Alassane Ndiaye (260376319)
Etienne Perot (260377858)

COMP 535
Computer networks

COMP 535 - fetchy project report

The fetchy project (lowercase “f”) is an implementation of a non-transparent HTTP proxy written in Python. Its main objective is to minimize bandwidth usage between clients and itself by modifying web pages and resources as they pass through it, while attempting to keep performance reasonable on low-latency links. This objective is achieved through various means.

1. **Multi-level compressed caching**
For any proxy to be reasonably efficient, caching has to be used to delivery reasonably quickly repeated requests, to avoid most of the network overhead to re-fetch the requested resource, and the processing overhead to optimize this resource. This is especially crucial for fetchy, because it has high processing costs on the most common web request types (web pages, JS, CSS, and images).
fetchy therefore caches all its post-processed responses in a multi-level cache. The cache consists of two levels by default: The upper level is an in-memory cache of size 32 megabytes by default (configurable), and the lower level is a disk-based cache of size 128 megabytes. The cache implements a simple least-recently-used policy and concurrency control.
A notable property of the HTTP protocol is its “`Content-Encoding`” header, which allows content of any type to be delivered in various encodings. The most straightforward of them is the `identity` encoding, which simply involves sending the resource as is. However, another such encoding is `gzip`, which allows the resource to be compressed using gzip compression. This makes the total payload size smaller, but the cache can effectively take advantage of this property by running gzip compression over all its resources. This makes the in-memory and on-disk footprints of the cache smaller, and means we can send always-compressed data to the client with little overhead once the resource is cached. As a result, the cache implements storage not as simple (key, value) pairs, but as (key, pristine data, compressible data) tuples. When used to store HTTP responses, the pristine data holds the HTTP headers of the response, while the compressible data holds the actual contents of the response.
Whenever an element is inserted, its compressible data (if any) is compressed in the background (non-blocking). Upon retrieval from the cache, the data may or may not have finished compressing. If it hasn’t, the calling function may request whether to wait until the gzip compressing is complete or not. The response will then be served using whichever `Content-Encoding` is appropriate.
The caching module also supports *reservations*. This system allows external modules to reserve keys, announcing to the cache that values will be provided to these keys in the future. Whenever a cache lookup is executed on such reserved keys, the `lookup` call blocks until the reservation is satisfied or cancelled.
The cache only stores the results of `GET` requests; it never caches data from `POST` or `CONNECT` requests.

2. **Web page optimizations**
    a. *JavaScript optimizations*
    According to the HTTP Archive, JavaScript is the second largest and fastest-growing resource type in terms of file size on the web these days[1] (after images), and is relied on more and more to provide a rich user experience. Web sites have realized this and various JavaScript optimization techniques have come to light, such as Yahoo's YUI Compressor or Google's Closure Compiler.
    fetchy uses the offline version of Google's Closure Compiler to achieve JavaScript optimization, but is a little more aggressive about it. Here is the detailed process it performs:
        - All `<script>` tags from the page's body are **extracted out of the page**. There are two kinds of `<script>` tags: Regular scripts, which have the JavaScript code to run inside the tag itself, and external scripts, which have an `src` attribute pointing to the URL that the browser should request in order to get the JavaScript code it should execute.
        - The JavaScript optimizer goes through all the extracted tags, in the order they appear in the HTML, and computes the following checksum (implemented as an MD5 hash):
            - If a `<script>` is a regular script, the string "text" plus the script's contents are added to the checksum
            - If a `<script>` is an external script, the string "url" plus the script's absolute URL are added to the checksum.
        - The hash obtained is thus **unique** and **stable** to this set and order of script. This hash is called `key`.
        - The JavaScript optimizer then adds, just before the `</body>` closing tag of the page, the tag "`<script src="http://fetchy/`***`key`***`.js"></script>`".
        - The value of "***`key`***`.js`" is registered to fetchy as a *internal resource* and given a callback function `f`; this means that fetchy will now know that it needs to call `f` should the client request the URL `http://fetchy/`***`key`***`.js` (which it will).
        - The modified page HTML is sent as is to the next optimization stage (CSS); the JavaScript optimizer still has work to do, but it does all the following operations in a different thread.
        - The Closure Compiler, which Google offers to download for free as a `.jar` archive, is launched using `java` (a JVM needs to be present in the system's `PATH` environment variable).
        - The JavaScript optimizer loops over all the previously-extracted scripts:
            - If a `<script>` is a regular script, it simply writes its contents to the Closure Compiler's standard input stream.
            - If a `<script>` is an external script, it asks fetchy's HTTP client to stream the script URL's contents to the Closure Compiler's

[1] http://httparchive.org/trends.php?s=Top1000

standard input stream.
  - Once all scripts have been sent to the Closure Compiler, the final compiled script is retrieved from the Closure Compiler's standard output stream.
  - The callback function `f` passed to fetchy as an internal resource callback will be called eventually if the client requests the script. It is usually called before the Closure Compiler can finish, as the Closure Compiler can take a while to process large scripts. As such, the callback will block until the Closure Compiler exits. Once unblocked, the final compiled script is passed to the cache, where it is compressed using gzip compression as everything else, and finally sent to the browser.

This process is quite long and involved, and it doesn't work for all websites, as the only point the script executes is at the end of the page. This works fine for most websites, but fails whenever a `<script>` uses a position-dependent instruction such as `document.write()`. Thankfully, most `<script>`s of this nature tends to be advertisements.
Since, to the client, the final script's URL looks like `http://fetchy/`***`key`***`.js`, it is easily cacheable. The `key`'s uniqueness and stability properties help in this regard.
The Closure Compiler features 3 optimization levels: `WHITESPACE_ONLY`, `SIMPLE_OPTIMIZATIONS`, and `ADVANCED_OPTIMIZATIONS`. fetchy defaults to `SIMPLE_OPTIMIZATIONS`, which produces significantly smaller JavaScript code while keeping its external interface the same (doesn't rename public functions), making it usually the best choice for arbitrary-code optimization. The optimization level can be changed in the configuration file.
The `SIMPLE_OPTIMIZATIONS` level applies several non-destructive optimizations to the code, such as:

  - Removing all unnecessary whitespace
  - Removing `// comments` and `/* multiline comments */`
  - Renaming local variables to short names
  - Removing dead code
    `(if(something that is always false) { this is dead code })`
  - Removing unused variables
  - Replacing single-use variables by their definition
    ```
    var a = f1(4);
    a = a * 2 + 3;
    var b = f2(6);
    var c = a + b;
    return c;
    ```
    becomes
    ```
    return f1(4)*2+3+f2(6);
    ```
  - Compressing some statements:
    - `if(x) { a = true; } else { a = false; }` becomes

`a=x;`

- `if(x) { a = e1; } else { a = e2; }` becomes `a=x?e1:e2;`

b. *CSS optimizations*

CSS (Cascading Style Sheets), while a less resource-consuming resource type on the web, can still be improved significantly. HTML allows multiple stylesheets to be defined at any point in the page, despite the fact that each stylesheet applies to the entire page after being loaded. This is significant because it means that there is no loss in concatenating all stylesheets into one. In fact, a consolidated stylesheet is more efficient not just banwidth-wise, but processing-wise as well. Indeed, on each stylesheet loaded, the browser has to recompute a global ruleset applied to all the page; the more stylesheets, the more recomputations. Additionally, on each stylesheet loaded, the browser's rendering engine triggers a reflow[2] due to the potentially-changed style rules on all the page's elements. Reflows are expensive processor-wise and should be avoided when possible, which is the case here. This is why optimizing CSS is a worthwhile effort.

The CSS optimization process is similar to the JavaScript one.

- All `<style>` and `<link>` tags are *extracted out of the page*.
  Like JavaScript, CSS is pulled using two methods. The regular `<style>` tags contain CSS as their tag's body, while `<link>` tags have an `href` attribute referencing to an external CSS file.
- The optimizer goes through the previously mentioned tags, in the order they appear in the HTML, and computes an MD5 hash of this CSS combination, just like the JavaScript case:
  - When processing a `<style>` tag, the string "text" plus the script's contents are added to the checksum
  - When processing a `<link>` tag, the string "url" plus the script's absolute URL are added to the checksum.
- The hash obtained is **unique**, **stable** and is called `key`.
- The tag `<link href="http://fetchy/`***`key`***`.css"/>` is then added just before the the head closing tag.
- The value of "***key***.css" is registered to fetchy as a internal resource and given a callback function `f`.
- Further optimization is done using `cssmin` library. Optimizations include:
  - Remove unnecessary whitespace and semicolons
  - Remove `/* comments */`
  - Remove empty rules, e.g.
    ```
    body
    { }
    ```
  - Normalize RBG colors to hexadecimal values (`rgb(255, 255, 255)` becomes `#ffffff`, which then can be further simplified to

[2] http://code.google.com/speed/articles/reflow.html

        `#fff` (shortened hex form))
    - Condense zero units, multidimensional zeros, floating points, and hex colors:
      `0px` → `0`
      `border: 0px 0px 0px 0px` → `border: 0`
      `0.3` → `.3`
      `#FF00FF` → `#F0F`

   c. *HTML optimizations*
   Lastly, HTML itself can be optimized. That process takes the least amount of resources since unnecessary whitespace and comment removal are the only actions performed. However, significant lossless compression can be achieved since spaces, tabs and newlines are treated as a single space in HTML except in a few cases (`<pre>` tags, `<textarea>` tags, etc).

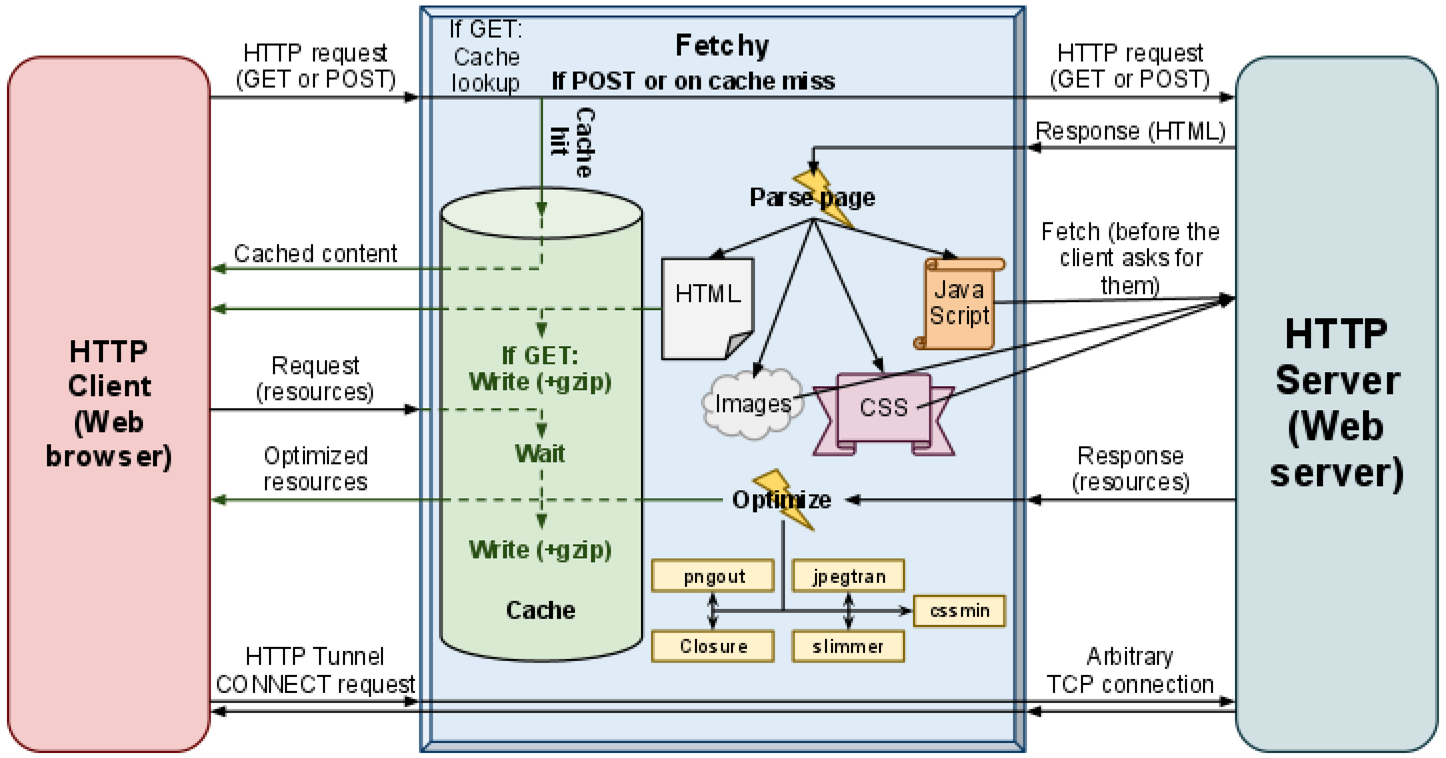
3. **Resource optimization**
   In addition to optimizing web pages, fetchy uses *resource prefetching* to speed itself up. This works by examining the contents of the HTML page and, for each external resource, start downloading it, optimizing it, and caching it in the background in anticipation for the client to request it later.
   a. *PNG images*
   There are various command-line tools available to losslessly reduce the file size of PNG images. The most well-known of such tools is `pngcrush`, but it is not the best regarding speed and compression efficiency. After testing, it was established that the free (but not open-source) `pngout` tool usually gives better compression ratios for reasonable amounts of time (usually less than a second for most images under 800x800 pixels). It provides multiple levels of compression aggressiveness, which can be modified in the configuration file.
   Whenever a page contains a PNG image, it is downloaded in the background, and fed to the `pngout` tool with the specified compression strategy. The `pngout` tool returns a file of size smaller or equal to the initial image, and this image is then passed on to the cache module where it will be gzip-compressed and ready for being requested by the client.
   To ensure that the client gets the optimized PNG image, the PNG optimizer reserves the image as soon as it is given the job to optimize it.
   b. *JPEG images*
   Similarly to PNG optimization tools, there exists command-line tools to losslessly reduce the file size of JPEG images. Unlike PNG, JPEG is a lossy image format; therefore, such tools do not guarantee losslessness (since the image isn't lossless in the first place), but guarantee that the resulting JPEG image will be as lossy as the original JPEG image was to the lossless version of the image.
   One such tool is the free `jpegtran` utility, distributes as part of `libjpeg-progs`. Like `pngout`, it usually runs in less than a second for most images, and is therefore viable to be used for optimizing all incoming JPEG images.
   Whenever a page contains a JPEG image, it is downloaded in the background,

and fed to the `jpegtran` tool. The rest of the process is the same as the PNG one; the resulting image is passed to the cache and then served to the client.

c. *BMP and TGA images*
Unlike the PNG and JPEG formats, which are both *compressed* image formats, the BMP and TGA formats are both *uncompressed* lossless formats. This makes them very unwieldy for usage on the web. As such, fetchy implements conversion of both the BMP and the TGA formats to the lossless PNG format. The result is a much lighter image, with no quality loss. The conversion is achieved using the Python Imaging Library (PIL), available for free but often distributed with Python itself. Whenever a page contains a BMP or TGA image, the `<img>` tag's `src` attribute is modified to replace the image's extension by ".png". The BMP or TGA is downloaded and converted in the background, and then passed on to the cache module where it will be gzip-compressed and ready for being requested by the client.

For completeness, fetchy also implements the HTTP `CONNECT` method, which allows tunneling of arbitrary TCP connections through an HTTP proxy. Notably, this allows SSL to pass through unhindered, so that https pages can still work as expected. However, since the communication is encrypted, there is no way for fetchy to examine the contents of web pages being requested. Therefore, it cannot implement its optimizations on such pages, nor is it a good idea to do so (secure, end-to-end communications should never be tampered with).

System overview diagram



Overall fetchy system diagram

Setup

This section describes how to run fetchy.

- **Strict dependencies**:
    - A Python 2.7 implementation. fetchy was only tested on CPython 2.7.2[3], available for free under the Python Software Foundation License for all platforms.

  fetchy has no strict dependency other than Python 2.7. However, most of its optimizations depend on external tools; therefore, using it without its optional dependencies will not yield good performance. If an optional dependency is not satisfied, the optimizations that require it will automatically be disabled.
- **Optional dependencies**:
    - Python Imaging Library (PIL)[4], available for free under the Python Imaging Library license[5]; sometimes bundled with Python installations.
      Used by fetchy to convert BMP and TGA images to PNG.
    - A JVM implementation such as the Oracle JVM[6] (available for free) or the OpenJDK HotSpot JVM[7] (available for free under the GPL). The main `java` binary must be available in the `PATH` environment variable.
      Used by fetchy to run Google's Closure Compiler.
    - The command-line `pngout` tool (available for free for Windows[8]; unofficial Mac OS X and Linux binaries are also available for free[9]). The `pngout` binary must be available in the **PATH** environment variable.
      Used by fetchy to optimize the file size of PNG images.
    - The command-line `jpegtran` tool (available for free for Windows and Unix[10]; often contained in Linux package repositories as part of the `libjpeg-progs` package).
      Used by fetchy to optimize the file size of JPEG images.

[3] http://www.python.org/download/releases/2.7.2/
[4] http://www.pythonware.com/products/pil/
[5] http://www.pythonware.com/products/pil/license.htm
[6] http://java.com/en/download/manual.jsp
[7] http://openjdk.java.net/groups/hotspot/
[8] http://advsys.net/ken/utils.htm
[9] http://www.jonof.id.au/kenutils
[10] http://jpegclub.org/jpegtran/

- **Internal dependencies**:
  fetchy depends on additional software, but the following are already included with fetchy itself. The end user should not need to download them, but they are included here for reference and completeness.
    - BeautifulSoup[11], a no-nonsense, real-world HTML parser written in pure Python available for free under the BSD license.
      Used by fetchy to parse (often poorly written) HTML.
      Note: The version distributed with fetchy has been patched according to Launchpad bug #686181[12].
    - chardet[13], a pure Python port of Mozilla's charset detector library[14], available for free under the LGPL.
      Used by BeautifulSoup to guess the character encoding used in web pages.
    - The Google Closure Compiler[15], available for free from Google as a `.jar` archive distributed under the Apache License 2.0.
      Used by fetchy to optimize JavaScript code.
    - cssmin[16], a pure Python port of Yahoo's YUI compressor for CSS only, available for free under the BSD license.
      Used by fetchy to optimize CSS code.
    - slimmer[17], a pure Python whitespace optimizer for CSS, HTML and XHTML, available for free under the Python Software Foundation License.
      Used by fetchy to optimize HTML (not CSS).
- **Configuration**:
  fetchy comes with a pre-populated configuration file called `config.py`. This file can be opened with a text editor and modified at will, as long as it follows the Python dictionary syntax[18].
  Accompanying each `'key': 'value'` pair is a `# comment` explaining the functionality and possible value of the setting.
  Any setting may be deleted from the file. In these cases, the setting will simply take its default value from `defaults.py` located within fetchy's source code.
  Settings include the listening port of the proxy (default: `9010`), the ability to adjust per-module verbosity, disabling certain optimizations, changing the cache sizes, etc.
- **Running**:
  To run fetchy, simply execute `./fetchy-run.py`.
  There is an additional script, `fetchy-parser-test.py`, that can be used as well to test

[11] http://www.crummy.com/software/BeautifulSoup/
[12] https://bugs.launchpad.net/beautifulsoup/+bug/686181
[13] http://pypi.python.org/pypi/chardet
[14] http://www-archive.mozilla.org/projects/intl/chardet.html
[15] http://code.google.com/closure/compiler/
[16] https://github.com/zacharyvoase/cssmin
[17] http://pypi.python.org/pypi/slimmer/0.1.30
[18] http://docs.python.org/tutorial/datastructures.html#dictionaries

single URLs using the web page minification engine. It is used as follows:

```
./fetchy-parser-test.py ‘URL1’ ‘URL2’ ‘URL3’ …
```

The script will output status and debug messages (resources to prefetch, internal resource registrations, etc) to the standard error stream, while the final HTML code will be written to the standard output stream. As such, if using this script with the purpose of testing the resource detection of the engine, it is often useful to redirect the standard output stream to `/dev/null`:

```
./fetchy-parser-test.py http://www.seamonkey-project.org/ > /dev/null
```

The standard output stream can also be redirected to a file to later be opened in a text editor. It is not recommended nor useful to open this file in a browser, since it contains fetchy-specific URLs of the form `http://fetchy/`***`key`***, which will not work without having been generated by the same fetchy instance.

- **Using**:
  To use fetchy, simply set your web browser’s HTTP proxy settings to the address fetchy is listening to (the port number is 9010 by default and may be modified in the configuration file), and browse the web as normal.

Results

fetchy has been tested on various websites around the web. Following is a list of websites along with their load times and bandwidth used, with and without fetchy.
All results were obtained using the SeaMonkey web browser[19], set to refuse all cookies and to not use any form of local caching. The fetchy proxy server was set up on an otherwise idle Linux machine that is 35 milliseconds (11 hops) away from the testing client machine. All fetchy settings were the default, except JavaScript optimizations turned off as hey can be quite slow on JavaScript-heavy websites, which the web’s top websites tend to be.
Load time was measured with the Firebug add-on[20] (rounded to centiseconds), while total page size was measured using the Web Developer add-on[21] (Information → View document size), which sums up the total bandwidth usage of all resources of a page, and shows compressed page size (gzip-compressed resources) and the equivalent in uncompressed page size. The table shows the compressed size, as it corresponds to bandwidth usage.
The page load also shows the speed of loading the page once it is in fetchy’s own cache (thereby not requiring any processing from fetchy). The difference between the regular fetchy load time and the “in cache” load time is the network overhead of fetchy to reach the real web server, plus the processing time on the web content. The client’s connection was a 30 Mbps link using Vidéotron as ISP.

[19] http://www.seamonkey-project.org/
[20] http://getfirebug.com/
[21] https://addons.mozilla.org/en-US/firefox/addon/web-developer/

Website	Page load time (ms)	% Delta	Bandwidth usage (kilobytes)	% Delta
New York Times nytimes.com	Without: **1230 ms** With: **8690 ms** In Cache: **2120 ms**	**+607%** **+72%**	Without: **806 KB** With: **345 KB**	**-57%**
Google search page google.com	Without: **980 ms** With: **2370 ms** In Cache: **930 ms**	**+142%** **-5%**	Without: **177 KB** With: **174 KB**	**-2%**
Google results page google.com/search?q=fetchy	Without: **750 ms** With: **4510 ms** In Cache: **980 ms**	**+501%** **+30.7%**	Without: **205 KB** With: **194 KB**	**-5%**
Wikipedia english home en.wikipedia.org/wiki/	Without: **1830 ms** With: **6690 ms** In Cache: **1590 ms**	**+265%** **-13%**	Without: **263 KB** With: **252 KB**	**-4%**
Youtube home page youtube.com	Without: **1750 ms** With: **13080 ms** In Cache: **1320 ms**	**+647%** **-25%**	Without: **422 KB** With: **388 KB**	**-8%**
Amazon home page amazon.com *(Note: amazon.com serves random variants of its home page on each load, making the comparison unfair)*	Without: **1570 ms** With: **4820 ms** In Cache: **700 ms**	**+207%** **-55%**	Without: **307 KB** With: **304 KB**	**-1%**
Twitter home page twitter.com *(fetchy did not load the full page)*	Without: **1290 ms** With: **8370 ms** In Cache: **980 ms**	**+549%** **-24%**	Without: **457 KB** With: **56 KB**	**-88%**
WordPress home page wordpress.com	Without: **1600 ms** With: **3390 ms** In Cache: **1130 ms**	**+112%** **-29%**	Without: **196 KB** With: **168 KB**	**-14%**
Apple Inc. home page apple.com	Without: **1010 ms** With: **4850 ms** In Cache: **960 ms**	**+380%** **-5%**	Without: **326 KB** With: **216 KB**	**-34%**
Ubuntu home page ubuntu.com *(fetchy did not load the full page)*	Without: **2180 ms** With: **3770 ms** In Cache: **1210 ms**	**+73%** **-44%**	Without: **255 KB** With: **158 KB**	**-38%**
SeaMonkey Project seamonkey-project.org	Without: **1190 ms** With: **2130 ms** In Cache: **920 ms**	**+79%** **-22%**	Without: **49 KB** With: **43 KB**	**-12%**
GINI homepage cgi.cs.mcgill.ca/~anrl/gini/	Without: **930 ms** With: **4450 ms** In Cache: **520 ms**	**+378%** **-44%**	Without: **370 KB** With: **347 KB**	**-6%**

The above data shows that fetchy always successfully shrinks down the page by a few kilobytes or more. However, the processing overhead of non-cached requests make this optimization a costly process. The caching mechanism does alleviate this problem by a large amount, often making the page load faster than it would if it were retrieved from the origin server.

The best use case of fetchy would then to be deployed on a high-capacity server just outside of the user's ISP network. If the user is on a high-latency and/or slow link, fetchy's overhead may be not noticeable or even prove faster than a regular connection. Otherwise, on modern, low-latency and fast links offered in regular consumer plans today, fetchy is worth it if the user is patient and has a relatively tight bandwidth limit (which is the case for most entry-level broadband plans).

Future work

- **Cache**
    - The cache is implemented as a 2-level (in-memory, on-disk) cache. The disk component of the cache stores (key, pristine data, compressible data) in separate files:
        - `key.prist.cache`: The pristine data, stored as-is
        - `key.raw.cache`: The compressible data, before compression is complete
        - `key.gz.cache`: The compressible data, after compression is com

      The choice for keeping these files separate is purposeful; the idea is to be able to take advantage of the Linux and BSD `sendfile`[22] system call to send over the contents of the response without moving the file's contents into process memory and back in kernel memory, with all the context switches involved.
    - The cache currently does not implement the HTTP caching correctness mechanisms (the `Cache-Control`, `ETag`, `If-Modified-Since` headers); it simply assumes all `GET` requests can be cached, and caches them with an LRU policy. For correctness, these mechanisms should be implemented.
- **Web page optimizations**
    - *JavaScript*
      The Closure Compiler is quite slow on the "`SIMPLE_OPTIMIZATIONS`" preset (the default), which is the required level to produce meaningful space savings without causing too many compatibility issues. Reimplementing the Closure Compiler within fetchy would eliminate a lot of the overhead.
      Additionally, web pages relying on tricky `<script>` placement do not work on fetchy. It is impossible to detect all of those without performing code inspection, which would slow the whole pipeline down. This is problematic, and may be worked around by maintaining a blacklist of such websites where fetchy should not modify JavaScript.

[22] http://kernel.org/doc/man-pages/online/pages/man2/sendfile.2.html

- *CSS optimizations*
  Inline style detection can be used to further optimize CSS. For instance, if several tags with the same class and style attributes are present, then the `style` attributes of each tag can be moved to a single instance in a style tag
  ```
  <div class="class1" style="background: red; border: blue;">
  …
  <div class="class1" style="background: red; border: blue;">
  ```
  becomes
  ```
  <div class="class1">
  ...
  <div class="class1">
  ```
  And the additional rule "`class1{background:red;border:blue}`" can be added to the page's global CSS code, served under the `http://fetchy/`***`key`***`.css`-style URL.
- *HTML optimizations*
  HTML can be improved in several ways. For instance:
  - Certain tags with no content can be removed (eg. `<b></b>`)
  - Default values for tags can be omitted:
    `<input type="text"/>` becomes `<input/>`
  - Removing deprecated attributes when superseder is present (This could lead to problems in older browsers)
    `<div name="name1" id="name1">`
    becomes
    `<div id="name1">`

- The proxy itself currently doesn't implement any sort of authentication mechanism, and is thus vulnerable if bound to a public-facing network interface. This can be worked around by binding fetchy to the local networking interface (`lo`) and using SSH tunnelling to contact it, but proper and built-in authentication would alleviate the need for such a heavy-handed solution.