**Multithreaded HTTP Proxy**

Disclaimer:

For building and running instructions see README.md

This document describes the structure of the code and how the proxy works.

**Project Overview**

This project implements a simple HTTP proxy server.

It's main features are:

\* Configurable blocklist for undesired web-sites

\* filtering of offensive language, offensive words specified in a text file

\* support of compressed (gzipped or deflated) HTTP streams

\* caching of results

\* detailed console logs

Main limitation is that the proxy currently doesn't support HTTPS, cookies and redirects,

which limits its use with sites like Google or Facebook (but even if not supported,

these sites can still be blocked if added to blocklist file).

If desired, the project can be easily extended to support all these features, but it will take

Additional time.

**Directory layout**

|----Makefile - allows for easy project's build

|

|--- include/ - contains projects' headers

| |- http\_utils.h - header for http\_utils.cpp and defines HttpMessage and HttpHeader types

| |- utils.h - header for utils.cpp

| |- request\_handler.h - header for request\_handler.cpp

|--- src/ - contains project's implementation files

| |- http\_utils.cpp - contains various HTTP-functions, like reading-writing HTTP messages and so on

| |- request\_handler.cpp - defines function that is executed in separate thread on each incoming request in order to process it

| |- server.cpp - contains main() function, it parses command-line options and starts the proxy

| |- utils.cpp - contains various helper functions

| |- utils\_test.cpp - contains tests for some of the functions from utils.cpp

|--- lib/ - contains external project's dependencies

| |- zlib-1.2.8/ - contains library for dealing with compressed HTTP messages

**How it works**

1. Startup

Upon startup, server.cpp parses command-line options and creates listening socket on given port.

Then it starts waiting for clients to connect.

2. A client requests an URL

The high-level logic of request processing is written in request\_handler.cpp.

For each incoming client connection, server creates a new thread. A new socket is created for this connection and the new thread communication with the client through this socket.

When a client wants to access some server, say, "www.hyperhero.com/en/insults.htm"

he enters in his browser's URL the following:

<PROXY\_ADDRESS>:<PROXY\_PORT>/www.hyperhero.com/en/insults.htm

For example, if we run the proxy on localhost like this:

PROJECT\_ROOT/bin/server 8888 ./blocklist.txt ./filter\_words.txt ./cache

then in order to access "www.hyperhero.com/en/insults.htm" we enter in browser's URL:

localhost:8888/www.hyperhero.com/en/insults.htm

The thread receives following HTTP request line:

GET /www.hyperhero.com/en/insults.htm HTTP/1.1

It then parses the request path and splits it in two components:

"www.hyperhero.com" + "/en/insults.htm"

Then it creates a copy of original client's request, changes its "Host" header and

request param like so:

GET /en/insults.htm HTTP/1.1

Host: www.hyperhero.com

... other headers...

and sends the modified request to the target server, which is the same as "Host" header.

After receiving the reply from the target server, and possible uncompressing it with the help of zlib library, it filters out offensive words, using a text file specified in command-line options.

Then, the thread checks if the reply can be cached or not, it depends on the header "Cache-Control".

If it can be cached, the thread will save the cached copy of the reply into a file with random name, and will store in a std::map that a given request string is cached in a given file.

If some other client will access the same url later, the server will fetch the reply from the cache.

A comment here: since many threads operate on one common cache, a synchronization is needed here - it is achieved with the help of a mutex (see the request\_handler.cpp file for details).