

# HTTP Caching Proxy

## Engineering Robust Server Software Homework 2

For this assignment you will be writing an `http proxy` – a `server` whose job it is to `forward requests to the origin server on behalf of the client`. Your proxy will `cache responses`, and, when appropriate, `respond with the cached copy of a resource rather than re-fetching it`.

While the HTTP specification is quite large (and includes many complex features), you should make an `http proxy` which functions with `GET, POST, and CONNECT`. (You MAY handle any of the other request methods if you want). Specifically, a `user should be able to configure their browser to use your proxy, and browse typical webpages` (e.g., perform a Google Search, view the results, etc). Note that many `webpages only do HTTPS`, so your `browser will use CONNECT to communicate with them` (and you won't see the actual GET requests).

Your proxy MUST `cache responses (when they are 200-OK) to GET requests`. You should follow the rules of expiration time and/or re-validation in determining if your proxy can serve a request from its local cache (versus re-fetching from the origin server). Other cache management policies (e.g., replacement policy) are up to you.

Your proxy MUST be able to `handle multiple concurrent requests effectively and SHOULD use multiple threads as part of your strategy to do so`. The remainder of the design of handling multiple requests is up to you, but your cache MUST be shared between all connections (and properly synchronized).

Your proxy MUST `produce a log (in /var/log/erss/proxy.log) which contains information about each request`. To keep the log understandable, your proxy will assign each request a unique identifier when it prints the first log message for that request.

In each of the following format descriptions `typewriter` text is literal, and *italics indicate a variable*.

- Upon receiving a `new request`, your proxy should assign it a unique id (*ID*), and print the ID, time received (*TIME*), IP address the request was received from (*IPFROM*) and the HTTP request line (*REQUEST*) of the request in the following format:

`ID: "REQUEST" from IPFROM @ TIME`

- If the request is a `GET` request, your proxy should `check its cache`, and print one of the following:

`ID: not in cache`

`ID: in cache, but expired at EXPIRETIME`

`ID: in cache, requires validation`

`ID: in cache, valid`

- If your proxy needs to **contact the origin server** about the request, it should print the request it makes to the origin server:  
**ID: Requesting "REQUEST" from SERVER**  
 Later, when it receives the response from the origin server, it should print:  
**ID: Received "RESPONSE" from SERVER**  
 Here, *REQUEST* and *RESPONSE* are the request line and response line (first line in the message), and *SERVER* is the server name.
- If your proxy **receives a 200-OK in response to a GET** request, it should print one of the following:  
**ID: not cacheable because REASON**  
**ID: cached, expires at EXPIRES**  
**ID: cached, but requires re-validation**
- Whenever your proxy **responds to the client**, it should **log**:  
**ID: Responding "RESPONSE"**  
 Where response is the response line of the reply. Note that you should do this if you reply with an error (e.g. if you receive a malformed request).
- When your proxy is handling a tunnel as a result of 200-OK, it should log (in addition to all other normal logging) when the tunnel closes with  
**ID: Tunnel closed**
- Your proxy MAY include any other log messages of your choice, as long as they adhere to the following formats:  
**ID: NOTE MESSAGE**  
**ID: WARNING MESSAGE**  
**ID: ERROR MESSAGE**  
 Where MESSAGE is text of your choice (not containing a new-line). You may use (no-id) as the ID if there is no id for one of these.

**All times should be printed in UTC**, with a format given by `asctime`. For example, your log might say:

```
104: "GET www.bbc.co.uk/ HTTP/1.1" from 1.2.3.4 @ Sun Jan 1 22:58:17
2017
104: not in cache
105: "GET www.duke.edu/foo/bar HTTP/1.1" from 11.12.42.40 @ Sun Jan 1
22:58:17 201
104: Requesting "GET www.bbc.co.uk/ HTTP/1.1" from www.bbc.co.uk
105: in cache, valid
105: Responding "HTTP/1.1 200 OK"
104: Received "HTTP/1.1 200 OK" from www.bbc.co.uk
104: NOTE Cache-Control: must-revalidate
104: NOTE ETag: W/"33bc8-F9Kn1zgYX0cHOaRFsmZORA"
104: cached, but requires re-validation
```

```
(no-id): NOTE evicted www.foo.bar.com/boring.txt from cache
104: Responding HTTP/1.1 200 OK
```

In addition to the functional requirements specified above, there are some other requirements for this assignment:

- You MUST implement your proxy in **C++**. Please make good use of OO design, RAII, exceptions, and other C++-concepts. Do NOT write C and call it C++.
- Your proxy MUST be **robust to external failures**. If it contacts the destination webserver and receives an error response, it must handle it gracefully. If it contacts the destination webserver and **receives a corrupted response**, it MUST reply with a **502 error code**. If the proxy receives a **malformed request**, it MUST reply with a **400 error code**.
- You SHOULD think carefully about the exception guarantees you make, and how you handle problems. You SHOULD document these in your code.
- You MUST provide a **docker-compose.yml file and a Dockerfile** (or more if appropriate) which allow your proxy to be run with `sudo docker-compose up`. The Docker/Docker-Compose setup that you create MUST do the following:
  - Connect the host computer's port 12345 to your proxy.
  - Mount a directory called logs (in the same directory as the docker-compose.yml file) to `/var/log/erss` in the container running your proxy. Given that your proxy will write its logs to `/var/log/erss/proxy.log` inside the container, this means you (and your TAs) should be able to find `proxy.log` in the logs directory on the host.
- You MUST provide a set of **testcases** which demonstrate the functionality of your proxy – both in the common case, and in the case where it **handles errors or unusual situations**. You may find **netcat** incredibly useful here as it will let you send malformed requests or responses. You may also find **wget** useful as it will generate standard web requests (and can print the request, giving you a starting point for a malformed request). We encourage you to provide an automated/self-contained setup to demonstrate these testcases. You might consider making use of something like **docker swarm** to show how your proxy handles many concurrent requests.