxy should then transform the message for forwarding, in particular:
        - Replace the request target with its *origin-form*.
        - Replace or insert any `Connection` header with `Connection: close`.
        - Remove any `Proxy-Connection` header.
        - Insert or append a `Via` header with `1.1 [your zID]`.
3. **Proxy-Server Connection**
    - The proxy initiates a connection with the origin server specified in the request target, which the origin server accepts.
4. **Client Request Forwarding**
    - The proxy sends the transformed request to the origin server.

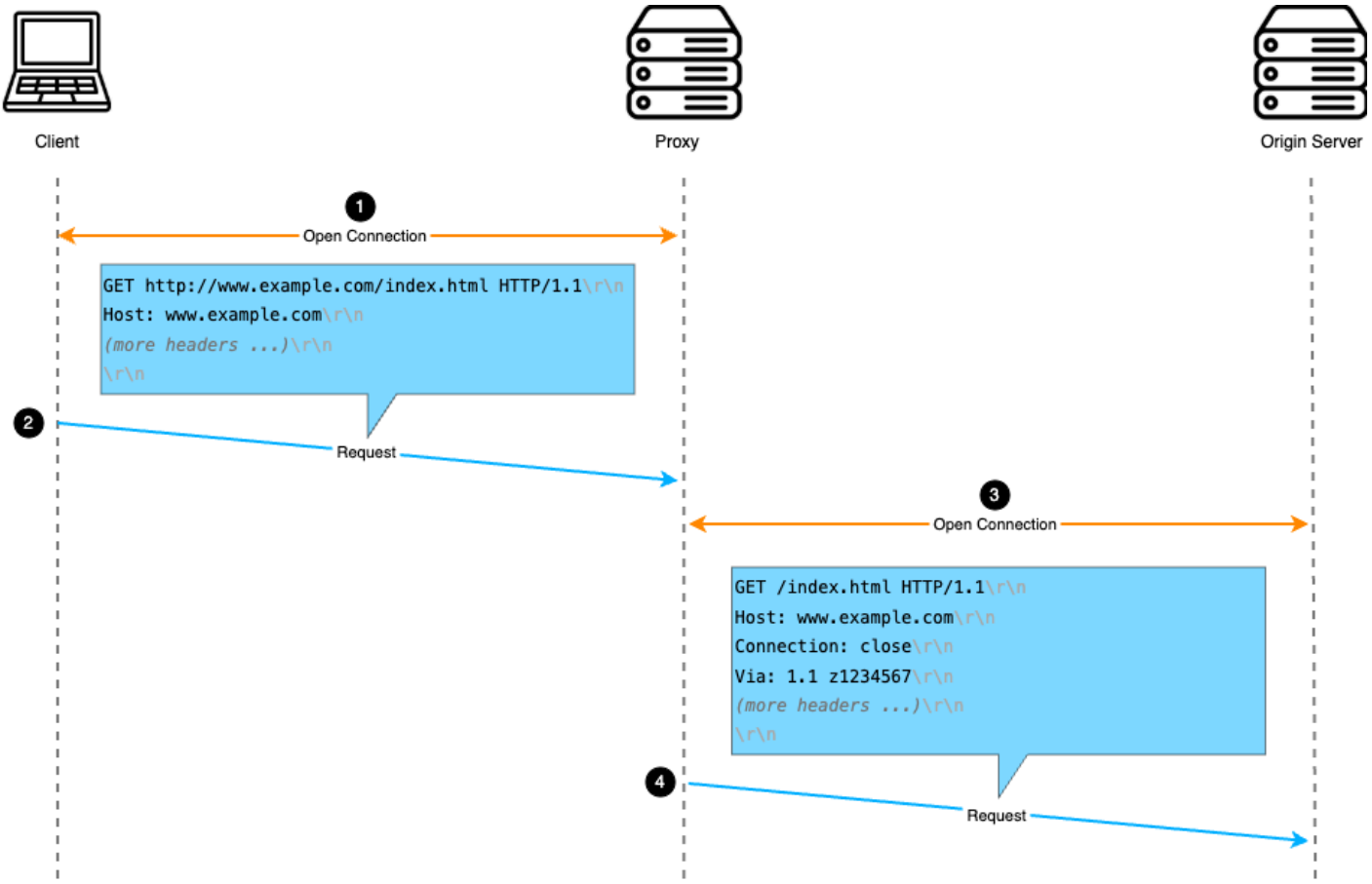


Figure 3: A client sending a GET request via a proxy

5. **Server Response**
    - The proxy should start receiving and parsing the response message, for example:
        - Parse the start line to extract the protocol version, status code, and optional reason phrase.
        - Read each header field line into a a data structure, until encountering an empty line.
        - Determine whether a message body is expected. In this case, the message is in response to a `GET` request and has a status code of `200`, therefore a body is expected.
        - Determine that no `Transfer-Encoding` has been specified, but a `Content-Length` of 1256 bytes is indicated, so continue to read data if and as necessary until the full body is received.
6. **Proxy-Server Connection Termination**
    - The proxy closes the connection with the server.
7. **Server Response Forwarding**
    - The proxy should then transform the message for forwarding, in particular:
        - If necessary, replace or insert any `Connection` header with `Connection: close`.
        - Insert or append a `Via` header with `1.1 [your zID]`.
    - The proxy sends the transformed response to the client.
8. **Client-Proxy Connection Termination**
    - The proxy closes the connection with the client and listens for the next connection request.

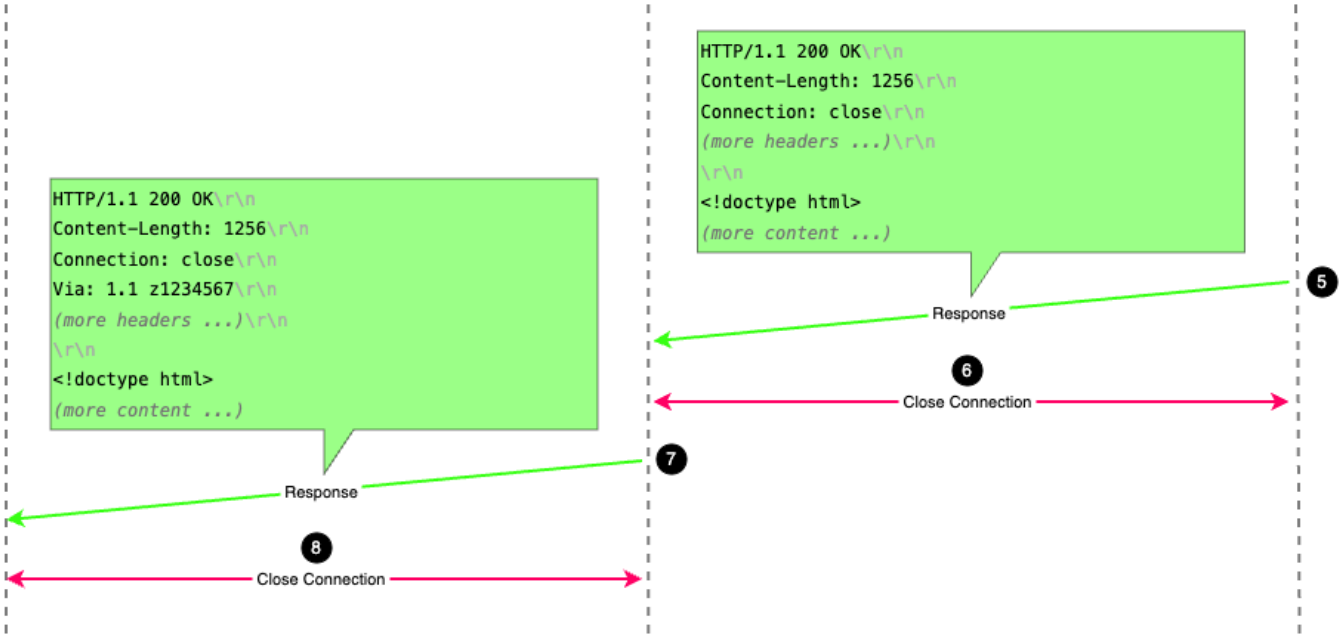


Figure 4: A server sending a response to a GET request via a non-persistent proxy

`HEAD` and `POST` requests follow quite similarly in the usual case, just accounting for potential differences in Message Body.

A basic test to perform a simple sanity check of your proxy is provided.

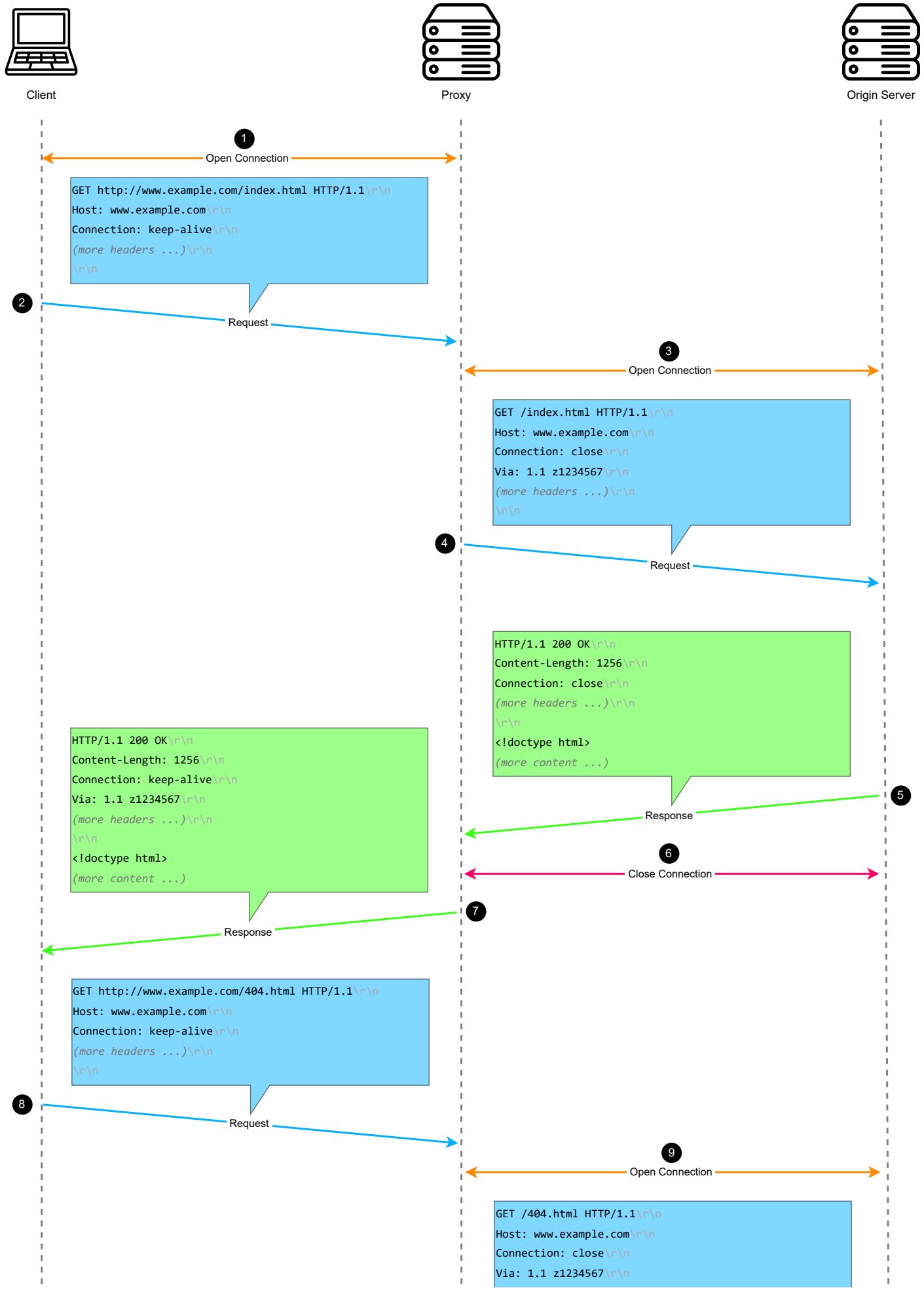
Basic Persistent Proxy

In this case, your proxy builds on the basic non-persistent proxy by allowing a client connection to be reused for multiple requests, which once again may include any combination of `GET`, `HEAD`, or `POST` requests.

This is illustrated in the Figure 5 example of a *typical* persistent session. It mirrors the basic non-persistent proxy example, with the minor differences noted below.

***Important reminder**: We do **not** expect your proxy to maintain persistent server connections.*

- In step 2, when the proxy receives the request, it also records whether the client intends for the connection to persist based on the explicit inclusion or absence of a `Connection: close` header.
- In step 7, when the proxy sends the response to the client, the `Connection` header needs to be set according to the client preference.
- In step 8, assuming the client preference is for the connection to persist, the connection remains open for the proxy to receive a subsequent request.
- The request-response process is repeated, potentially any number of times, until step 14, when the connection is finally closed. This may occur for a number of reasons. For example, because the client has indicated their intention in the previous request to `close` the `Connection`, the client has timed out the proxy, or the proxy has timed out the client.


Client
Proxy
Origin Server
1
Open Connection
GET http://www.example.com/index.html HTTP/1.1\r\n
Host: www.example.com\r\n
Connection: keep-alive\r\n
(more headers ...)\r\n
\r\n
2
Request
3
Open Connection
GET /index.html HTTP/1.1\r\n
Host: www.example.com\r\n
Connection: close\r\n
Via: 1.1 z1234567\r\n
(more headers ...)\r\n
\r\n
4
Request
HTTP/1.1 200 OK\r\n
Content-Length: 1256\r\n
Connection: close\r\n
(more headers ...)\r\n
\r\n
<!doctype html>
(more content ...)
5
Response
HTTP/1.1 200 OK\r\n
Content-Length: 1256\r\n
Connection: keep-alive\r\n
Via: 1.1 z1234567\r\n
(more headers ...)\r\n
\r\n
<!doctype html>
(more content ...)
6
Close Connection
7
Response
GET http://www.example.com/404.html HTTP/1.1\r\n
Host: www.example.com\r\n
Connection: keep-alive\r\n
(more headers ...)\r\n
\r\n
8
Request
9
Open Connection
GET /404.html HTTP/1.1\r\n
Host: www.example.com\r\n
Connection: close\r\n
Via: 1.1 z1234567\r\n

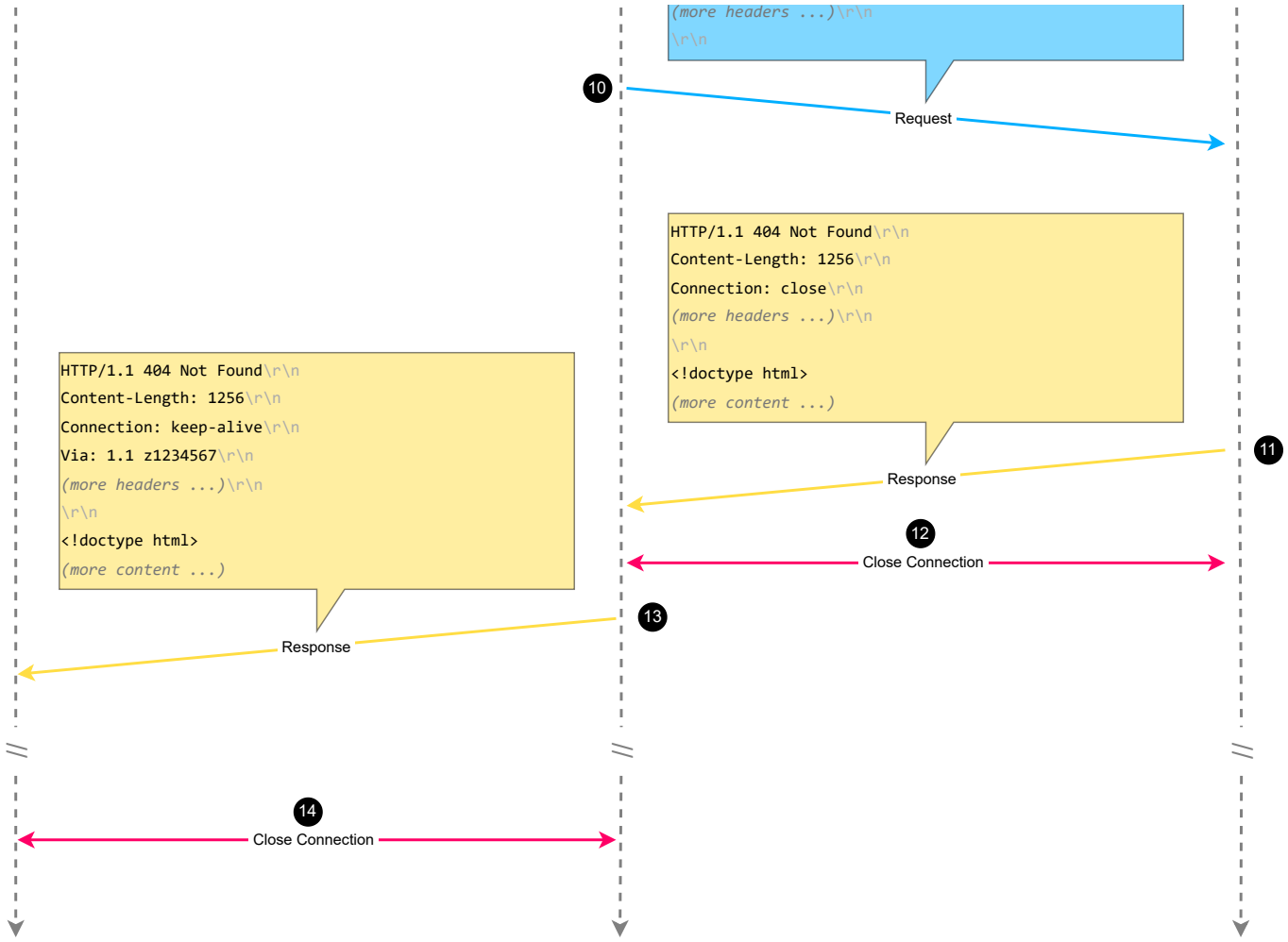


Figure 5: A client sending multiple GET requests via a persistent proxy

---

HTTP Tunneling with `CONNECT`

At this point your proxy is quite functional, but it's limited to HTTP traffic, which is increasingly rare as most websites and services have transitioned to HTTPS for security and privacy.

However, an HTTP proxy implementing the `CONNECT` method can handle HTTPS traffic by tunneling encrypted connections, allowing it to support modern web applications securely.

Consider the *typical* example in Figure 6:

1. **Client-Proxy Connection**
    - The client initiates a connection with the proxy, which the proxy accepts.
2. **Client Request**
    - The proxy should start receiving and parsing the request message, as previously described, but in this case:
        - Since the method is `CONNECT`, the request target is expected in *authority-form*.
        - Since the method is `CONNECT`, no body is expected.
    - The proxy does not forward this message and can ignore all headers, but the proxy does need to validate that the request target explicitly includes a port, and that the port is `443`.
3. **Proxy-Server Connection**

- The proxy initiates a connection with the origin server specified in the request target, which the origin server accepts.
  - Note: the proxy does **not** send any HTTP messages to the server.

4. **Proxy Response**
   - The proxy sends an `HTTP/1.1 200 Connection Established` response to the client, with no additional headers and no body.
5. **Raw Data Relay**
   - The proxy starts blindly forwarding data in both directions. Any data it receives on one socket, it writes to the other socket, until...
6. **Connection Termination**
   - Some time later, when one of the endpoints closes its connection, the proxy ensures any outstanding data is written and gracefully closes both sockets.

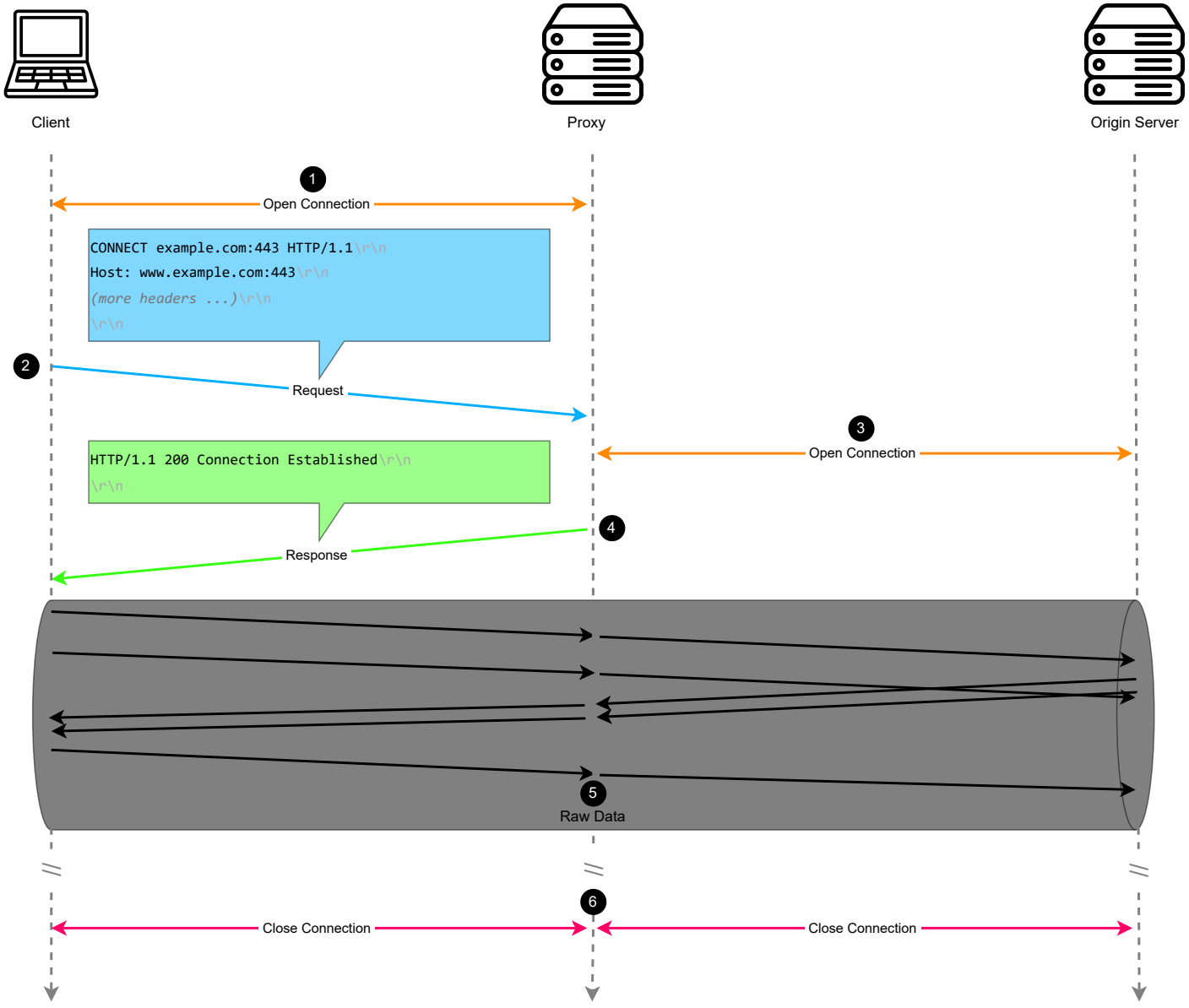


Figure 6: A client sending a CONNECT request to a proxy

When implementing support for the `CONNECT` method, the proxy must be able to **blindly forward traffic in both directions** without any awareness of message framing. This means that once the tunnel is established, the proxy

simply relays raw data between the client and the server, without interpreting or modifying it. A **naïve** single-threaded approach that reads from one socket and then writes to the other **won't work reliably**, as it may block indefinitely if one side isn't actively sending data. This can lead to deadlocks where neither endpoint is receiving data because the proxy is stuck waiting on a read operation. To avoid this, implementations typically use **multi-threading, non-blocking I/O (e.g., `select()`, `poll()`), or asynchronous event loops** to ensure data is forwarded as soon as it's available on either side.

Error Handling

Connections may close unexpectedly at any time, whether due to network conditions or intentional termination. Your proxy **must** handle such events gracefully to ensure robustness. Failure to do so may cause it to break under even basic test scenarios.

Additionally, your proxy should handle explicit error conditions as outlined below. In most cases, it should generate and send an HTTP response with the appropriate status code and reason phrase. The response must include a representation (i.e., a response body) providing some minimal information about the error, including the mentioned required phrases. You may use any text-based format, such as `text/html` or `text/plain`, but ensure the response adheres to that format and includes properly set `Content-Type` and `Content-Length` headers.

Explicit Error Conditions

- **Handling Client Timeouts**
  - If a client is idle for `timeout` seconds, the proxy should gracefully close the socket and move on.
    - Response: Not applicable
- **Malformed `CONNECT` request**
  - If no port or a port other than `443` is specified:
    - Response: `400 Bad Request` with "`invalid port`" mentioned in the body.
- **Invalid Request Target**
  - If the request target is missing a host:
    - Response: `400 Bad Request` with "`no host`" mentioned in the body.
  - If the host/port in the request target points to the proxy itself (self-loop detection):
    - Response: `421 Misdirected Request` with "`proxy address`" mentioned in the body.
- **Connection Issues**
  - If the origin server refuses the connection:
    - Response: `502 Bad Gateway` with "`connection refused`" mentioned in the body.
  - If the origin server cannot be resolved via DNS:
    - Response: `502 Bad Gateway` with "`could not resolve`" mentioned in the body.
  - If the origin server closes the connection unexpectedly:
    - Response: `502 Bad Gateway` with "`closed unexpectedly`" mentioned in the body.

        - Note: A response is **not** required if this happens during HTTP tunneling.
    - If the origin server fails to respond within `timeout` seconds:
        - Response: `504 Gateway Timeout` with "`timed out`" mentioned in the body.
        - Note: A response is **not** required if this happens during HTTP tunneling.

Caching

Your proxy should implement an in-memory cache that stores cacheable responses. This is to reduce the response time and network bandwidth consumption on future equivalent requests.

Parameters for the cache are provided as command-line arguments to the proxy, specifically:

- `max_object_size` in bytes, a strictly positive integer.
- `max_cache_size` in bytes, an integer at least equal to `max_object_size`.

When considering the object or cache size, your proxy must only count bytes used to store the actual objects. Metadata, such as response headers and timestamps, will also need to be stored, but for the purposes of these calculations should be ignored.

We take a very simplified view of HTTP caching, as outlined in this section, ignoring all cache directives that may be included in messages.

A response is only cacheable if it is:

- received in response to a `GET` request;
- has a status code of `200`; and
- the object is no greater than `max_object_size`.

If caching such a response would cause the cache to exceed `max_cache_size`, then one or more responses must first be evicted using a least recently used (LRU) replacement policy. No more responses should be evicted than necessary.

Upon receiving a `GET` request, your proxy should first check its cache to see if the request can be satisfied by a stored response. If it can, a "cache hit" occurs, and the proxy should respond directly using the stored response. Otherwise, a "cache miss" occurs, and the proxy should forward the request to the origin server.

The normalised target URL of a `GET` request is the "cache key", which is used to identify a stored response. Normalisation of the target URL considers the following rules:

- No port is equivalent to the default port (80).
- No path is equivalent to "/".
- The scheme and host are case insensitive.
- All other components are case sensitive.

For example, all of the following URLs are equivalent:

```
http://example.com
http://example.com/
HTTP://EXAMPLE.COM/
HTTP://EXAMPLE.COM:80/
HTTP://EXAMPLE.COM:0080/
```

While none of the following URLs are equivalent:

```
http://example.com/
http://example.com/index.html
http://example.com/INDEX.HTML
http://example.com:8080/INDEX.HTML
http://example.com:8080/INDEX.HTML?FOO=BAR
```

Request methods other than `GET` should simply bypass the cache.

If your proxy supports concurrency, then you should be particularly careful about how it interacts with any shared data structures.

Note, marking will rely entirely on your logging to assess the operation of your cache. If your proxy does not produce a log, then it will not be possible to award any marks for caching.

Logging

Your proxy should write to standard output a log of each HTTP transaction as it's completed. The log should follow a variant of the Common Log Format, with the following syntax:

```
host port cache date request status bytes
```

Where:

- `host`: is the IP address of the client which made the request.
- `port`: is the port of the client which made the request.
- `cache`: is `-` for all requests other than `GET`, in which case it is either `H` for a cache hit or `M` for a cache miss.
- `date`: is the date, time, and time zone that the request was received, in `strftime` format `%d/%b/%Y:%H:%M:%S %z`, enclosed in square brackets. Please note that if your proxy supports concurrency, then transactions may appear to be logged out of order and is to be expected.
- `request`: is the request line from the client, as received by the proxy, enclosed in double quotation marks.
- `status`: is the HTTP status code returned to the client.
- `bytes`: is the size of the object returned to the client, not including the response headers.

Here is an example:

```
127.0.0.1 56150 - [22/May/2025:10:17:24 +1000] "HEAD http://www.example.org/
HTTP/1.1" 200 0
127.0.0.1 56149 M [22/May/2025:10:17:24 +1000] "GET http://www.example.org/
HTTP/1.1" 200 648
127.0.0.1 56154 - [22/May/2025:10:17:24 +1000] "POST http://httpbin.org/post
HTTP/1.1" 200 480
127.0.0.1 56150 H [22/May/2025:10:17:25 +1000] "GET http://www.example.org/
HTTP/1.1" 200 648
127.0.0.1 56156 - [22/May/2025:10:17:25 +1000] "CONNECT api.github.com:443 HTTP/1.1"
200 0
127.0.0.1 56149 M [22/May/2025:10:17:24 +1000] "GET http://httpstat.us/404?
sleep=5000 HTTP/1.1" 404 13
```

A basic test to perform a simple sanity check of your log format is provided.

Concurrency

Extend your proxy to handle multiple client connections concurrently. This means your proxy should be able to process requests from multiple clients at the same time, rather than handling them sequentially. You may use threading, multiprocessing, or asynchronous I/O to achieve this. Ensure proper synchronisation when accessing any shared resources, such as the cache, to prevent race conditions.

***Reminder**: You **must not** use any ready-made server or HTTP libraries to implement any aspects of this assignment.*

Report

Submit a short report (no more than 3 pages) named `report.pdf`. The report should cover the following sections:

1. **Programming Language and Code Organisation**
    - State the programming language used (i.e., C, Java, or Python).
    - Briefly describe your code structure (e.g., key directories, files like `Makefile`).
2. **High-Level Design**
    - Provide a brief overview of your program's main components and their interactions.
    - You may include a simple diagram if it helps clarify the design.
3. **Data Structures**
    - Summarise the key data structures used in your proxy (e.g., for HTTP message handling, caching).
    - Focus on the most important structures and how they support your design.
4. **Limitations**

- Mention any known limitations of your implementation (e.g., incomplete feature support, specific conditions where the program might fail).

5. **Acknowledgments**

   - Indicate any external code or resources you've used, with links or brief citations.

Tips on Getting Started

This is a complex assignment, and the best way to tackle a complex task is to start early and to do it in stages.

Before attempting this asssignment, we advise that you read this specification and the FAQ in full, more than once, and finish Lab 2 and Lab 3.

Lab 3, in particular, may serve as an excellent starting point for this assignment.

1. **Understand Socket Basics**

   - Understand the difference between TCP (stream-oriented, reliable) and UDP (datagram-oriented, best-effort).
   - Know the roles of client and server sockets in a TCP connection and the purpose of API calls such as:
     - `socket()` – Creating a socket with appropriate domain (`AF_INET` for IPv4) and type (`SOCK_STREAM` for TCP).
     - `bind()` – Binding a socket to an IP address and port (for servers).
     - `listen()` – Marking a socket as passive to accept incoming connections.
     - `accept()` – Accept a new client connection.
     - `connect()` – Initiating a connection from the client to the server.
   - TCP is **stream-oriented**—data can be split across multiple `recv()` calls.
   - The need for **handling partial reads/writes**, ensuring all expected data is sent and received.

2. **Understand HTTP/1.1 Basics**

   - Make sure you have a good grasp of HTTP/1.1, particularly connection management, request/response structures, and important headers like `Connection`.
   - Familiarise yourself with how HTTP proxies work within this framework.

3. **Data Structure Design**

   - Spend some time thinking about your data structures.
   - Designing good abstractions from the outset for things like HTTP messages and your cache will greatly simplify the proxy logic itself.

4. **Plan and Document**

   - Draw a **high-level architecture diagram** to visualise how the components will interact.
   - Organise your code with clear separation of concerns (e.g., separate modules for message parsing, caching, logging, etc.). Keep your code clean and document important design decisions.

5. **Break Down the Problem**
    - Tackle the assignment in small steps, for example:
        - First, focus on message parsing and basic proxy behavior for a single method like `GET`, before moving to `HEAD` and `POST`.
        - Next, implement persistence, caching, and `CONNECT`.
        - Finally, add concurrency handling once the foundation is solid.
6. **Start with Logging**
    - **Implement logging early** to help with debugging and tracking the flow of requests and responses. Log all incoming requests, outgoing requests to the origin server, and the responses from the server.
    - Keep the log format simple at first (e.g., log request details, response status, and any errors) so that you can easily trace issues during development, but eventially make sure logging conforms to the expected format.
7. **Handle Errors Early and Gracefully**
    - Implement error handling **from the beginning** to prevent unexpected crashes.
    - Handle **timeouts**, **closed sockets**, **failed connections**, and **invalid HTTP requests** gracefully.
8. **Handle Basic Proxy Functionality**
    - Begin with the **core proxy functionality**: parsing HTTP requests, forwarding them to the origin server, and sending the server's response back to the client.
    - This step will help you verify that the basic mechanics of the proxy are working before adding complexity.
9. **Implement Client-Proxy Persistence**
    - After the basic proxy is functioning, handle the **client-proxy connection persistence** based on the client's `Connection` header.
    - If the client requests a persistent connection, ensure that you respect this preference and keep the connection open as required.
10. **Introduce Caching**
    - Implement a **basic cache** that stores responses from the origin server. Start by focusing on caching all responses and later introduce the notion of cacheable responses and the LRU replacement policy.
    - Log the cache hits and misses so you can monitor whether your caching mechanism is working as expected.
11. **Test and Debug Incrementally**
    - Regularly test your proxy with **real HTTP requests** (using tools like those suggested) to verify functionality and correctness.
    - Use your logging to track issues and help identify where things go wrong, especially with caching or connection persistence.

12. **Focus on Concurrency Later**
    - Once the basic functionality (logging, client-proxy persistence, caching) is working as expected, **tackle concurrency**.
    - You can then start adding **multithreading** or an event-driven model to handle multiple client connections concurrently. Focus on ensuring correctness first—concurrency can often add complexity that's easier to handle once your program works in a single-threaded fashion.
13. **Read the Requirements Carefully**
    - Refer back to the assignment specifications regularly to ensure you're meeting all the requirements and haven't missed any crucial details.

Testing and Debugging

HTTP is a ubiquitous protocol, so fortunately there are many tools and services that you can use to debug and test your proxy. Simply bypassing your proxy also gives you a convenient mechanism to determine the expected response for a given request.

Some tools and services are outlined in Useful Tools for Testing and Debugging. Marking will utilise similar tools. During development you should endeavour to use multiple user agents and communicate with as many origin servers as possible.

It is imperative that you rigorously test your code to ensure that all possible (and logical) interactions can be correctly executed. **Test, test, and test**.

Submission

Please ensure that you use the mandated file names of `report.pdf` and, for the entry point of your application, one of:

- `proxy.c`
- `proxy.py`
- `Proxy.java`

If you are using C or Java, then you **must** additionally submit a `Makefile`. This is because we need to know how to resolve any dependencies. See Sample Client-Server Programs and Networking Programming Resources for a guide on writing a `Makefile`.

After running `make`, we should have one of the following executable files:

- `proxy` (for C)
- `Proxy.class` (for Java)

Submission is via `give` using the following command syntax:

```
$ give cs3331 assign <file1> [<file2> ... <fileN>]
```

Note, this is the same command for both COMP3331 and COMP9331 students.

If your codebase does not rely on a directory structure then you may submit the files directly. For example, assuming your implementation is in C, and you additionally have `helper.c` and `helper.h` files that your `Makefile` expects to find in the same directory as `proxy.c`:

```
$ give cs3331 assign report.pdf Makefile proxy.c helper.c helper.h
```

If your codebase relies on some directory structure, for example you've created helper functions or classes in a sub-directory to your main program, you **must** first `tar` the parent directory as `assign.tar`. For instance, assuming a directory `assign` contains all the relevant files and sub-directories (including your report), open a terminal and navigate to the parent directory, then execute:

```
$ tar -cvf assign.tar assign
$ give cs3331 assign assign.tar
```

Please **do not** submit any build artefacts, test files/programs, or other particulars that are not required to compile and run your application.

Upon running `give`, ensure that your submission is accepted. You may submit often. Only your last submission will be marked.

Emailing your code to course staff **will not** be considered as a submission.

Submitting the wrong files, failing to submit certain files, failing to complete the submission process, or simply failing to submit, **will not** be considered as grounds for re-assessment.

If you wish to validate your submission, you may execute:

```
$ 3331 classrun -check assign  # show submission status
$ 3331 classrun -fetch assign  # fetch most recent submission
```

Important: It is your responsibility to ensure that your submission is accepted, and that your submission is what you intend to have assessed. **No exceptions**.

Late Submission Policy

Late submissions will incur a 5% per day penalty, for up to 5 days, calculated on the achieved mark. Each day starts from the deadline and accrues every 24 hours.

For example, an assignment otherwise assessed as 12/20, submitted 49 hours late, will incur a 3 day x 5% = 15% penalty, applied to 12, and be awarded 12 x 0.85 = 10.2/20.

Submissions after 5 days from the deadline will not be accepted unless an extension has been granted, as detailed in Special Consideration and Equitable Learning Services.

Special Consideration and Equitable Learning Services

Applications for Special Consideration **must** be submitted to the university via the Special Consideration portal. Course staff **do not** accept or approve special consideration requests.

Students who are registered with Equitable Learning Services **must** email cs3331@cse.unsw.edu.au to request any adjustments based on their Equitable Learning Plan.

Any requested and approved extensions will defer late penalties and submission closure. For example, a student who has been approved for a 3 day extension, will not incur any late penalties until 3 days after the standard deadline, and will be able to submit up to 8 days after the standard deadline.

Plagiarism

Group submissions **will not** be allowed. Your programs **must be** entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar assessments in previous years, if applicable) and serious penalties will be applied, including an entry on UNSW's plagiarism register.

You are not permitted to use code generated with the help of automatic tools such as GitHub Copilot, ChatGPT, Google Bard.

- **Do not copy ideas or code from others.**
- **Do not use a publicly accessible repository or allow anyone to see your code.**
- **Code generated by GitHub Copilot, ChatGPT, Google Bard and similar tools will be treated as plagiarism.**

Please refer to the online sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Plagiarism and Academic Integrity
- UNSW Plagiarism Policy
- UNSW Plagiarism Management Procedure

Marking Rubric

***Important Reminder**: Your proxy **must** compile and run within the CSE environment. Ensure it is thoroughly tested in that environment.*

Functionality	**Marks**
Basic Non-Persistent Proxy:	
- `GET`	2
- `HEAD`	1
- `POST`	1
Basic Persistent Proxy:	
- `GET` only	1
- `GET` + `HEAD` + `POST`	2
`Via` Header	1
`CONNECT`	2
Explicit Error Conditions	2
Logging	1
Caching	2
Non-Persistent Concurrency	1
Persistent Concurrency	1
Stress Test	1
Report	1
Code Quality	1
Total	**20**

No particular coding style is mandated, just ensure your code style is consistent, your code is clean, and your code is adequately documented.

⇧ Back to top